RA78K/I ASSEMBLER PACKAGE USER'S MANUAL for LANGUAGE

NEC Corporation

Phase-out/Discontinued

Errata

The followings are corrections in the RA78K/I Assembler Package User's Manual for Language:

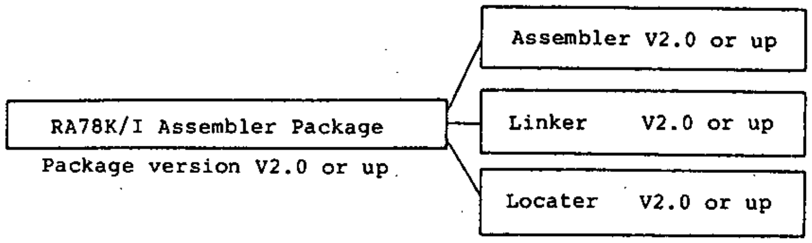
Page	For	Read
i,[Target Device]	uPD78122,uPD78124,uPD78P124	Expurgation
iv,"Assembler related"	EEM-	EEU-1205
iv,"Debugger related"	IE-78120-R ~ (PC-DOS) based	Expurgation
	EEM-1047	Expurgation
1-14,Table 1-1	PC-9800 series	IBM PC
	MS-DOS	PC-DOS
A-15~17	Table C-2	Expurgation
A-18,line 3~4	For the uPD78122 ~ to read as "addr16"	Expurgation
A-28,Last	$0\underline{110}\ 0\underline{001}$	$0\underline{001}\ 0\underline{110}$
	A register	L register
	L register	A register

RA78K/ I ASSEMBLER PACKAGE USER'S MANUAL for LANGUAGE

NEC Corporation

INTRODUCTION

This manual is designed to facilitate correct understanding of the basic functions of each program in the RA78K/I Assembler Package (hereinafter referred to as "this package or the package") and the methods of describing source programs for the RA78K/I.
This manual does not cover how to operate the respective programs of the RA78K/I assembler package. Therefore, after you have comprehended the contents of this manual, read the RA78K/I Assembler Package User's Manual for Operation (hereinafter referred to as "the Operation Manual") to operate each program in the assembler package.
This manual is applicable to the package product versions V2.0 and upwards of the RA78K/I assembler package.



[Target Devices]
The software of the following microcomputers can be developed with this package:

- o uPD78112, uPD78P112
- o uPD78122, uPD78124, uPD78P124

[Readers of Manual]
Although this manual is intended for those who are familiar with the functions and instructions of the microcomputer subject to software development, the manual can also be used by those who use an assembler program for the first time.

[Organization of Manual]
This manual consists of the following six chapters and appendixes:

Chapter 1 - General
Outlines the functions of this package including the role of the package in microcomputer development.

Chapter 2 - How to Describe Source Programs
Describes the general rules applicable to the description of a source program such as the basic configuration and description format of source programs, and expressions and operators of the assembler.

Chapters 3 & 4 Directives and Control Instructions
Details the description format, function, and use of each of the assembler directives and control instructions, including application examples.

Chapter 5 - Macros
Outlines macro functions such as macrodefintion, macro reference (macrocall), and macroexpansion.
Macro directives are explained in Chapter 3.

Chapter 6 - Product Utilization
Introduces some measures recommended for effective utilization of this package.

Appendixes
Contain a list of instructions (uCOM-78K/I), a list of directives and control instructions, a list of reserved words, a list of hints on use and restrictions, etc.
NOTE: The uCOM-78K/I instruction set is not detailed in this manual. For these instructions, refer to the user's manual of each microcomputer subject to software development.

[Recommended Usage of Manual]

For those who use an assembler for the first time: Read from Chapter 1, General of this manual.

For those who have a general understanding of assembler programs: You may skip Chapter 1, General of this manual. (However, it is advisable to read Section 1.3, "Reminders Before Program Development".)

Source programs for the uCOM-78K/I can be described in several different ways. Be sure to read Chapter 2, "How to Describe Source Programs".

For those which wish to know the directives and control instructions of the assembler: Read Chapters 3 and 4, respectively, because the format, function, use, and application examples of each directive or control instruction are detailed in these chapters.

The uCOM-78K/I instructions are listed in the respective appendixes. Use the lists for quick reference.

[Symbols and Abbreviations]

The following symbols and abbreviations are used in this manual:

Symbol	Meaning
...	Continuation (repetition) of data in the same format
[ ]	Parameter(s) in brackets can be omitted.
" "	Characters enclosed in " " (double quotes) must be input as is.
( )	Characters enclosed in parentheses must be input as is.
< >	Characters enclosed in < > must be input as is (or indicates a title).
——	Important point
␣	One or more blanks (spaces) must be input:
△	Indicates one blank (space).
⋮	This part of the program description is omitted.

[References]

The documents related to this manual are shown below.

Document name	Document No.
o Assembler related	
RA78K/I Assembler Package User's Manual for Operation - PC-9800 series (MS-DOS) and IBM PC (PC-DOS) based	EEM-
o Debugger related	
IE-78112-R User's Manual Volume I - Hardware	EEP-1026
IE-78112-R User's Manual Volume II - Software	EEM-1023
IE-78112-R System Software Operation Manual IBM PC (PC-DOS) based	EEM-1028
IE-78120-R System Software User's Manual IBM PC (PC-DOS) based	EEM-1047
o Device related	
uPD78112 User's Manual	IEM-1136
Product Catalog for D78112CW/GF	IP-1091

NOTE: Contact your nearest NEC dealer for the latest information on the reference documentation.

TABLE OF CONTENTS

ILLUSTRATIONS AND TABLES

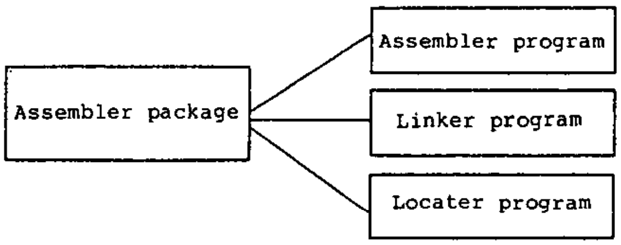
CHAPTER 1. GENERAL

1.1 Assembler Overview

The RA78K/I Assembler Package is a series of programs designed to translate a source program coded in the assembly language of the uCOM-78K/I series microprocessors, into machine language coding.

The assembler package contains three programs: Assembler, Linker, and Locater.

Fig. 1-1. Assembler Package



1.1.1 What is an assembler?

(1) Assembly language and machine language

An assembly language is the most fundamental programming language for microprocessors.

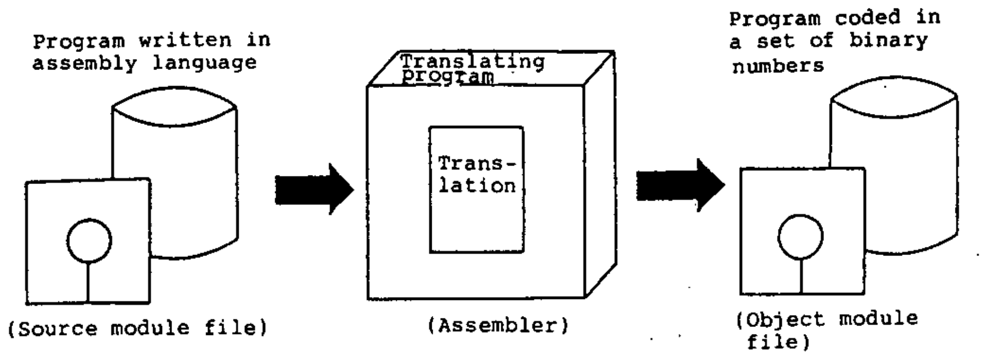
To have a microprocessor do its job, programs and data are required. These programs and data must be written by a human being (i.e., a programmer) and stored in the memory section of a microcomputer. Programs and data that can be handled by the microcomputer is nothing but a set or combinations of binary numbers which is called machine language (i.e., the language that can be understood or interpreted by the computer).

To create a program in machine language coding, namely, by a set of binary numbers is not an easy job for a human being, because it's difficult for him or her to remember the coding and he or she is likely to make errors in coding.

For this reason, there is a method of creating a program using an abbreviated symbol (or mnemonic symbol) which represents the meaning of a machine language instruction to assist the human memory. A programming language system by this symbolic coding is called an assembly language.

To translate a program created in the assembly language into a set of binary numbers that can be understood by the microprocessor, another program is required. This program is called an assembler.

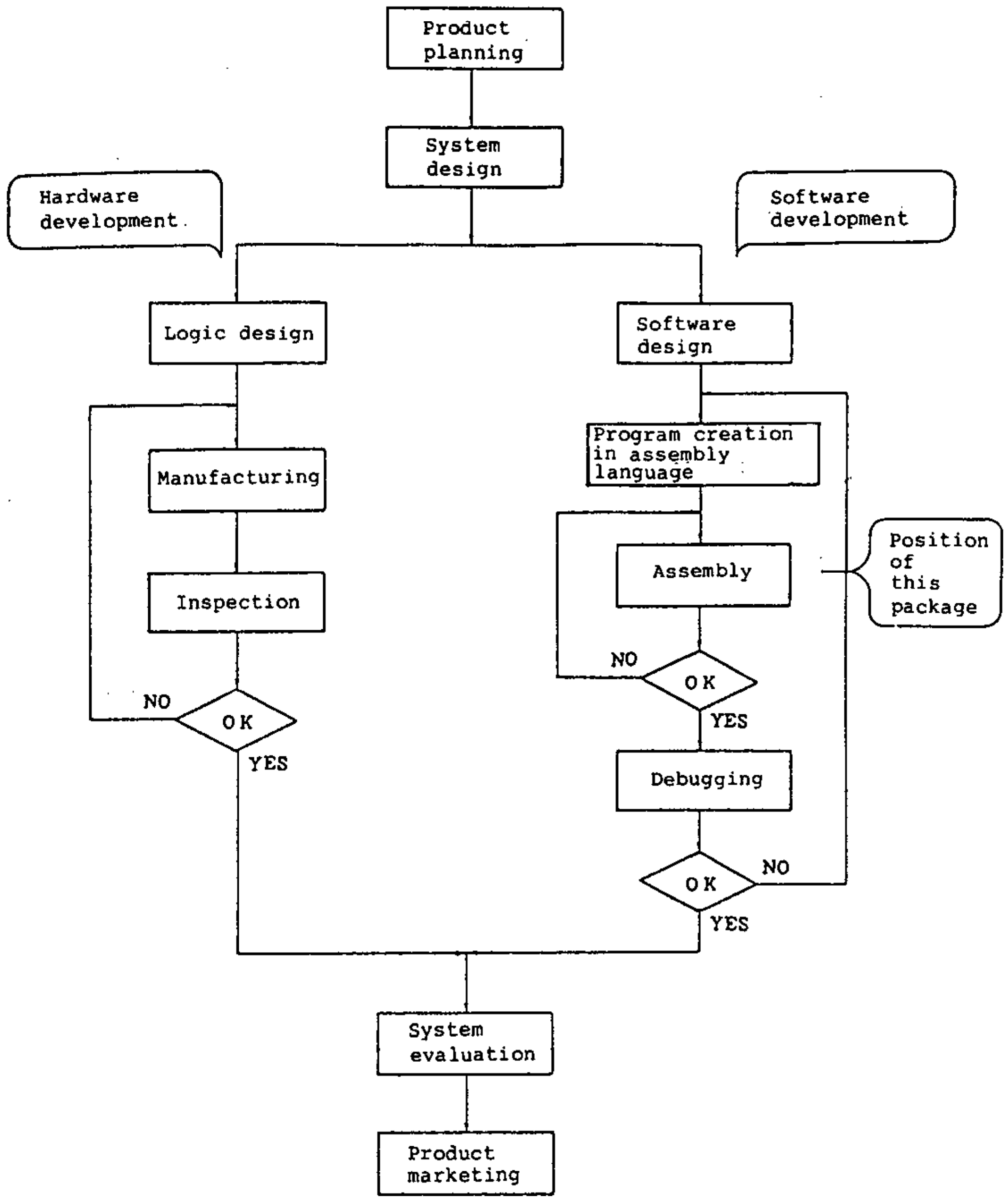
Fig. 1-2. Flow of Assembler



(2) Development of microcomputer-applied products and role of this package

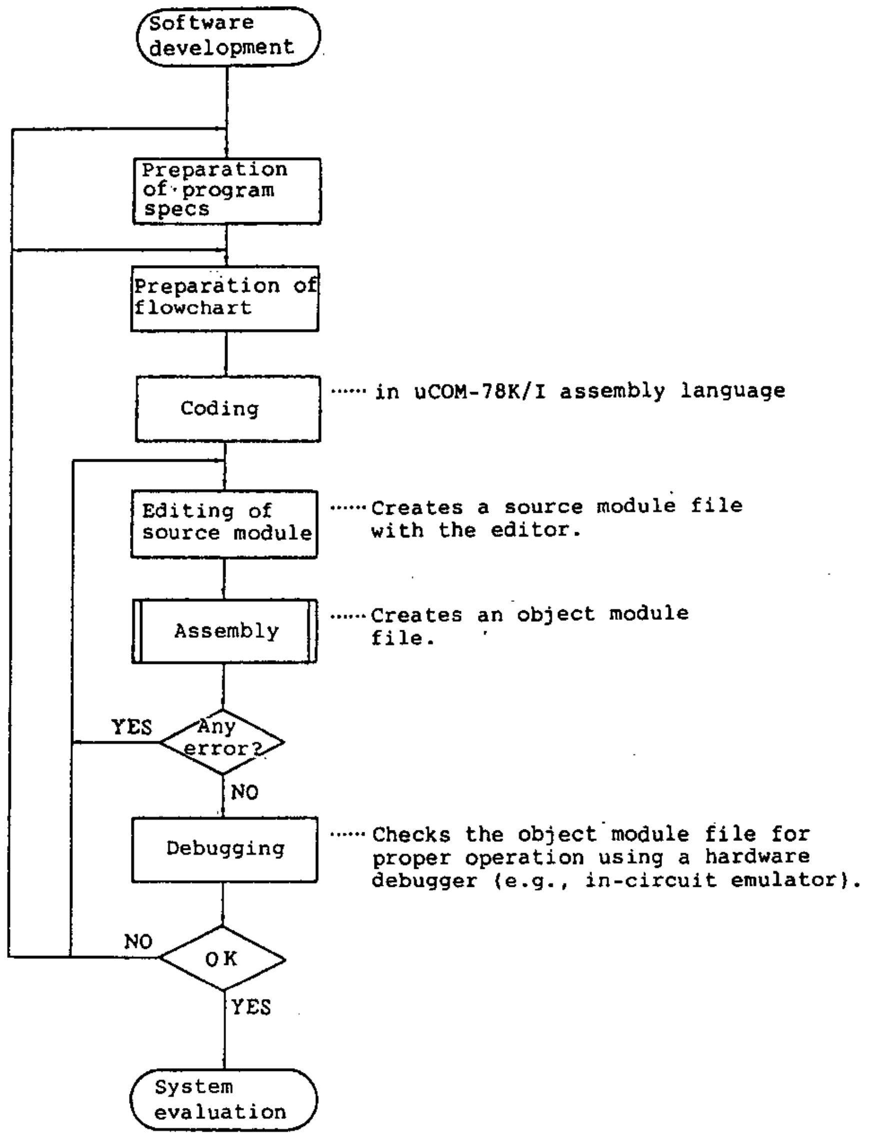
Fig. 1-3 illustrates the standing of the programming in assembly language in the development process of microcomputer-applied product.

Fig. 1-3 Development Process of Microcomputer-applied Product



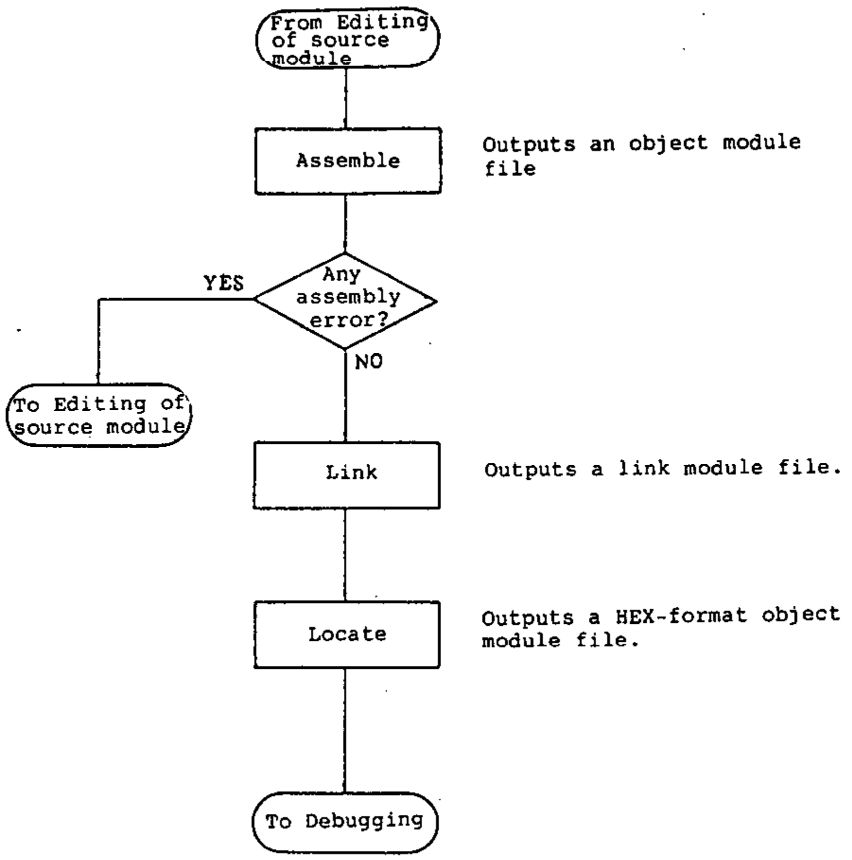
The software development process will be further detailed in Fig. 1-4 on the next page.

Fig. 1-4. Software Development Process


Software development
Preparation of program specs
Preparation of flowchart
Coding
...... in uCOM-78K/I assembly language
Editing of source module
...... Creates a source module file with the editor.
Assembly
...... Creates an object module file.
YES
Any error?
NO
Debugging
...... Checks the object module file for proper operation using a hardware debugger (e.g., in-circuit emulator).
NO
OK
YES
System evaluation

The assembly phase in the software development process will be reviewed in further detail by giving an example of this package.

Fig. 1-5. Assembly Phase by This Package


From Editing of source module
Assemble
Outputs an object module file
Any assembly error?
YES
NO
To Editing of source module
Link
Outputs a link module file.
Locate
Outputs a HEX-format object module file.
To Debugging

1.1.2 What is a relocatable assembler?

The machine language translated from a source language by the assembler will be stored in the memory of the microcomputer before use. In this case, in which memory location each machine language instruction will be stored must have been determined. Therefore, information on "the allocation of each machine language instruction to a specific address in memory" will be added to the machine language converted by the assembler.

Depending on the method of allocating addresses to machine language instructions, an assembler can be broadly divided into an absolute assembler and a relocatable assembler.

- o Absolute assembler

  Allocates the machine language instructions converted in one-time assembly operation to absolute addresses.
- o Relocatable assembler

  Addresses determined for the machine language instructions converted in one-time assembly operation are tentative. Absolute addresses will be determined by a program called the locater.

In the past, when a program was created with the absolute assembler, programmers had to, as a rule, complete programming at a time. However, if you create a large program at a time, the program becomes complicated, making analysis and maintenance of the program troublesome. To avoid this, such a large program is developed by dividing it into several subprograms (i.e., modules) for each functional unit. This programming technique is called the modular programming.

The relocatable assembler is an assembler suitable for modular programming. The following advantages can be derived from modular programming with the relocatable assembler:

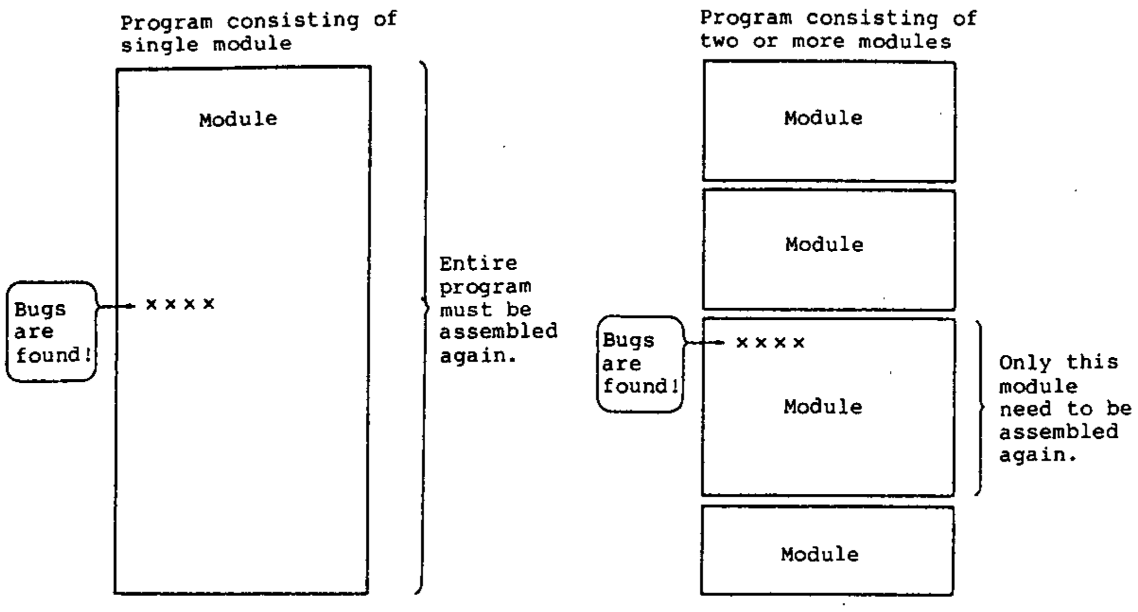
(1) Increase in development efficiency

It's difficult to write a large program at a time.

In such a case, divide the program into modules for each function and the program can be developed with two or more programmers engaged in writing subprograms at the same time. This will certainly increase development efficiency of the program.

If any bugs are found in the program, you do not need to re-assemble the entire program just to correct part of the program. Only the subprogram (module) requiring correction(s) can be re-assembled. This will help shorten the debugging time.

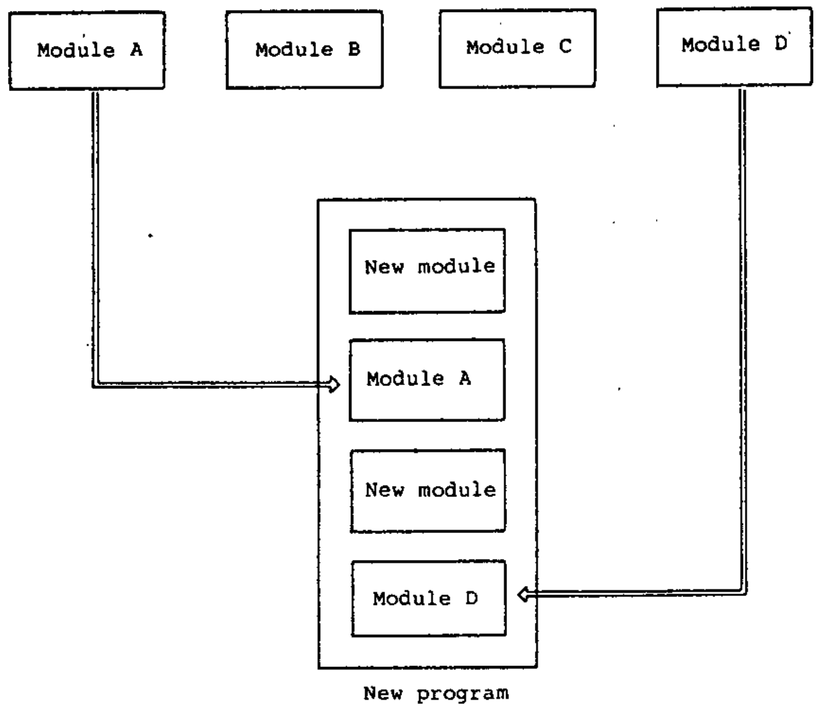
Fig. 1-6. Re-assembly for Debugging


Program consisting of single module
Module
Bugs are found!
x x x x
Entire program must be assembled again.
Program consisting of two or more modules
Module
Module
Bugs are found!
x x x x
Module
Only this module need to be assembled again.
Module

(2) Utilization of resources

Highly reliable, highly versatile modules which have been previously created can be utilized for creation of another program. If you accumulate such high-versatility modules as software resources, you can save time and labor in developing a new program.

Fig. 1-7. Program Development Utilizing Existing Modules

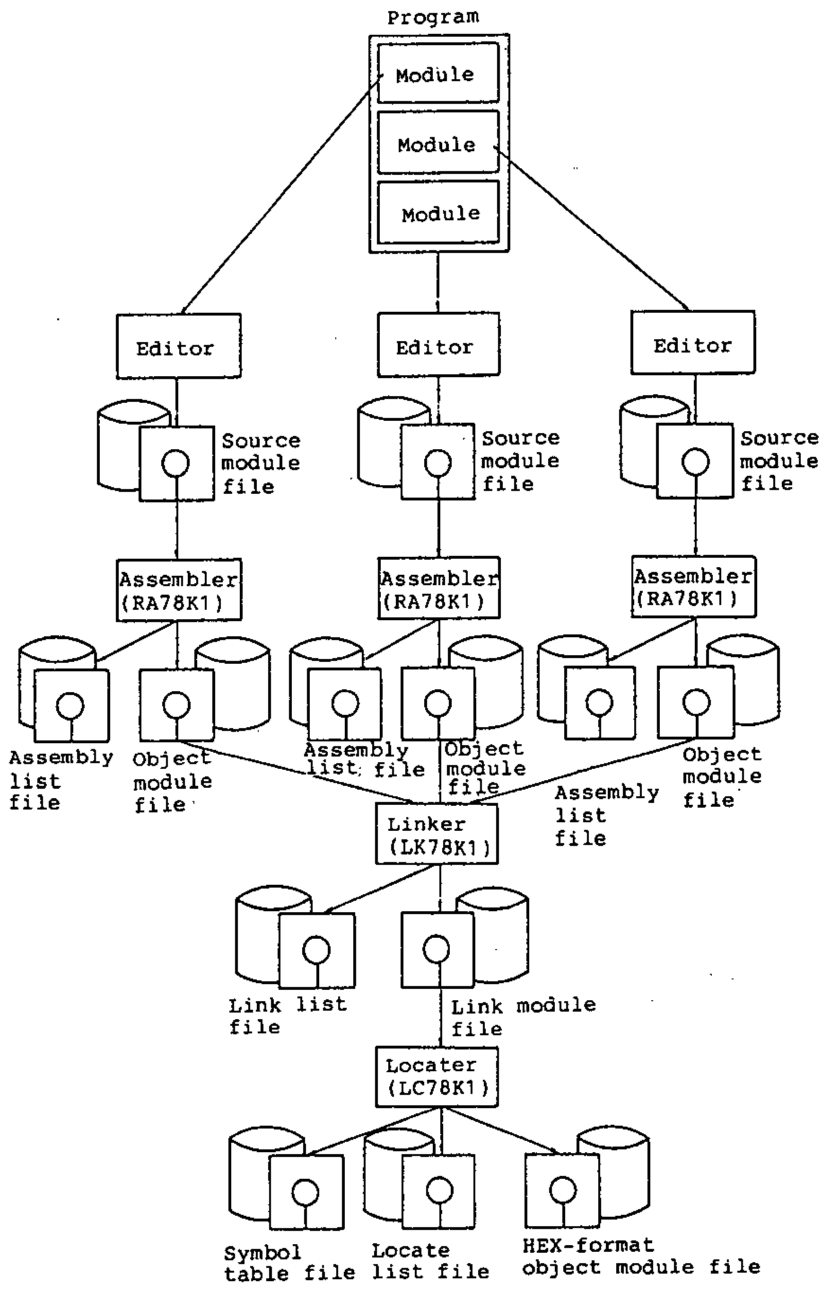


1.2 Functional Outline of Assembler Package

An ordinary program development procedure with this assembler package is illustrated in Fig. 1-8. The development of a program is basically performed by using the assembler, linker, and locater.

In the following discussions, the assembler, linker, and locater programs are collectively referred to as "the assembler package or this package" and the assembler program, as "the assembler".

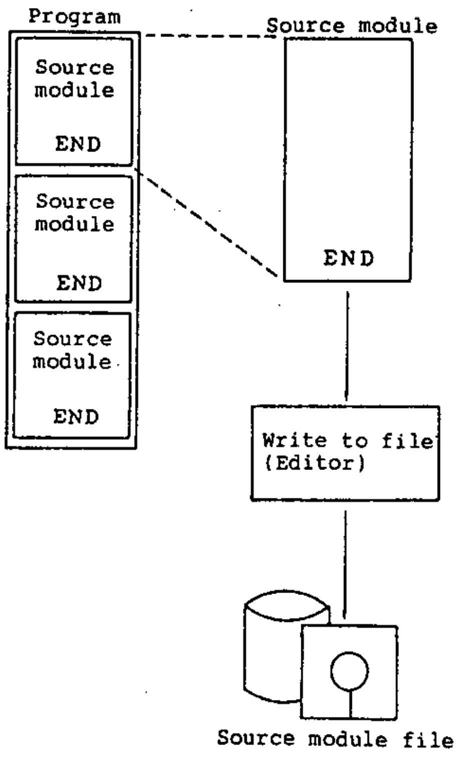
Fig. 1-8. Program Development Procedure with This Package



1.2.1 Creation of source module file with editor

Devide one program functionally into several modules.
Each module becomes the unit of coding as well as the unit of input to the assembler. A module serving as the unit of input to the assembler is called a source module.
After coding each source module, the source module is written into a file with the editor. The file thus created is called a source module file.
The source module file becomes an input file to the assembler (RA78K1).

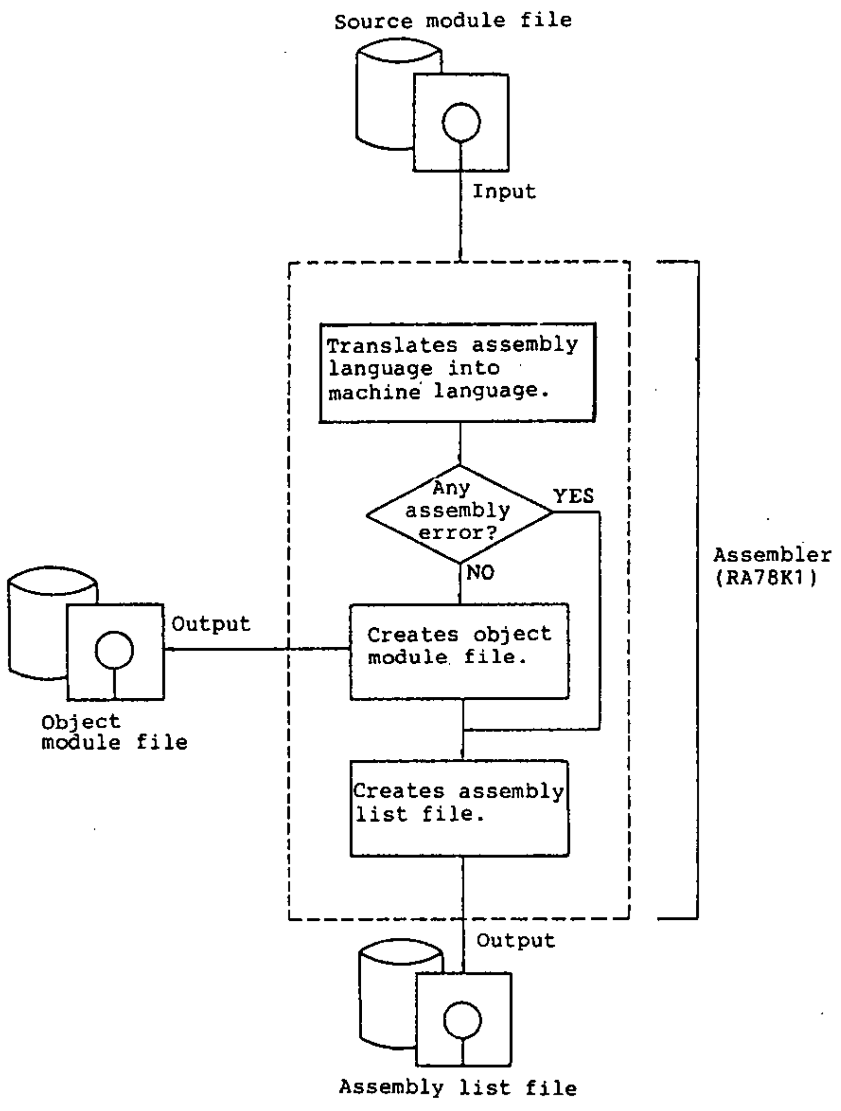
Fig. 1-9. Creation of Source Module File



1.2.2 Assembler

The assembler accepts source module files as input files and translates assembly language into machine language.
If any coding error is found in the input source module, the assembler outputs an assembly error. If no assembly error is found, the assembler outputs an object module file which contains machine language information and relocation information relating to the allocation address of each machine language instruction.
The assembler also outputs information at assembly time as an assembly list file.

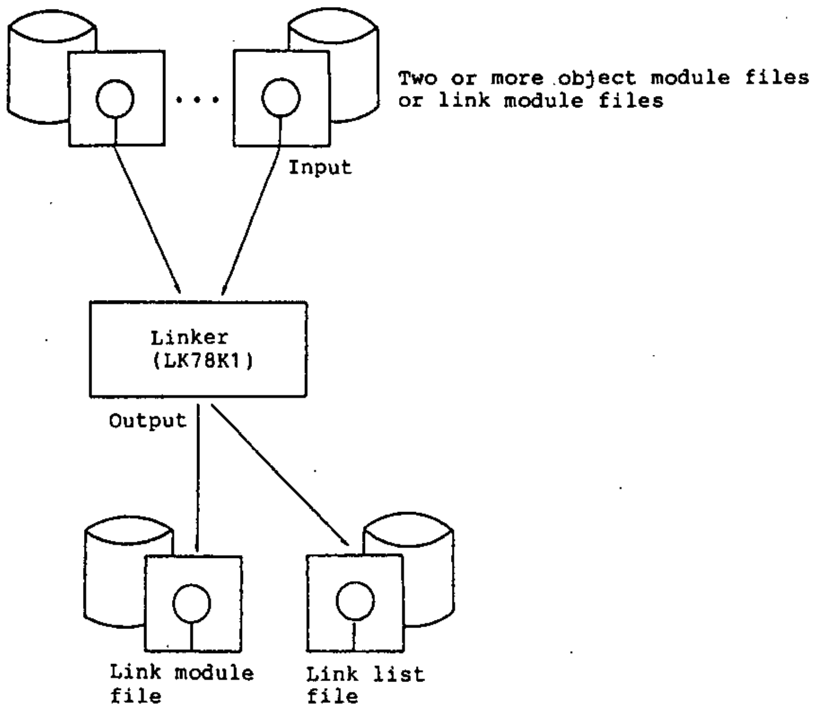
Fig. 1-10. Functions of Assembler



1.2.3 Linker

The linker accepts two or more object module files output by the assembler or link module file output by the linker itself as input files and gathers them for output as a single link module file. The linker also outputs information at link time as a link list file.

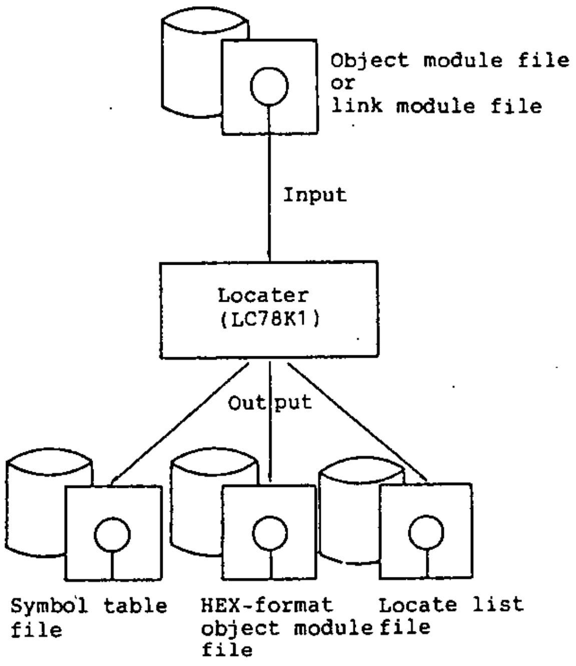
Fig. 1-11. Functions of Linker



1.2.4 Locater

The locater accepts the object module file output by the assembler or the link module file output by the linker as an input file and determines the addresses to be allocated to machine language instructions and outputs the result of the locate operation as a HEX-format object module file.
The locater also outputs the symbol information required in symbolic debugging with an in-circuit emulator as a symbol table file and the allocation address information at locate time as a locate list file.

Fig. 1-12. Functions of Locater



The processing of this package terminates when the processes up to the locater have been completed normally.

1.3 Reminders Before Program Development

Before you set your hand to the development of a program, keep in mind the following points:

1.3.1 Size of source module file

The size of a source module file that can be input to the assembler is limited to one of the following values depending on the operating (or execution) environment of the assembler.

Table 1-1. Size of Source Module File

Host machine	OS	Size of source module file that can be input
MD-086/116 series	Concurrent CP/M	64K bytes max.
PC-9800 series	MS-DOS	64K bytes max.
	CP/M-86	
VAX-11 series	VMS	Approx. 850K bytes max.

With a source module file having a size of 64K bytes, assuming that each line of the source module consists of an average of 30 characters, a program of up to about 2K steps can be assembled as a single source module file.

1.3.2 Number of files than can be input to Linker

The number of object module files and/or link module files that can be input to the linker is 100.

1.3.3 Restriction on number of symbols

The number of local symbols and that of PUBLIC symbols, which can be defined in the assembler, linker, and locater, respectively, are restricted as shown in the table below.

	Number of symbols	
	No. of local symbols	No. of PUBLIC symbols
Assembler	Approx. 1,800	256
Linker	1,800 x No. of modules	Approx. 2,000
Locater	1,800 x No. of modules	Approx. 2,000

NOTE: If any PUBLIC symbols have been defined, the number of local symbols that can be defined in the assembler must be reduced to the following value:

No. of local symbols that can be used in Assembler
= Approx. 1,800 - No. of PUBLIC symbols defined

1.3.4 Maximum performance characteristics of assembler package

The maximum performance characteristics of the assembler package that should be kept in your mind before program development are listed in the following tables.

(1) Maximum performance characteristics of Assembler

Item	Restriction
Symbol length	6 characters
No. of characters per line	99 characters
No. of code segments per type	1 segment
No. of absolute segments	10 segments
No. of macrodefinitions	10 definitions

(2) Maximum performance characteristics of Linker

Item	Restriction
No. of input module files	100 files
No. of different segment names	255 names
No. of absolute segments	100 segments

(3) Maximum performance characteristics of Locater

Item	Restriction
No. of relocatable segments in the input module	256 segments

1.4 Features of Assembler Package

This package has the following features:

(1) Macro function
When the same group of instructions must be described in a source program over and over again, a macro can be defined by giving a single macro name to the group of instructions.
By using this macro function, coding efficiency and readability of the program can be increased.

(2) Optimize function of branch instructions
The assembler package has an assembler directive to automatically select a branch instruction (i.e., BR directive).
To create a program with high memory efficiency, a 2-byte branch instruction must be described according to the branch destination range of the branch instruction. However, it is troublesome for the programmer to describe a branch instruction by paying attention to the branch destination range for each branching. If the BR directive is described, the assembler generates the appropriate branch instruction according to the branch destination range. This is called the optimize function of branch instructions.

(3) Conditional assembly function
With this function, part of a source program can be specified for assembly or non-assembly according to a predetermined condition. If a debug statement is described in a source program, whether or not the debug statement should be translated into machine language can be selected by setting a switch for conditional assembly. When the debug statement is no longer required, the source program can be assembled without major modifications to the program.

CHAPTER 2. HOW TO DESCRIBE SOURCE PROGRAMS

2.1 Basic Configuration of Source Program

When a source program is described by dividing it into several modules, each module which becomes the unit of input to the assembler is called a source module. (If a source program consists of only one module, the source program means the same as the source module.)

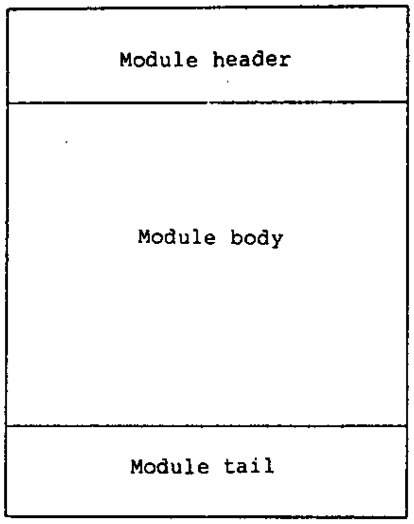
Each source module which becomes the unit of input to the assembler consists mainly of the following three parts:

(1) Module header

(2) Module body

(3) Module tail

Fig. 2-1. Configuration of Source Module



2.1.1 Module header

In the module header, five different items can be described as shown in Table 2-1 below.

Table 2-1. Items That Can Be Described in Module Header

Item No.	Item that can be described	Explanation	Chapter/section in this manual
1	Assembler options	Assembler option(s) which are normally specified in the start-up command line of the assembler can be described before NAME directive in the module header.	See 4, "Control Instructions".
2	NAME directive	This directive can be described only in the module header and must always be described.	See 3.5, "Linkage directives".
3	PUBLIC, EXTRN, or EXBIT directive	These directives can be described only in the module header.	See 3.5, "Linkage directives."
4	EQU or SET directive	These directives can be described after PUBLIC, EXTRN, or EXBIT directive in the module header.	See 3.3, "Symbol definition directives."
5	Control instructions	Control instructions can be described after NAME directive in the module header.	See 4, "Control Instructions."

NOTE: These items must be described in the module header in the order of item numbers. (Namely, assembler option(s) (Item No. 1) must be described before the NAME directive (Item No. 2). Comment(s) can be described anywhere after the assembler option(s) in the module header and control instruction(s) can be described after the NAME directive in the module header.

2.1.2 Module body

In the module body, the following items cannot be described:

- o Assembler options
- o NAME directive
- o PUBLIC, EXTRN, and EXTBIT directives

All other directives, control instructions, and uCOM-78K/I instructions can be described in the module body.

The module body must be described by dividing it into units each called a segment.

The user may define the following four segments with a directive corresponding to each segment:

(1) Code segment .... Must be defined with the CSEG directive.
(2) Data segment .... Must be defined with the DSEG directive.
(3) Bit segment .... Must be defined with the BSEG directive.
(4) Absolute segment ... Must be defined with the ORG directive.

The module body may be configured with any segment combinations, provided a data segment and a bit segment must be defined before a code segment.

2.1.3 Module tail

The module tail indicates the end of the source module. The END directive must be described in this part.

2.1.4 Overall configuration of source program

The overall configuration of a source module becomes as shown below.

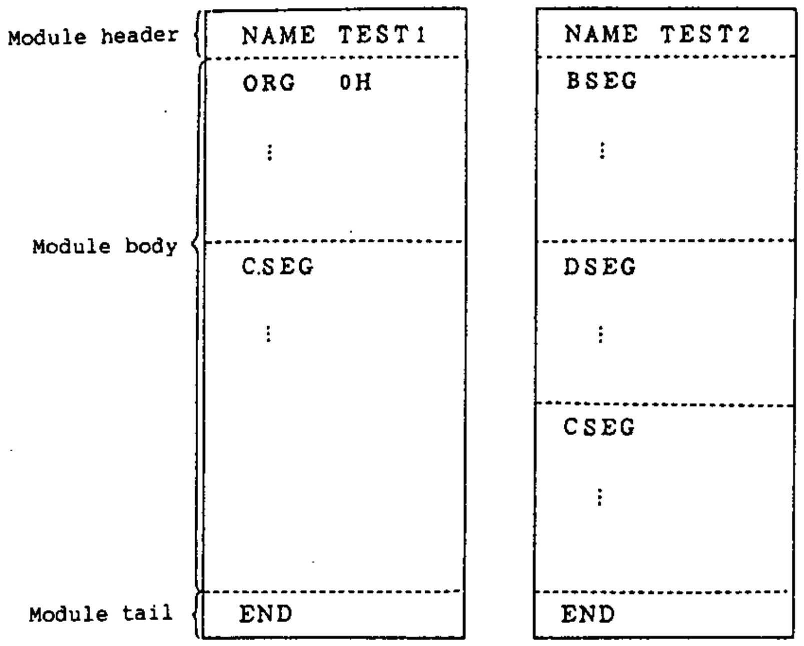
Fig. 2-2. Overall Configuration of Source Program

Assembler option(s) $\boxed{\text{NAME directive}}$ EQU and SET directives PUBLIC, EXTRN, and EXTBIT directives Control instruction(s)	Module header
Directives (other than NAME, PUBLIC, EXTRN, EXTBIT) Control instruction(s) Instruction(s)	Module body
$\boxed{\text{END directive}}$	Module tail

NOTE: $\boxed{}$ indicates that the directive must always be described.

Examples of simple source module configurations are shown in Fig. 2-3 on the next page.

Fig. 2-3. Examples of Source Module Configurations

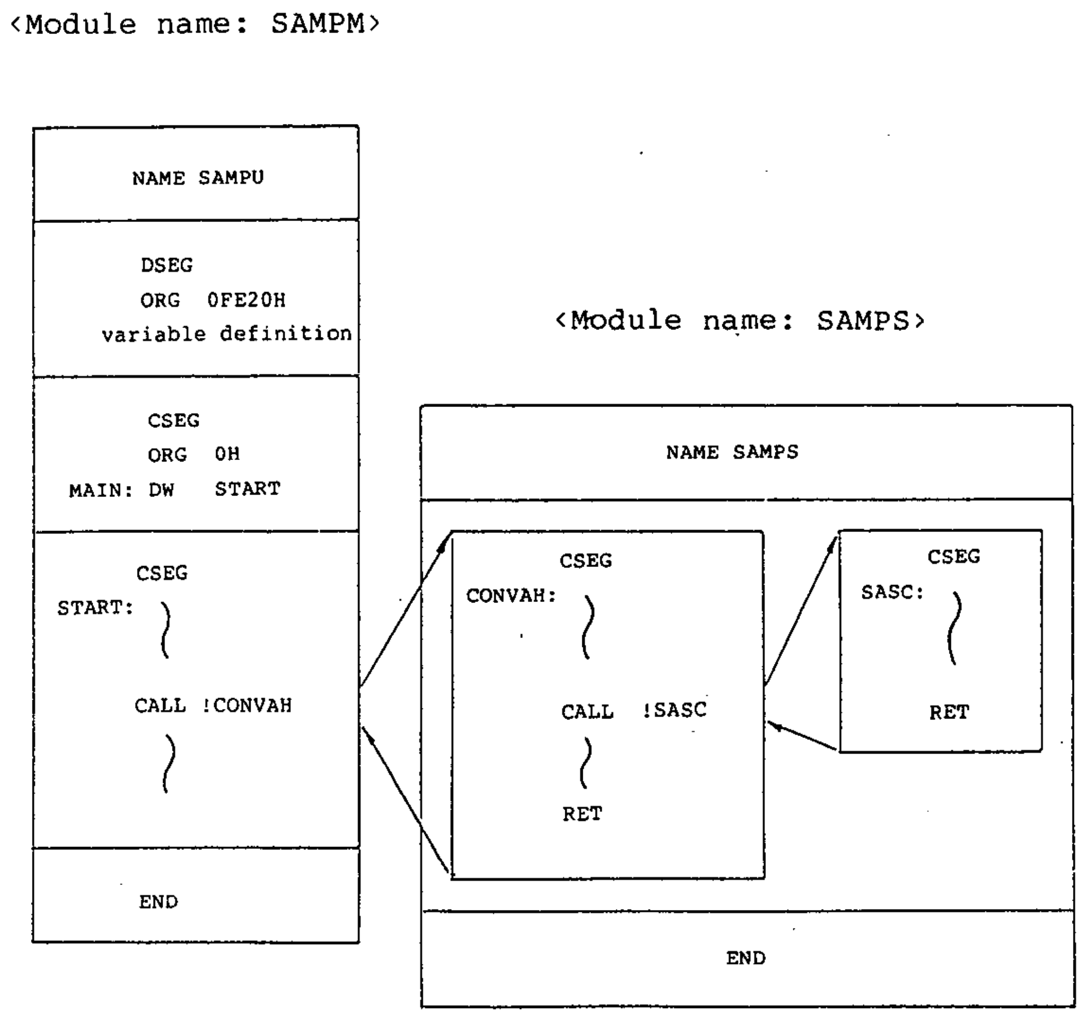


2.1.5 Description example of source program

In this subsection, a description example of a uCOM-78K/I source program is shown, in the hope that you can have a general idea of how to describe a source module. (This example is attached to the package product as a sample program file.)

The configuration of the sample program can be illustrated simply as follows:

Fig. 2-4. Configuration of Sample Program



This sample program was created by dividing a single source program into two modules. The module "SAMPM" is a main routine of this program and the module "SAMPS", a subroutine which is to be called within the main routine.

<Main routine>

```
$       PROCESSOR(112)                                          ;(1)

        NAME    SAMPM                                           ;(2)
;************************************************
;*                                              *
;*      HEX -> ASCII Conversion Program         *
;*                                              *              } Module header
;*              main-routine                    *
;*                                              *
;************************************************

        PUBLIC  MAIN, START                                     ;(3)
        EXTRN   CONVAH                                          ;(4)

        DSEG                                                    ;(5)
        ORG     0FE40H
HDTSA:  DS      1
STASC:  DS      2

        CSEG                                                    ;(6)
        ORG     0H                                              ;(7)
MAIN:   DW      START

        CSEG                                                    ;(8)
START:  MOV     SP, #0E0H
        MOV     MM, #00
        MOV     STBC, #00
                                                                   } Module body
        MOV     HDTSA, #1AH
        MOVW    HL, #HDTSA              ;set hex 2-code data in HL register

        CALL    !CONVAH                 ;convert ASCII <- HEX
                                        ;output BC-register <- ASCII code
        MOVW    HL, #STASC              ;set HL <- store ASCII code table
        MOV     A, B
        MOV     [HL], A
        INC     L
        MOV     A, C
        MOV     [HL], A

        BR      $$

        END                                                     ;(9)   } Module tail
```

(1) Assembler option
(2) Declaration of a module name
(3) Declaration of a symbol referenced from another module as an external definition symbol
(4) Declaration of a symbol defined in another module as an external reference symbol
(5) Declaration of the start of a data segment
(6) Declaration of the start of a code segment
(7) Declaration of the start of an absolute segment from address 0H
(8) Declaration of the start of the code segment (meaning the end of the absolute segment)
(9) Declaration of the end of the module

<Subroutine>

```
$       PROCESSOR(112)                                              ;(10)

        NAME    SAMPS                                               ;(11)
;*****************************************************
;*                                                   *
;*   HEX -> ASCII Conversion Program                 *
;*                                                   *
;*               sub-routine                         *
;*                                                   *
;*   input condition  : (HL) <- hex 2 code           *
;*                                                   *
;*   output condition : BC-register <-ASCII 2 code   *
;*                                                   *
;*****************************************************

        PUBLIC  CONVAH                                              ;(12)

        CSEG                                                        ;(13)
CONVAH: MOV     A,[HL]          ;load hex code -> Acc
        SHR     A,4             ;hex upper code load
        CALL    !SASC
        MOV     B,A             ;store result

        MOV     A,[HL]          ;load hex code -> Acc
        AND     A,#0FH          ;hex lower code load
        CALL    !SASC
        MOV     C,A             ;store result

        RET

;********************************************************
;* subroutine    convert ASCII code                     *
;*      input    Acc (lower 4bits) <- hex code          *
;*      output   Acc             <- ASCII code          *
;********************************************************

        CSEG
SASC:   CMP     A,#0AH          ;check hex code > 9
        BC      $SASC1
        ADD     A,#07H          ;bias(+7)
SASC1:  ADD     A,#30H          ;bias(+30)
        RET

        END                                                         ;(14)
```

Module header

Module body

Module tail

(10) Assembler option
(11) Declaration of a module name
(12) Declaration of a symbol referenced from another module as an external definition symbol
(13) Declaration of the start of a code segment
(14) Declaration of the end of the module

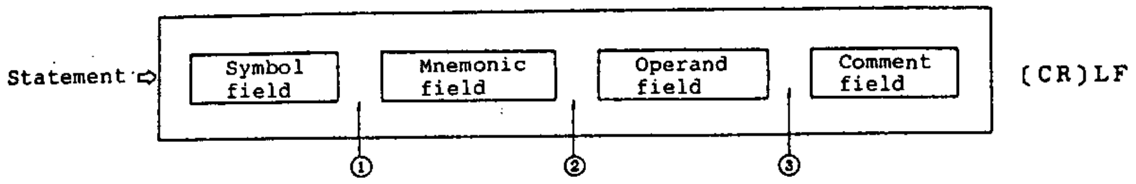
2.2 Description Format of Source Program

2.2.1 Configuration of statement

A source program consists of statements.

Each statement consists of the four fields shown in Fig. 2-5.

Fig. 2-5. Fields That Make Up A Statement



① The Symbol field and the Mnemonic field must be separated from each other with a colon (:) or one or more blank (or TAB) characters.
② The Mnemonic field and the Operand field must be separated from each other with one or more blank (or TAB) characters. Depending on the instruction described in the Mnemonic field, the Operand field may not be required.
③ The Comment field if used must be preceded with a semicolon (;).

A statement must be described within a line. (A line must be terminated with a LF (0AH) code.)

Up to 99 characters can be described per line.

The following lines may also be described:

- o Dummy line (a line without statement description)
- o Line consisting of the Symbol field alone
- o Line consisting of the Comment field alone

2.2.2 Character Set

Use the following alphabetic, numeric, and special characters to describe statements.

(1) Alphabetic characters

A	B	C	D	E	F	G	H	I	J	K	L	M	N
O	P	Q	R	S	T	U	V	W	X	Y	Z		
a	b	c	d	e	f	g	h	i	j	k	l	m	n
o	p	q	r	s	t	u	v	w	x	y	z		

NOTE 2-1

When any lowercase letter is used in a symbol or reserved word description, the lowercase letter is interpreted as its uppercase equivalent.

(2) Numeric characters

0	1	2	3	4	5	6	7	8	9

(3) Special characters

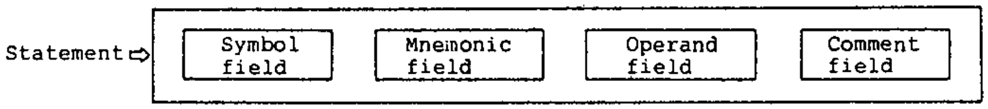
Character	Name	Main use
?	Question mark	Symbol equivalent to alphabetic characters
@	Unit price symbol	Symbol equivalent to alphabetic characters
_	Underscore	Symbol equivalent to alphabetic characters
Blank		Delimiter of each field
HT (09H)	TAB code	Character equivalent to Blank
,	Comma	Delimiter between the 1st and 2nd operands
:	Colon	Delimiter between the Symbol and Mnemonic fields
;	Semicolon	Symbol indicating the start of the Comment field
CR (0DH)	Carriage return code	Symbol indicating the end of a line
LF (0AH)	Line feed code	Same as above
+	Plus sign	ADD operator or positive sign
-	Minus sign	SUBTRACT operator or negative sign
*	Asterisk	MULTIPLY operator
/	Slash	DIVIDE operator
.	Period	BIT operator
( )	Left and right parentheses	Symbols specifying the order of arithmetic operations to be performed
'	Single quotation mark	Symbol indicating the start or end of a character constant

Character	Name	Main use
$	Dollar sign	o Symbol indicating the location counter o Symbol indicating the start of an assembler option o Symbol specifying a relative addressing mode
#	Sharp sign	Symbol specifying an immediate addressing mode
!	Exclamation point	o Symbol specifying an absolute addressing mode o Symbol specifying the operand representation format addr16 of a MOV instruction
[ ]	Braces	o Symbol specifying an indirect addressing mode

2.2.3 Fields of Statement

The respective fields that make up a statement are detailed in this subsection.

(1) Symbol field



A symbol is described in the Symbol field. The term "symbol" refers to a name given to a numerical data or address.
By using symbols, the contents of a source program can be understood more easily.
The types and attributes of symbols and the conventions of symbol description are explained below.

[Symbol types]

Symbols are available in the types shown in Table 2-2, depending on their use and method of definition.

Table 2-2. Symbol Types

Symbol type	Use	Method of definition
Name	Used as a numerical data in a source program.	This type is described in the Symbol field of the EQU, SET, or DBIT directive, or in the Operand field of the EXTBIT directive.
Label	Used as an address data in a source program.	This type is described in the Symbol field of an instruction or the DB, DW, or DS directive. <u>A colon (:) must be used as a delimiter.</u> This type is also described in the Operand field of the EXTRN directive.
Segment name	Used as a segment name subject to operation in a locater option	This type is described in the Symbol field of the CSEG, DSEG, BSEG, or ORG directive.
Module name	Used as a module name in symbolic debugging	This type is described in the Operand field of the NAME directive.
Macro name	Used as a macro name for macro reference in a source program.	This type is described in the Symbol field of the MACRO directive.

[Conventions of symbol description]

All symbols must be described according to the following rules:

① A symbol must be made up of alphanumeric characters and special characters (?, @, and _) that can be used as a symbol in a manner equivalent to alphabetic characters.
The first character of the symbol must always be one of the alphabetic characters or special characters (?, @, and _).

② A symbol must be made up of not more than six characters.
If a symbol consisting of seven or more characters is described, an error will result.

③ No reserved word can be used as a symbol.
Reserved words are indicated in Appendix A, List of Reserved Words.

④ The same symbol cannot be described two or more times (provided that the name defined with the SET directive can be re-defined with the SET directive).

⑤ Lowercase letters described as a symbol will be interpreted as their uppercase equivalents.

⑥ When describing a label in the Symbol field, separate the Symbol field from the Mnemonic field with a delimiter ":".

(Example of correct symbol descriptions)

```
    TEN       EQU     10H            ; "TEN" is a name.
    NEXT:     BR      !100H          ; "NEXT" is a label.
    C1        CSEG                   ; "C1" is a segment name.
              NAME    SAMPLE         ; "SAMPLE" is a module name.
    MAC1      MACRO                  ; "MAC1" is a macro name.
```

(Example of incorrect symbol descriptions)

```
   SEVENTY    EQU     70H            ; "SEVENTY" is a name.
   1ST:       MOV     A,#0H          ; No numeric character can be used
                                       as the 1st character of a symbol.
   NEXT       BR      !100H          ; "NEXT" is a label and must be
                                       separated from Mnemonic field
                                       with a colon (:)
   TEN        EQU     10H            ; "TEN" and "ten" are the same named
   ten        EQU     20H              symbols. Description of "ten" will
                                       thus result in an error.
```

[Symbol attributes]

Names and labels each have a value and an attribute.
Segment names, module names, and macro names have no value.
A value refers to the value of a defined numerical data or address data itself.
The attribute of a symbol is called a symbol attribute and must be one of the types indicated in Table 2-3.

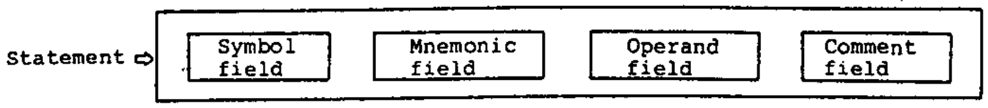
Table 2-3. Types of Symbol Attributes

Attribute type	Classification
NUMBER	o Names defined with EQU and SET directives (provided that labels, bit values, and symbols having a bit value defined with the directives are excluded.)
ADDRESS	o Symbols defined as labels o Names defined as labels with EQU and SET directives
BIT	o Names defined as bit values with EQU and SET directives o Names defined with DBIT directive

(Examples)

```
   TEN     EQU    10H          ; Name "TEN" has attribute NUMBER
                                 and value 10H.

           ORG    80H
   START:  MOV    A,#10H       ; Label "START" has attribute
                                 ADDRESS and value 80H.

   BIT1    EQU    0FE20H. 0    ; Name "BIT1" has attribute BIT and
                                 value 0FE20H. 0.
```

(2) Mnemonic field



In the Mnemonic field, a mnemonic instruction or directive is described.
With an instruction or directive requiring an operand or operands, the Mnemonic field must be separated from the Operand field with one or more Blank or TAB characters.

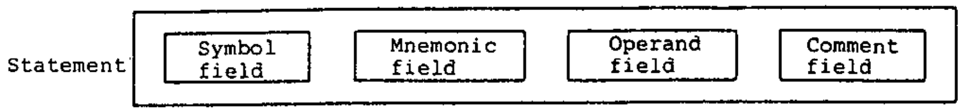
(Example of correct descriptions)

```
MOV    A, #0H
CALL   !CONVAH
RET
```

(Example of incorrect descriptions)

```
MOVA,#0H          ; No blank exists between Mnemonic and
                    Operand fields.
CAL L !CONVAH     ; A blank exists in Mnemonic field.
HLT               ; uCOM78/I has no such instruction as
                    "HLT".
```

(3) Operand field



In the Operand field, the data (operands) required for the instruction or directive described in the Mnemonic field must be described. Depending on the instruction or directive, no operand can be described in the Operand field or two or more operands must be described in the Operand field.
When describing two or more operands, delimit each operand with a comma (,).
The following five types of data can be described in the Operand field:

o Constants (numeric constant and character constant)
o Register names
o Special characters ($, #, !, and [ ])
o Names and labels
o Expressions

The size and attribute of the required operand may be different depending on the instruction or directive. Refer to Section 2.4, "Characteristics of Operands" for the sizes and attributes of operands.
See APPENDIX C, "uCOM-78K/I Instruction Set" for the operand representation formats and description methods in the uCOM-78K/I instruction set.
Each of these five types of data that can be described in the Operand field is detailed below.

[Constants]

A constant is a fixed value or data item and is also referred to as an immediate data.
Constants are divided into numeric constants and character constants.

o Numeric constants

A binary, octal, decimal, or hexadecimal number can be described as a numeric constant. The method of representing each numeric constant type is shown in Table 2-4 below.

Table 2-4. Methods of Representing Numeric Constant Types

Constant type	Method of representation	Example
Binary constant (Binary number)	Character "B" is suffixed to a string of binary characters (value).	1101B
Octal constant (Octal number)	Character "O" is suffixed to a string of octal characters (value).	74O
Decimal constant (Decimal number)	A string of decimal characters (value) is described with or without suffixing character "D" to the string.	128 128D
Hexadecimal constant (Hexadecial number)	Character "H" is suffixed to a string of hexadecimal characters (value). If the first character of the constant begins with one of the characters "A through F", "0" must be prefixed to the constant.	8CH 0A6H

o Character constants

A character constant is expressed by enclosing a character or a string of characters shown in 2.2.2, "Character set" with a pair of single quotation marks.
As a result of an assembly process, the character or characters are converted into 7-bit ASCII code with the parity bit (MSB) set as "0".
To use a single quotation mark as it is originally intended, the single quotation mark must be input twice in succession.

Examples:

```
'A'
' '          ; Represents a single blank.
''''         ; Represents a pair of single quotation
               marks.
'main'
```

[Register names]

The following registers can be described in the Operand field.

- o General-purpose registers
- o General-purpose register pairs
- o Special function registers

General-purpose registers and general-purpose register pairs include those which can be described with their absolute names (R0 to R7 and RP1 to RP3), as well as with their function names (X, A, B, C, D, E, H, L,, AX, BC, DE, and HL). A register name that can be described in the Operand field may be different depending on the type of instruction. See APPENDIX C.1, "Instruction Set and Its Operation" for details of the method of describing each register.

[Special characters]

Special characters that can be described in the Operand field are shown in Table 2-5.

Table 2-5. Special Characters That Can Be Described in Operand Field

Special character	Function
$	o Indicates the location address of the instruction having this operand (or the 1st byte of the address with a multiple-byte instruction). o Indicates a relative addressing mode for a Branch instruction.
!	o Indicates an absolute addressing mode for a Branch or Call instruction. o Indicates the specification of addr16 which allows all memory space to be specified for a MOV instruction.
#	Indicates an immediate data.
[ ]	Indicates an indirect addressing mode.

(Application examples of special characters)

```
Address           Source program
100               LOOP:   INC A
101                       BNZ $$-1    .......①
```

In ① above, the first "$" in the Operand field indicates the relative addressing of the conditional branch instruction BNZ. The second "$" indicates the location address 101 to which the first byte of the object code for the instruction "BNZ $$-1" is to be assigned.
The description in ① can be substituted with "BNZ $LOOP".

```
Source program
     BR   !100H          ; "!" indicates the absolute address-
                           ing of BR (unconditional branch)
                           instruction.
     MOV A, !2000H       ; "!" indicates addr16 specification
                           of MOV instruction
     SUB A, #10H         ; "#" indicates an immediate data.
```

```
   TEN   EQU 10H
         SUB A, #TEN          ; "#" indicates an immediate data.

         AND A, [HL]          ; "[ ]" indicate an indirect
                                addressing mode.
```

[Names and labels]

If a name or label is described in the Operand field, the value of the name or label becomes a numerical data subject to operation by the instruction or directive described in the Mnemonic field.

(Application example of name)

```
TEN     EQU  10H
        MOV  A, #TEN     ; This description can be substituted
                           with "MOV A, #10H".
```

(Application example of label)

```
        ORG  100H
LOOP:   INC  A
        BNZ  $LOOP       ; This description can be substituted
                           with "BNZ $100H".
```

[Expressions]

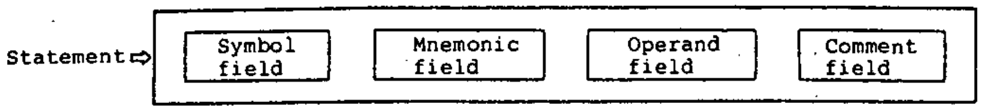
In the Operand field, expressions can be described.
An expression is a valid series of constants, $ indicating a location address, names, or labels, that are connected with operators and can be used as an operand of an instruction.
For the expressions and operators, see Section 2.3, "Expressions and Operators".

(Example of expression description)

```
TEN   EQU  10H
      MOV  A, #TEN-5H
```

In this example, "#TEN-5H" is an expression. In this expression, name "TEN" and numeric constant "5H" are connected with the "-" (minus) operator. The value of the expression is 0BH. Therefore, this description can be substituted with "MOV A, #0BH".

(4) Comment field



In the Comment field, any remarks to identify or explain a particular step or operation in a program may be described following the input of a semicolon (;).
By describing comments in the Comment field, an easy-to-understand source program can be created. The comments described in the Comment field are not subject to assembler operation (i.e., conversion into machine language) but are output as is on an assembly list.

(Examples of comment descriptions)

```
$       PROCESSOR(112)

        NAME    SAMPS
;*************************************************
;*                                               *
;*   HEX -> ASCII Conversion Program             *
;*                                               *
;*              sub-routine                      *
;*                                               *
;*   input condition  : (HL) <- hex 2 code       *
;*                                               *
;*   output condition : BC-register <-ASCII 2 code *
;*                                               *
;*************************************************

        PUBLIC  CONVAH

        CSEG
CONVAH: MOV     A,[HL]          ;load hex code -> Acc
        SHR     A,4             ;hex upper code load
        CALL    !SASC
        MOV     B,A             ;store result

        MOV     A,[HL]          ;load hex code -> Acc
        AND     A,#0FH          ;hex lower code load
        CALL    !SASC
        MOV     C,A             ;store result

        RET
```

Lines consisting of Comment field only

Lines in which comments are described

2.3 Expressions and Operators

An expression is a valid series of constants, $ indicating a location address, names, or labels connected with operators. Elements of an expression other than the operators are called terms and are referred to as the 1st term, 2nd term, and so forth from left to right, in their order of description.

Operators are available in the types shown in Table 2-6, and their order of precedence in calculation has been predetermined as shown in Table 2-7.

A pair of parentheses (i.e., left and right parentheses) are used to change the order in which calculations are to be performed.

```
Example:  MOV A, #5*(SYM+1)        ; ①
```

In ① above, "5*(SYM+1)" is an expression. "5" is the 1st term of the expression and "SYM" and "1" are the 2nd and 3rd terms, respectively. "*, +, and ( )" are operators.

Table 2-6. Types of Operators

Type of operator	Operators
Arithmetic operators	+, -, *, /, MOD, + sign, - sign
Logical operators	OR, AND, NOT, XOR
Relational operators	EQ, NE, GT, GE, LT, LE
Shift operators	SHR, SHL
Byte-separating operators	HIGH, LOW
Bit operator	. (Period)
Other operators	( )

Table 2-7. Order of Precedence of Operators

Priority		Operator
High	1	HIGH, LOW
↑	2	+ sign, - sign, NOT
	3	* (Multiply), / (Divide), MOD, SHR, SHL
	4	+ (Add), - (Subtract)
	5	AND
	6	OR, XOR
↓	7	EQ, NE, GT, GE, LT, LE
Low	8	. (Bit operator)

Operations on expressions are performed according to the following rules:

① Operations are performed according to the order of precedence given to each operator. If two or more operators of the same order of precedence exists in an expression, the operation designated by the leftmost operator is first carried out.

② An expression in parentheses is carried out before expressions outside the parentheses.

③ Each term in an operation is handled as an unsigned 16-bit data and the result of the operation is also handled as an unsigned 16-bit. (However, the result of the operation designated by the HIGH, LOW, EQ, NE, GT, LT, or LE operator will be handled as an 8-bit data.)

```
Example: 65535 = 0FFFFH
            -1 = -(0001H) = 0FFFFH
```

④ If an overflow occurs in an operation due to its result exceeding 16 bits, only the low-order 16 bits of the result become valid and thus an error will not result.

```
Example: 65535 + 1 = (0FFFFH) + (0001H)
                   = (10000H) ——→ 0000H
```

2.3.1 Functions of operators

The functions of the respective operators are described in this subsection.

(1) + (ADD) operator

Function

Returns the sum of the value of the 1st term of an expression and the value of its 2nd term.

Application Example

```
            ORG     100H
START:      BR      !$+6H     ;(a)
              :
```

Explanation

The BR instruction causes a jump to "current address + address 6", namely, to address "100H + 6H = 106H".
Therefore, (a) in the above example can also be described as:
START: BR !106H

(2) - (SUBTRACT) operator

Function

Returns a difference between the value of the 1st term of an expression and the value of its 2nd term.

Application Example

```
            ORG     100H
BACK:       BR      !BACK-3H ;(b)
              :
```

Explanation

The BR instruction causes a jump to "address assigned to BACK - address 3", namely, to address "100H - 3H = 0FDH".
Therefore, (b) in the above example can also be described as:
BACK: BR !0FDH

(3) * (MULTIPLY) operator

Function

Returns the product of the value of the 1st term of an expression and the value of its 2nd term.

Application Example

```
TEN   EQU   10H
      MOV   A,#TEN*3  ; (c)
        :
```

Explanation

With the EQU directive, value "10H" is defined in name "TEN". "#" indicates an immediate data. Expression "TEN*3" is the same as "10*3" and returns value 30H.
Therefore, (c) in the above expression can also be described as: MOV A, #30H

(4) / (DIVIDE) operator

Function

Divides the value of the 1st term of an expression by the value of its 2nd term and returns the integer part of the result with its fractional part truncated. An error will result if the divisor is 0.

Application Example

```
MOV   A,#256/50  ; (d)
```

Explanation

The result of division "256/50" is 5 with remainder 6.
The * operator returns value "5" which is the integer part of the result of the division.
Therefore, (d) in the above example can also be described as:
MOV A, #5

(5) + sign

<u>Function</u>

Returns the value of the term of an expression as is.

<u>Application Example</u>

```
FIVE  EQU  +5
```

<u>Explanation</u>

The value "5" of the term is returned without change.
With the EQU directive, value "5" is defined in name "FIVE".

(6) - sign

<u>Function</u>

Returns the value of the term of an expression by twos complement.

<u>Application Example</u>

```
NO  EQU  -1
```

<u>Explanation</u>

-1 becomes the twos complement of 1.

```
The twos complement of binary 0000 0000 0000 0001
becomes:                      1111 1111 1111 1111
```

Therefore, with the EQU directive, value "0FFFFH" is defined in name "NO".

(7) MOD (Remainder) operator

Function

Obtains the remainder in the result of dividing the value of the 1st term of an expression by the value of its 2nd term. An error will result if the divisor (2nd term) is 0.

Application Example

```
MOV   A, #256 MOD 50   ; (e)
```

Explanation

The result of division "256/50" is 5 with remainder 6.
The MOD operator returns the remainder 6.
Therefore, (e) in the above example can also be described as:
MOV A, #6

(1) NOT operator

<u>Function</u>

Negates the value of the term of an expression on a bit-by-bit basis and returns the result.

<u>Application Example</u>

```
MOVW   AX, #NOT 3H   ; (a)
```

<u>Explanation</u>

NOT logical operation is performed on value 3H as follows:

```
 NOT)  0000 0000 0000 0011
       ───────────────────
       1111 1111 1111 1100
```

The result becomes 0FFFCH.

Therefore, (a) in the above example can also be described as:

MOV AX, #0FFFCH

(2) AND operator

<u>Function</u>

Performs an AND (logical sum) operation between the value of of the 1st term of an expression and the value of its 2nd term on a bit-by-bit basis and returns the result.

<u>Application Example</u>

```
MOV A, #110H AND 0FFH ; (b)
```

<u>Explanation</u>

AND operation is performed between two values 110H and 0FFH as follows:

```
       0000 0001 0001 0000
 AND)  0000 0000 1111 1111
       ───────────────────
       0000 0000 0001 0000
```

The result becomes 10H.

Therefore, (b) in the above example can also be described as:

MOV A, #10H

(3) OR operator

Function

Performs an OR (logical product) operation between the value of the 1st term of an expression and the value of its 2nd term on a bit-by-bit basis and returns the result.

Application Example

```
MOV   A, #0AH  OR  1101B    ; (c)
```

Explanation

OR operation is performed between two values 0AH and 1101B as follows:

```
         0000 0000 0000 1010
   OR)   0000 0000 0000 1101
  ---------------------------
         0000 0000 0000 1111
```

The result becomes 0FH.

Therefore, (c) in the above example can also be described as:
MOV A, #0FH

(4) XOR operator

Function

Performs an Exclusive-OR operation between the value of the 1st term of an expression and the value of its 2nd term on a bit-by-bit basis and returns the result.

Application Example

```
MOV   A, #9AH  XOR  9DH    ; (d)
```

Explanation

XOR operation is performed between two values 9AH and 9DH as follows:

```
         0000 0000 1001 1010
  XOR)   0000 0000 1001 1101
  ---------------------------
         0000 0000 0000 0111
```

The result becomes 7H.

Therefore, (d) in the above example can also be described as:
MOV A, #7H

(1) EQ (Equal) operator

Function

Returns 0FFH if the value of the 1st term of an expression is equal to the value of its 2nd term (i.e., true) and 00H if both values are not equal (i.e., false).

Application Example

```
A1   EQU   12C4H
A2   EQU   12C0H

     MOV   A, #A1  EQ  (A2+4)  ;(a)
     MOV   X, #A1  EQ   A2     ;(b)
```

Explanation

In (a) above,
expression "A1 EQ (A2+-4)" becomes "12C4H EQ (12C0H+4)".
The operator compares the value of the 1st term and that of the 2nd term and returns 0FFH because the value of the 1st term is equal to the value of the 2nd term.
In (b) above,
expression "A1 EQ A2" becomes "12C4H EQ 12C0H".
The operator compares the value of the 1st term and that of the 2nd term and returns 00H because the value of the 1st term is not equal to the value of the 2nd term.

(2) NE (Not Equal) operator

Function

Returns 0FFH if the value of the 1st term of an expression is not equal to the value of its 2nd term (i.e., true) and 00H if both values are equal (i.e., false).

Application Example

```
A1   EQU   5678H
A2   EQU   5670H

     MOV   A, #A1  NE   A2        ; (c)
     MOV   A, #A1  NE  (A2+8H)    ; (d)
```

Explanation

In (c) above,
expression "A1 NE A2" becomes "5678H NE 5670H".
The operator compares the value of the 1st term and that of the 2nd term and returns 0FFH because the value of the 1st term is not equal to the value of the 2nd term.
In (d) above,
expression "A1 NE (A2+8H)" becomes "5678H NE (5670H+8H)".
The operator compares the value of the 1st term and that of the 2nd term and returns 00H because the value of the 1st term is equal to the value of the 2nd term.

(3) GT (Greater-Than) operator

Function

Returns 0FFH if the value of the 1st term of an expression is greater than the value of its 2nd term (i.e., true) and 00H if the value of the 1st term is equal to or less than the value of the 2nd term (i.e., false).

Application Example

```
A1   EQU   1023H
A2   EQU   1013H

     MOV   A, #A1  GT   A2           ;(e)
     MOV   X, #A1  GT  (A2+10H)      ;(f)
```

Explanation

In (e) above,

expression "A1 GT A2" becomes "1023H GT 1013H".

The operator compares the value of the 1st term and that of the 2nd term and returns 0FFH because the value of the 1st term is greater than the value of the 2nd term.

In (f) above,

expression "A1 GT (A2+10H)" becomes "1023H GT (1013H+10H)".

The operator compares the value of the 1st term and that of the 2nd term and returns 00H because the value of the 1st term is equal to the value of the 2nd term.

(4) GE (Greater-than or Equal) operator

Function

Returns 0FFH if the value of the 1st term of an expression is greater than or equal to the value of its 2nd term (i.e., true) and 00H if the value of the 1st term is less than the value of the 2nd term (i.e., false).

Application Example

```
A1   EQU    2037H
A2   EQU    2015H

     MOV    A, #A1  GE   A2         ;(g)
     MOV    X, #A1  GE  (A2+23H)    ;(h)
```

Explanation

In (g) above,
expression "A1 GE A2" becomes "2037H GE 2015H".
The operator compares the value of the 1st term and that of the 2nd term and returns 0FFH because the value of the 1st term is greater than the value of the 2nd term.
In (h) above,
expression "A1 GE (A2+23H)" becomes "2037H GE (2015H+23H)".
The operator compares the value of the 1st term and that of the 2nd term and returns 00H because the value of the 1st term is less than the value of the 2nd term.

(5) LT (Less-Than) operator

Function

Returns 0FFH if the value of the 1st term of an expression is less than the value of its 2nd term (i.e., true) and 00H if the value of the 1st term is equal to or greater than the value of the 2nd term (i.e., false).

Application Example

```
A1      EQU     1000H
A2      EQU     1020H

        MOV     A, #A1 LT A2          ;(i)
        MOV     X, #(A1+20H) LT A2 ;(j)
```

Explanation

In (i) above,
expression "A1 LT A2" becomes "1000H LT 1020H".
The operator compares the value of the 1st term and that of the 2nd term and returns 0FFH because the value of the 1st term is less than the value of the 2nd term.
In (j) above,
expression "(A1 + 20H) LT A2" becomes "(1000H + 20H) LT 1020H". The operator compares the value of the 1st term and that of the 2nd term and returns 00H because the value of the 1st term is equal to the value of the 2nd term.

(6) LE (Less-Than or Equal) operator

Function

Returns 0FFH if the value of the 1st term of an expression is less than or equal to the value of its 2nd term (i.e., true) and 00H if the value of the 1st term is greater than the value of the 2nd term (i.e., false).

Application Example

```
A1   EQU    103AH
A2   EQU    1040H

     MOV    A, #A1 LE A2           ;(k)
     MOV    X, #(A1+7H) LE A2      ;(l)
```

Explanation

In (k) above,
expression "A1 LE A2" becomes "103AH LE 1040H".
The operator compares the value of the 1st term and that of the 2nd term and returns 0FFH because the value of the 1st term is less than the value of the 2nd term.
In (l) above,
expression "(A1 + 7H) LE A2" becomes "(103AH + 7H) LE 1040H".
The operator compares the value of the 1st term and that of the 2nd term and returns 00H because the value of the 1st term is greater than the value of the 2nd term.

(1) SHR (Shift Right) operator

Function

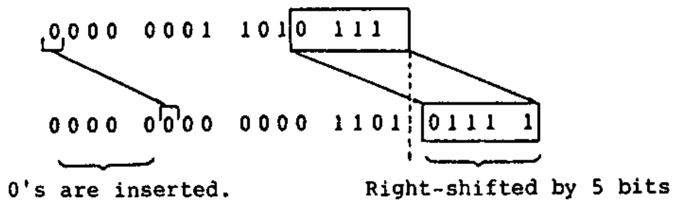
Returns a value obtained by shifting the value of the 1st term of an expression to the right the number of bits specified by the value of the 2nd term. In this case, zeroes equivalent to the specified number of bits shifted move in from the MSB side.

Application Example

```
MOV A, #1AFH SHR 5   ; (a)
```

Explanation

This operator shifts value "1AFH" to the right by 5 bits.



As the result of the Shift Right operation, value 0DH is returned.

Therefore, (a) in the above example can also be described as:

MOV A, #0DH

(2) SHL (Shift Left) operator

Function

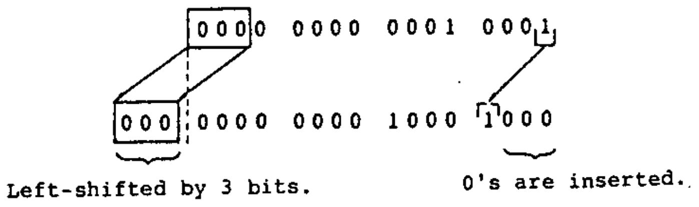
Returns a value obtained by shifting the value of the 1st term of an expression to the left the number of bits specified by the value of the 2nd term. In this case, zeroes equivalent to the specified number of bits shifted move in from the LSB side.

Application Example

```
MOV A, #11H SHL 3    ; (b)
```

Explanation

This operator shifts value "11H" to the left by 3 bits.



As the result of the Shift Left operation, value 88H is returned.

Therefore, (b) in the above example can also be described as:

MOV A, #88H

(1) HIGH operator

Function

Returns the high-order 8-bit value of the term of an expression.

Application Example

```
        ORG    1234H
START:
         :
        MOV    A, #HIGH START   ; (a)
```

Explanation

Because label "START" has value 1234H, the HIGH operator returns the high-order 8 bits of the value, namely, 12H. Therefore, (a) in the above example can also be described as: MOV A, #12H

(2) LOW operator

Function

Returns the low-order 8-bit value of the term of an expression.

Application Example

```
        ORG    5678H
WORK:
         :
        MOV    A, #LOW WORK   ; (b)
```

Explanation

Because label "WORK" has value 5678H, the LOW operator returns the low-order 8 bits of the value, namely, 78H. Therefore, (b) in the above example can also be described as: MOV A, #78H

Other Operators

(1) . (Bit position specification)

Function

Calculates a bit address from the value indicated by the 1st term of an expression and the bit position indicated by the 2nd term of the expression and returns the result.

Application Example

```
MOV CY,0FE20H.3
```

Explanation

This operator sets the value of the 3rd bit at address 0FE20H in the Carry (CY) flag.

Assuming that data 2BH is stored at address 0FE20H,

0FE20H	0	0	1	0	1	0	1	1

0 (i.e., the value of the 3rd bit) is set in the CY flag.

NOTE 2-2

- o The value of the 1st term of an expression must be within the range of 0FE20H to 0FFFFH.
- o The value of the 2nd term of the expression must be within the range of 0 to 7.

(2) ( )

Function

Causes an operation in parentheses to be performed prior to operations outside the parentheses.
This operator is used to change the order of precedence of other operators.

Application Example

```
MOV A,#(4+3)*2
```

Explanation

$$\underbrace{\underbrace{(4+3)}_{\text{①}}*2}_{\text{②}}$$

Calculations are performed in the order of expressions ① and ② and value 14 is returned as the result.
If parentheses are not used as shown below,

$$\underbrace{4+\underbrace{3*2}_{\text{①}}}_{\text{②}}$$

calculations are performed in the order of expressions ① and ② and a different value 10 is returned as the result.
See Table 2-7 for the order of precedence of operators.

2.3.2 Restrictions on Operations

The operation of an expression is performed by connecting terms with operator(s). Elements that can be described as terms include constants, $, names, and labels. Each term has a relocation attribute and a symbol attribute.

Depending on the types of relocatable attribute and symbol attribute inherent in each term, operators that can work on the term are limited. Therefore, when describing an expression, it is important to pay attention to the relocation attribute and symbol attribute of each of the terms constituting the expression.

(1) Operators and relocation attributes

As previously mentioned, each of the terms which constitute an expression has a relocation attribute. Terms can be divided into three types when classified by their relocation attributes: Absolute term, Relocatable term, and External reference term.

Types of relocation attributes in operations, nature of each attribute, and terms applicable to each attribute are shown in Table 2-8.

Table 2-8. Types of Relocation Attributes

Type	Nature	Applicable term
Absolute term	Term whose value is determined at assembly time	o Constants o Labels defined within an absolute segment o $ indicating the location address defined within an absolute segment o Names defined the above constants, labels, or $
Relocatable term	Term whose value is not determined at assembly time	o Labels defined within a relocatable segment o $ indicating the location address defined within a relocatable segment o Names defining relocatable labels

Table 2-8. Types of Relocation Attributes (contd)

Type	Nature	Applicable term
External reference term	Term which externally references the symbol of another module	o Labels defined with EXTRN directive o Names defined with EXTBIT directive

Combinations of the type of operator and terms on which each operator can work are shown in Table 2-9.

Table 2-9. Combinations of Operators and Terms by Relocation Attribute

Relocation attribute of term / Type of operator	X: ABS Y: ABS	X: ABS Y: REL	X: REL Y: ABS	X: REL Y: REL
X + Y	A	R	R	-
X - Y	A	-	R	A*
X * Y	A	-	-	-
X / Y	A	-	-	-
X MOD Y	A	-	-	-
X SHL Y	A	-	-	-
X SHR Y	A	-	-	-
X EQ Y	A	-	-	A*
X LT Y	A	-	-	A*
X LE Y	A	-	-	A*
X GT Y	A	-	-	A*
X GE Y	A	-	-	A*
X NE Y	A	-	-	A*
X AND Y	A	-	-	-
X OR Y	A	-	-	-
X XOR Y	A	-	-	-
NOT X	A	A	-	-
+ X	A	A	R	R
- X	A	A	-	-
HIGH X	A	A	R	R
LOW X	A	A	R	R
X . Y	A	-	R	-

<Legend> ABS: Absolute term

REL: Relocatable term

A : The result of the operation becomes an absolute term.

R : The result of the operation becomes an relocatable term.

- : The operation cannot be performed.

* The operation is allowed only between the symbols defined within the same segment.

The following four operators can work on external reference terms: +, HIGH, LOW, and . (Bit position specification). (However, note that only one external reference term can be described in an expression.)
Combinations of the type of operator and external reference terms on which each operator can work are shown in Table 2-10.

Table 2-10. Combinations of Operators and Terms by Relocation Attribute (External Reference Term)

Relocation attribute of term / Type of operator	X: ABS Y: EXT	X: EXT Y: ABS	X: REL Y: EXT	X: EXT Y: REL	X: EXT Y: EXT
X + Y	-	E	-	-	-
+ X	A	E	R	E	E
HIGH X	A	E	R	E	E
LOW X	A	E	R	E	E
X . Y	E	E	-	E	-

<Legend> ABS: Absolute term

REL: Relocatable term

EXT: External reference term

A : The result of the operation becomes an absolute term.

R : The result of the operation becomes an relocatable term.

E : The result of the operation becomes an external reference term.

- : The operation cannot be performed.

(2) Operators and symbol attributes

As previously mentioned, each of the terms which constitute an expression has a symbol attribute in addition to a relocation attribute. Terms can be divided into three types when classified by their system attributes: NUMBER term, ADDRESS term, and BIT term.

Types of system attributes in operations and terms applicable to each attribute are shown in Table 2-11.

Table 2-11. Types of Symbol Attributes

Type	Applicable term
NUMBER term	o Names and labels which have NUMBER attribute o Constants
ADDRESS term	o Names and labels which have ADDRESS attribute o $ indicating the location counter
BIT term	o Names which have BIT attribute

Combinations of the type of operator and terms on which each operator can work are shown in Table 2-12.

Operations on BIT terms are possible only with unary operator "+ X".

Table 2-12. Combinations of Operators and Terms by Symbol Attribute

Symbol attribute of term / Type of operator	X:ADDRESS Y:ADDRESS	X:ADDRESS Y:NUMBER	X:NUMBER Y:ADDRESS	X:NUMBER Y:ADDRESS
X + Y	-	A	A	N
X - Y	N	A	-	N
X * Y	-	-	-	N
X / Y	-	-	-	N
X MOD Y	-	-	-	N
X SHL Y	-	-	-	N
X SHR Y	-	-	-	N
X EQ Y	N	-	-	N
X LT Y	N	-	-	N
X LE Y	N	-	-	N
X GT Y	N	-	-	N
X GE Y	N	-	-	N
X NE Y	N	-	-	N
X AND Y	-	-	-	N
X OR Y	-	-	-	N
X XOR Y	-	-	-	N
NOT X	-	-	N	N
+ X	A	A	N	N
- X	-	-	N	N
HIGH X	A	A	N	N
LOW X	A	A	N	N
X . Y	-	B	-	B

<Legend> ADDRESS	:	ADDRESS term
NUMBER	:	NUMBER term
A	:	The result of the operation becomes an ADDRESS term.
N	:	The result of the operation becomes a NUMBER term.
B	:	The result of the operation becomes a BIT term.
-	:	The operation cannot be performed.

(3) How to check restrictions on the operation

An example of an operation by the relocation attribute and symbol attribute of each term is shown here.

Example: BR <u>$TABLE+5H</u>

Here, assume that "TABLE" is a label defined in a relocatable code segment.

① Operator and relocation attribute

Because "TABLE+5H" is "relocatable term + absolute term", apply this operation to Table 2-9, "Combinations of Operators by Relocatable Attribute".

Type of operator $\longrightarrow$ X + Y

Relocation attribute of term $\longrightarrow$ X:ABS, Y:REL

From the table, you will find that the result becomes a relocatable term.

② Operator and symbol attribute

Because "TABLE+5H" is "ADDRESS term + NUMBER term", apply this operation to Table 2-12, "Combinations of Operators and Terms by Symbol Attribute".

Type of operator $\longrightarrow$ X + Y

Symbol attribute of term $\longrightarrow$ X:ADDRESS, Y:NUMBER

From the table, you will find that the result becomes an ADDRESS term.

2.4 Characteristics of Operands

Instructions and directives requiring an operand or operands differ from one type of instruction to another in the size and address range of the required operand value and in the symbol attribute of the operand.

For example, an instruction "MOV r1, #byte" functions to transfer the value indicated by "byte" to register "r1". In this case, because r1 is an 8-bit register, the size of the data "byte" to be transferred must be 8 bits or less.

If an instruction is described as "MOV R0, #100H", an assembly error occurs because the size of the 2nd operand "100H" of the instruction exceeds the capacity of the 8-bit register R0.

So, when you describe an operand, attention must be paid to the following points:

- o Is the size of the operand value or its address range suitable for the operand (numerical data, name, or label) of the instruction?
- o Is the symbol attribute suitable for the operand (name or label) of the instruction?

2.4.1 Size and address range of operand value

Certain conditions are set for the size and address range of the value of a numerical data, name, or label that can be described as the operand of an instruction or directive.

With instructions, conditions for the size and address range of an operand value are governed by the operand representation format of each instruction. With directives, such conditions are governed by the type of directive.

These conditions are shown in Tables 2-13 and 2-14 below.

Table 2-13. Conditions for Size of Operand Value

Instructions	Representation format of operand	Size of operand value
	word	16 bits max.
	byte	8 bits max.
	bit	3 bits max.
	n	3 bits max.
Directives	Type of directive	Size of operand value
	EQU	16 bits max.
	SET	16 bits max.
	DB	8 bits max.
	DW	16 bits max.
	DS	16 bits max.

NOTE 2-3

When describing names or labels as operands, note that names or labels with the following attributes will be handled as 16-bit data.

- o Name or label with symbol attribute ADDRESS
- o Name or label defined with the EXTRN directive

If any of these names or labels is described as an 8-bit operand, an error will result. To avoid this error, describe the name or label by correcting it to an 8-bit value with the HIGH or LOW operator.

Table 2-14. Conditions for Address Range of Operand Value

Instructions	Representation format of operand	Address range of operand value
	saddr	0FE20H to 0FF1FH
	saddrp	Even value of 0FE20H to 0FF1FH
	addr13	0000H to 1FFFH
	addr16	0000H to 3EFFH
	addr11	0800H to 0FFFH
	addr5	Even value of 0040H to 007EH
Directives	Type of directive	Address range of operand value
	ORG	0000H to 0FFFFH
	BR	0000H to 0FEFFH

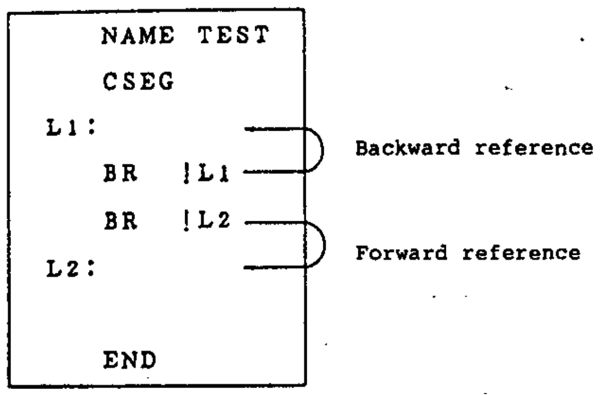
2.4.2 Symbol attributes and relocation attributes of operands

When describing a name, label, or $ (location counter) as the operand of an instruction, the name or label may or may not be described as an operand depending on the symbol attribute and relocation attribute of the operand as the term in an expression (see 2.3.2, "Restrictions on operations") or the reference direction of the name or label.

Names and labels are referenced in two directions: Backward and Forward.

o Backward reference	Name or label to be referenced as an operand has been defined in a previous line.
o Forward reference	Name or label to be referenced as an operand has been defined in a subsequent line.

Example:



```
        NAME TEST
        CSEG
L1:
        BR    !L1
        BR    !L2
L2:

        END
```

These attributes of the operands of instructions and directives are shown in Tables 2-15 and 2-16 below.

Table 2-15. Operand Attributes of Instructions

Symbol attribute	NUMBER		ADDRESS				NUMBER or ADDRESS		Special function register (sfr name)
Relocation attribute	Absolute term		Absolute term		Relocatable term		External ref. term		
Reference / Operand format	BW	FW	BW	FW	BW	FW	BW	FW	
sfr (FF00H - FFFFH)	-	-	-	-	-	-	-	-	o
sfrp (FF00H - FFFFH)	-	-	-	-	-	-	-	-	o Cf.2
saddr (FE20H - FF1FH)Cf.1	o	-	o	-	-	-	o	-	o (FF00H- FF1FH)
saddrp (Even value of FE20H - FF1FH) Cf.1	o	-	o	-	-	-	o	-	o (FF00H- FF1FH)
addr13 addr16	o	o	o	o	o	o	o	-	-
addr11 (800H - FFFFH)	o	o	o	o	o Cf.3	o Cf.3	o	-	-
addr5 (Even value of 40H - 7EH)	o	o	o	o	o Cf.4	o Cf.4	o	-	-

Table 2-15. Operand Attributes of Instructions (contd)

Symbol attribute	NUMBER		ADDRESS				NUMBER or ADDRESS		Special function register (sfr name)
Relocation attribute	Absolute term		Absolute term		Relocatable term		External ref. term		
Reference / Operand format	BW	FW	BW	FW	BW	FW	BW	FW	
word (16 bits)	o	o	o	o	o	o	o	-	-
byte (8 bits)	o	o	△ Cf.5	△ Cf.5	△ Cf.5	△ Cf.5	△ Cf.5	-	-
bit (3 bits)	o	o	o	o	o	- Cf.6	o	-	-
n (3 bits)	o	o	o	o	o	- Cf.6	o	-	-

BW ... Backward reference FW ... Forward reference

o ... Can be described. - ... Cannot be described.

△ ... May be described by changing it to an 8-bit data.

NOTE: 1. uPD78112 .... FE40H to FF1FH
uPD78122, uPD78124 .... FE20H to FF1FH
2. Only the register names capable of 16-bit manipulation can be described.
3. Only labels defined in the code segment with relocation attribute FIXED can be described.
4. Only labels defined in the code segment with relocation attribute CALLTO can be described.
5. A name or label with symbol attribute ADDRESS, $ (which indicates the current location counter), and an external reference term are handled as 16-bit data regardless of their value range. If any of these symbols is described as operand "byte", an error will result. Change the operand value to an 8-bit data with the HIGH or LOW operator.
6. A term with symbol attribute ADDRESS cannot be described for forward reference as an operand "bit" or "n". Even if you describe such operand by mistake, an error will not occur. However, note that correct object code generation may not be expected from the operand.

Table 2-16. Operand Attributes of Directives

Symbol attribute	NUMBER				ADDRESS						BIT					
Relocation attribute	Abs. term		Ext. term		Abs. term		Rel. term		Ext. term		Abs. ref.		Rel. term		Ext. term	
Reference / Directive	BW	FW	BW	FW	BW	FW	BW	FW	BW	FW	BW	FW	BW	FW	BW	FW
ORG	o	-	-	-	o	-	-	-	-	-	-	-	-	-	-	-
EQU	o	-	-	-	o	-	o	-	-	-	o	-	o	-	o	-
SET	o	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
DB (see Note)	o	o	△	-	△	△	△	△	△	-	-	-	-	-	-	-
DW	o	o	o	-	o	o	o	o	o	-	-	-	-	-	-	-
DS	o	-	-	-	o	-	-	-	-	-	-	-	-	-	-	-
BR	o	o	o	-	o	o	o	o	o	-	-	-	-	-	-	-

Abs. term ... Absolute term   Rel. term ... Relocatable term
Ext. term ... External reference term
BW ... Backward reference   FW ... Forward reference
o ... Can be described.   - ... Cannot be described.
△ ... May be described by changing it to an 8-bit data.

NOTE: A name or label with symbol attribute ADDRESS, $ (which indicates the current location counter), and an external reference term are handled as 16-bit data regardless of their value range. If any of these symbols is described as the operand of the DW directive, an error will result. Change the operand value to an 8-bit data with the HIGH or LOW operator.

CHAPTER 3. DIRECTIVES

3.1 Overview of Directives

Directives are described in a source program just the same as ordinary instructions. Directives are pseudoinstructions in a program which give the assembler processor various instructions necessary for this package to perform a series of processes. Instructions will be translated into object codes (i.e., machine language), but directives will not, as a rule, be converted into object codes.

Directives are available in seven types as listed in Table 3-1 and have the following major functions:

- o Facilitate description of source programs.
- o Initialize memory and reserve areas.
- o Provide the information required for the assembler, linker, and locater to perform their intended processing.

Table 3-1. List of Directives

No.	Type of directive	Directives
1	Segment definition directives	CSEG, DSEG, BSEG, ORG, ENDS
2	Symbol definition directives	EQU, SET
3	Memory initialization/ area reservation directives	DB, DW, DS, DBIT
4	Linkage directives	PUBLIC, EXTRN, EXTBIT, NAME
5	Automatic branch instruction selection directive	BR
6	Macro directives	MACRO, LOCAL, REPT, IRP, EXITM, ENDM
7	Assembly termination directive	END

A detailed description of each of these directives will be provided in the following sections.

In the description format of each directive, "[   ]" indicates that the parameter in braces may be omitted from specification and "..." indicates the repetition of description in the same format.

For example, if the description format reads:

```
[(size)][initial value[, ...]]
```

you may describe any of the following three:

- o (size)
- o (size) initial value 1, initial value 2, initial value 3
- o initial value 1, initial value 2

3.2 Directives for Segment Definition

A source module is described in units of segments.
Segment definition directives are used to define these segments.
Segments are divided into the following five types:

- o Code segment
- o Data segment
- o Bit segment
- o Absolute segment
- o Stack segment

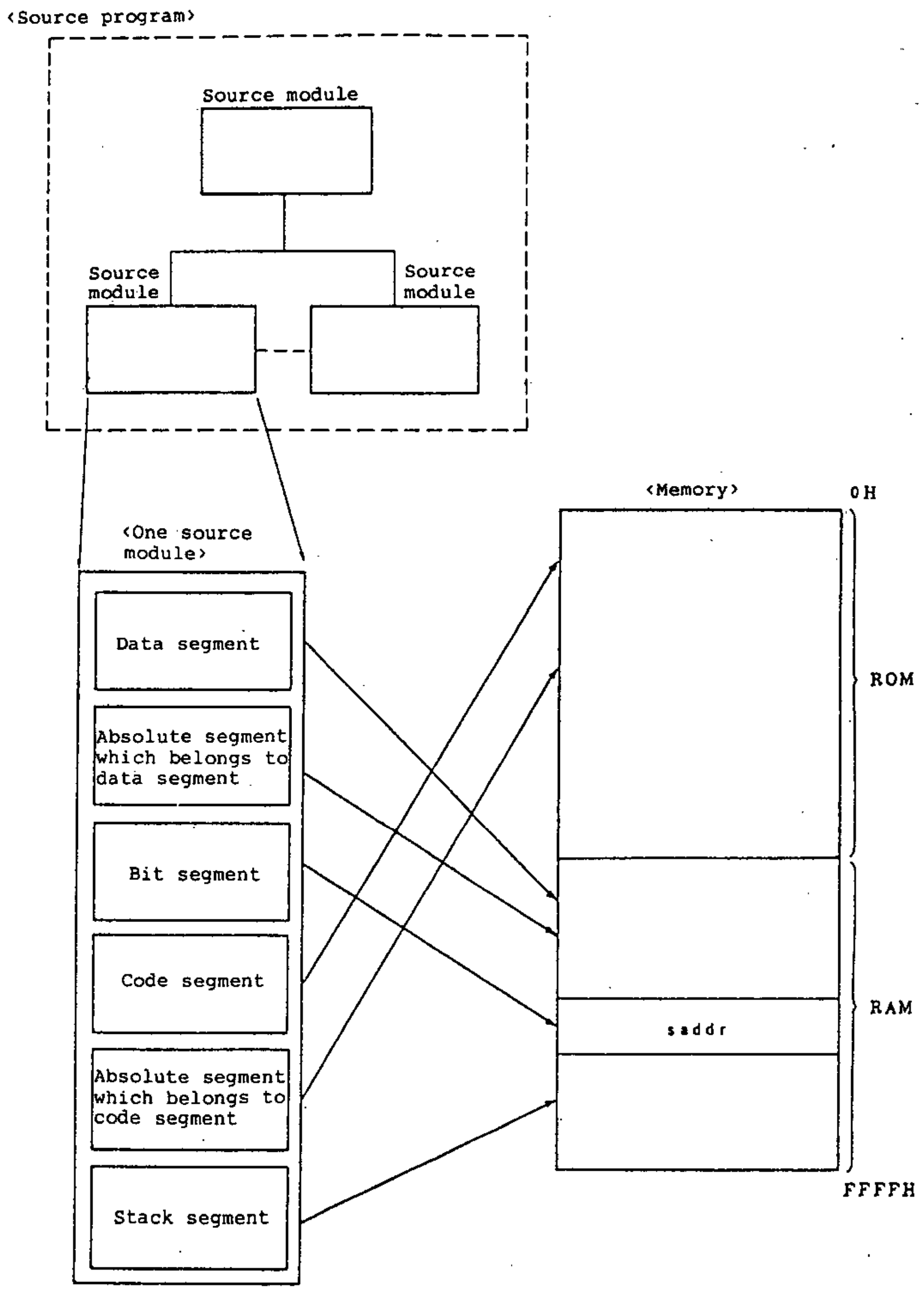
To which address range in memory each segment will be allocated is determined by the type of segment.
Table 3-2 shows the method of defining each segment and the memory address area in which each segment will be allocated.

Table 3-2. Segment Definition Methods and Memory Address Allocation

Type of segment	Method of definition	Memory address area to be allocated to each segment
Code segment	CSEG directive	Within the internal ROM area
Data segment	DSEG directive	Within the internal RAM area
Bit segment	BSEG directive	Within the saddr area in the internal RAM
Absolute segment	ORG directive	Addresses specified by ORG directive
Stack segment	Automatically generated by the assembler	Within the internal RAM area

If the user wishes to determine the memory allocation to a segment, describe (define) the segment as an absolute segment.
An example of memory allocation to segments is shown in Fig. 3-1.

Fig. 3-1. Memory Allocation to Segments



(1) CSEG (code segment)

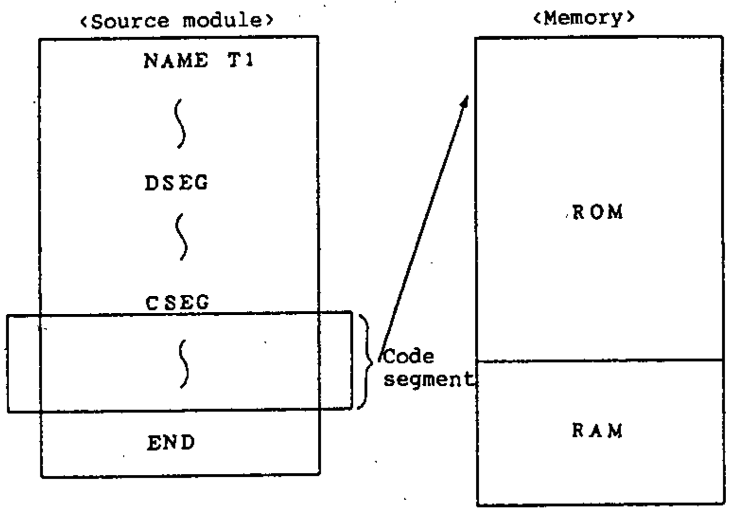
Description Format

Symbol field	Mnemonic field	Operand field	Comment field
[segment name]	CSEG	[relocation attribute]	[;comment]

Function

- o The CSEG directive indicates to the assembler the start of a code segment.
- o All instructions described after this directive until the re-appearance of any segment definition directive (CSEG, DSEG, BSEG, ORG, or ENDS) in the source module will belong to the code segment and will be allocated within the ROM address area upon conversion into machine language.

Fig. 3-2. Relocation of Code Segment



- o If the relocation attribute of a code segment is specified in the Operand field of the CSEG directive, the address range to be allocated to the code segment can be made further definitive.
  There are two types of relocation attributes as shown in Table 3-3.

Table 3-3. Functions of Relocation Attributes

Relocation attribute	Function
CALLT0	Instructs the assembler to allocate the code segment within addresses 40H to 7FH.
FIXED	Instructs the assembler to allocate the code segment within addresses 800H to 0FFFH.

Use

o Describe instructions, DB directive, or DW directive in the code segment defined by the CSEG directive. (However, to allocate the segment from a fixed address, the ORG directive must be described.)

o Specify relocation attributes in the following cases:

Table 3-4. Uses of Relocation Attributes

Relocation attribute	Use
CALLT0	Code segment which defines the entry address of a subroutine to be called with a one-byte instruction "CALLT"
FIXED	Code segment which defines a subroutine to be called with a two-byte instruction "CALLF"

o Description of one functional unit such as a subroutine should be defined as a single code segment. If the size of the unit is relatively large or if the subroutine is highly versatile (can be utilized for development of other programs), the subroutine should be defined as a single module.

Explanation

- o By describing a segment name in the Symbol field of the CSEG directive, the code segment can be named.
  If no segment name is specified for a code segment, the assembler will automatically give a segment name to the code segment. This default segment name is available in three types and is assumed appropriately according to the type of code segment.
  The type of code segment is determined by the type of relocation attribute. The relationship of default segment names and code segment types is shown in Table 3-5.

Table 3-5. Default Segment Names of Code Segments

Default segment name	Code segment type	Classification of code segment type
CSEG	CSEG	Code segment without specified relocation attribute
CSEGT0	CST0	Code segment with relocation attribute "CALLT0"
CSEGFX	CSFX	Code segment with relocation attribute "FIXED"

- o If two or more code segments of the same segment type exist in a source module, these code segments must have the same segment name. (See Application Example 4.)
  This is because of that the code segments of the same segment type described in a source module will be processed as a single code segment within the assembler. (See Application Examples 5 and 6.)
- o Code segments with the same segment name and the same segment type can be described in two or more different modules. These segments are called the same named segments. The same named code segments will be combined into a single code segment at linkage time.

Application Examples

Example 1

```
        NAME   SAMP1
        CSEG   CALLT0      ; (1)
TLAB1:  DW     LAB1

        CSEG               ; (2)
          :
        CALLT  [TLAB1]     ; (3)

        CSEG               ; (4)
LAB1:     :

        END
```

(1) Within this code segment, the entry address of a subroutine to be called by the CALLT instruction is defined. Therefore, relocation attribute "CALLT0" must be specified for this code segment.

(2) Within this code segment, instructions which may be allocated to any locations in the ROM area are described. Therefore, no relocation attribute will be specified for this code segment.

(3) Label "TLAB1" is described to indicate the address in which the entry address of the subroutine is stored.

(4) In this code segment, the subroutine to be called by the CALLT instruction in (3) is defined.

Example 2

```
        NAME   SAMP3
        CSEG   FIXED    ;(1)
SUB1:   :

        CSEG
         :
        CALLF  !SUB1    ;(2)
         :
        END
```

(1) Within this code segment, the entry address of a subroutine to be called by the CALLF instruction is defined. Therefore, relocation attribute "FIXED" must be specified for this code segment.

(2) Label "SUB1" is described as the operand of the CALLF instruction to indicate the address in which the entry address of the subroutine is stored.

Example 3

```
        NAME   SAMP4
SN41    CSEG            ;(1)
         :
SN42    CSEG            ;(2)   ←—— This description is
         :                          erroneous.
        END
```

(1) A code segment is defined without relocation attribute specification (segment type "CSEG"). Segment name is "SN41".

(2) A code segment without specified relocation attribute (segment type "CSEG") has already been defined in (1) above. Therefore, this description results in an error. Change the segment name to "SN41".

Example 4

```
        NAME    SAMP5
SN5     CSEG                ;(1)
          :
        ORG     1000H       ;(2)
          :
SN5     CSEG                ;(3)
          :
        END
```

(1) A code segment is defined without relocation attribute specification (segment type "CSEG"). Segment name is "SN5".

(2) An absolute segment is defined with start address "1000H".

(3) This segment will be processed as a continuous segment which follows the code segment defined in (1) above, with segment type "CSEG" and segment name "SN5".
The first address of the segment defined in (3) will be the address next to the last address of the segment defined in (1) above.

Example 5

<Module 1>

```
        NAME    SAMP61
SN6     CSEG                ;(1)
          :
        END
```

<Module 2>

```
        NAME    SAMP62
SN6     CSEG                ;(2)
          :
        END
```

(1) & (2)

Code segment "SN6" defined in (1) in module 1 becomes the same in segment name and segment type as the code segment defined in (2) in module 2. Therefore, these two code segments will be processed as a single code segment at linkage time.

(2) DSEG (data segment)

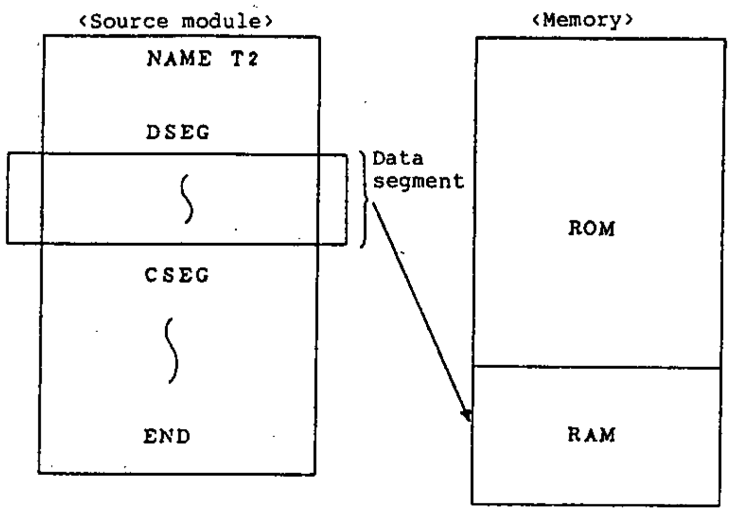
Description Format

Symbol field	Mnemonic field	Operand field	Comment field
[segment name]	DSEG	None	[;comment]

Function

- o The DSEG directive indicates to the assembler the start of a data segment.
- o Memory areas defined by the DS directive after this directive until the re-appearance of any segment definition directive (CSEG, DSEG, BSEG, ORG, or ENDS) in the source module will belong to the data segment and will be finally allocated within the RAM address area.

Fig. 3-3. Relocation of Data Segment



Use

o Mainly describe the DS directive in the data segment defined by the DSEG directive. The data segment will be allocated within the RAM area. Therefore, no instructions can be described in any data segment.

o In a data segment, the DS directive must be described to reserve a RAM work area to be used by the program and a label must be given to the address of each work area. When describing a source program, this label is used. Each area reserved as a data segment will be allocated by the locater so that it does not overlap with any other work areas on the RAM (stack area, general-purpose register area, and work areas defined by other modules).

o A label defined within a relocatable data segment cannot be described as a label to indicate an address in the short direct addressing area.
Accessing data in the short direct addressing area can be processed by instructions with a short word length and a fewer number of clocks. Use this area by specifying absolute addresses for effective utilization.
Short direct addressing area:

```
     uPD78112 ..............0FE40H to 0FF1FH
     uPD78122, uPD78124 ... 0FE20H to 0FF1FH
```

o A label defined within a relocatable data segment can be described only as an operand which is indicated by the operand representation format "addr13", "addr16", or "word".

Explanation

o By describing a segment name in the Symbol field of the DSEG directive, the data segment can be named.
If no segment name is specified for a data segment, the assembler will automatically give a default segment name "DSEG" to the data segment.

o All data segments will have "DSEG" as their segment type.

- o If two or more data segments exist in a source module, these data segments must have the same segment name. (See Application Example 2.)
  This is because of that two or more data segments described in a source module will be processed as a single data segment within the assembler.
- o Data segments with the same segment name can be described in two or more different modules. These segments are called the same named segments.
  The same named data segments will be combined into a single data segment at linkage time.
- o Data segments are segments which define the RAM addresses to be used in the source module. So, define these segments in the early part of the module body.

Application Examples

Example 1

```
        NAME SAMP1
        DSEG                 ;(1)
WORK1:DS      1
WORK2:DS      2
        CSEG
          :
        MOV   A, !WORK1      ;(2)
        MOV   A,WORK1        ;(3)   ——— This description is
                                        erroneous.
          :
        MOVW DE,#WORK2       ;(4)
        MOVW AX,[DE]
          :
        MOVW AX,WORK2        ;(5)   ——— This description is
                                        erroneous.
          :
        END
```

(1) The start of a data segment is defined with the DSEG directive.

(2) This description corresponds to "MOV A, !addr16(addr13)".

(3) This description corresponds to "MOV A, saddr". Relocatable label "WORK1" cannot be described as "saddr". Therefore, an error will occur as a result of this description.

(4) This description corresponds to "MOVW rp1,#word".

(5) This description corresponds to "MOVW AX,saddrp". Relocatable label "WORK2" cannot be described as "saddrp". Therefore, an error will result as a result of this description.

Example 2

```
        NAME  SAMP2
DATA1   DSEG                 ;(1)
          :
DATA2   DSEG                 ;(2)   ←—— This description is
          :                              erroneous.
        DSEG                 ;(3)   ←—— This description is
          :                              erroneous.
DATA1   DSEG                 ;(4)
          :
        END
```

(1) A data segment with segment name "DATA1" is defined with the DSEG directive.

(2) A data segment has already been defined in (1) above. The segment name of the segment defined in (2) must be "DATA1", or an error will occur.

(3) If no segment name is specified, "DSEG" is assumed as the segment name. Therefore, an error will also occur in this case for the same reason as (2) above.

(4) This segment will be processed as a continuous segment which follows the data segment defined in (1) above. The first address of the segment defined in (4) will be the address next to the last address of the segment defined in (1) above.

(3) BSEG (bit segment)

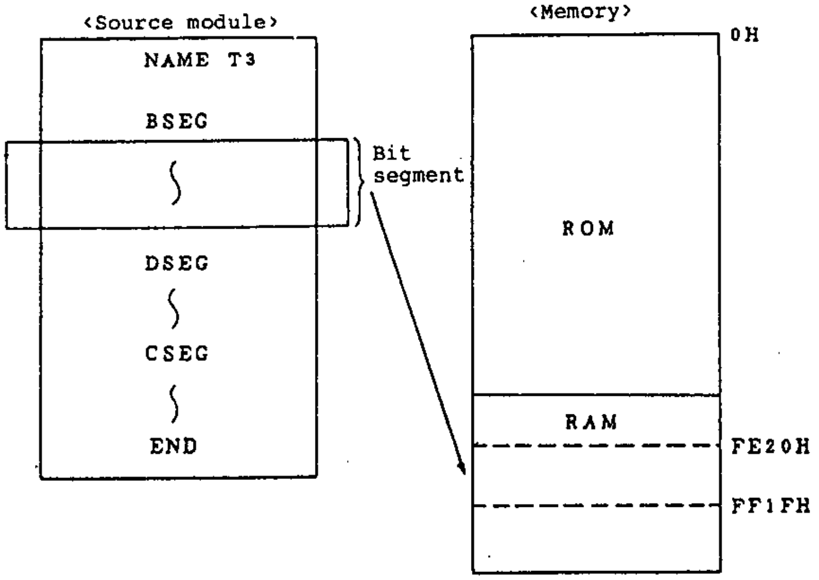
Description Format

Symbol field	Mnemonic field	Operand field	Comment field
[segment name]	BSEG	None	[;comment]

Function

- o The BSEG directive indicates to the assembler the start of a bit segment.
- o A bit segment is a segment which defines the RAM addresses to be used in the source module.
- o Memory areas defined by the DS directive after this directive until the re-appearance of any segment definition directive (CSEG, DSEG, BSEG, ORG, or ENDS) in the source module will belong to the bit segment and will be finally allocated within the RAM address area 0FE20H to 0FF1FH.

Fig. 3-4. Relocation of Bit Segment



Use

- o Describe the DBIT directive in the bit segment defined by the BSEG directive. (See Application Examples 1 and 2.)
  The bit segment will be allocated to the short direct addressing area in the RAM. Therefore, no instructions can be described in any bit segment.
  Short direct addressing area:

uPD78112	0FE40H to 0FF1FH
uPD78122, uPD78124 ...	0FE20H to 0FF1FH

- o In a bit segment, the DBIT directive must be described to reserve a one-bit work area to be used by the program and a name must be given to the bit address.
  When describing a source program, this name is used. Therefore, a bit segment defining this name should be described at the early part of the module body. (See Application Example 1.)
- o Each area reserved as a bit segment will be allocated by the locater so that it does not overlap with any other data segments or work areas.
- o A bit segment will be allocated without consideration to any byte boundary. If a bit address is to be defined with attention paid to a byte boundary, define the bit address with the EQU directive. (See Application Example 2.)
  For the bit address map, see Table 3-7 in 3.3 (1), "EQU directive".

Explanation

- o By describing a segment name in the Symbol field of the BSEG directive, the bit segment can be named.
  If no segment name is specified for a bit segment, the assembler will automatically give a default segment name "BSEG" to the bit segment.
- o All bit segments will have "BSEG" as their segment type.
- o The BSEG directive can be described only once in a source module. (See Application Example 3.)
- o Bit segments with the same segment name can be described in two or more different modules. These segments are called the same named segments.

The same named bit segments will be combined into a single data segment at linkage time.

o No labels can be defined in a source module.
The ORG directive cannot be described in a source module.
(See Application Example 3.)
The following directives and all control instructions can be used in a bit segment:
Directives: DBIT, EQU, SET, MACRO, REPT, IRP

Application Examples

Example 1

```
       NAME   SAMP1
       BSEG              ;(1)
B1     DBIT
B2     DBIT
B3     DBIT
        .
       CSEG
       MOV1   CY,B1      ;(2)
        :
       AND1   CY,B2      ;(3)
        :
       END
```

(1) A bit segment is defined with the BSEG directive. Within the bit segment, a bit work area is defined for each bit with the DBIT directive. A bit segment should be described at the early part of the module body.
(2) This description corresponds to "MOV1 CY, saddr.bit".
(3) This description corresponds to "AND1 CY, saddr.bit".

Example 2

```
        NAME  SAMP2

FLAG    EQU   0FE20H
FLAG0   EQU   FLAG.0        ;(1)
FLAG1   EQU   FLAG.1        ;(1)

        BSEG
FLAG2   DBIT                ;(2)

        CSEG
        MOV1  CY,FLAG0      ;(3)
          :
        MOV1  CY,FLAG2      ;(4)
          :
        END
```

(1) Bit addresses (Bit 0 and Bit 1 of 0FE20H) are defined with consideration given to byte address boundaries.

(2) Bit address FLAG2 defined in the bit segment is allocated without consideration to any byte address boundary.

(3) This description can be substituted with "MOV1 CY, FLAG.0". Here, FLAG indicates a byte address. in this case for the same reason as (2) above.

(4) In this description, no consideration is given to byte addresses.

Example 3

```
        NAME   SAMP3
        BSEG
        ORG    100H        ;(1)   ———— This description is
          :                            erroneous.
        BSEG               ;(2)   ———— This description is
          :                            erroneous.
        END
```

(1) This description results in an error, because the ORG directive has been described in the bit segment.

(2) The BSEG directive cannot be described more than once within the same module.

(4) ORG (origin)

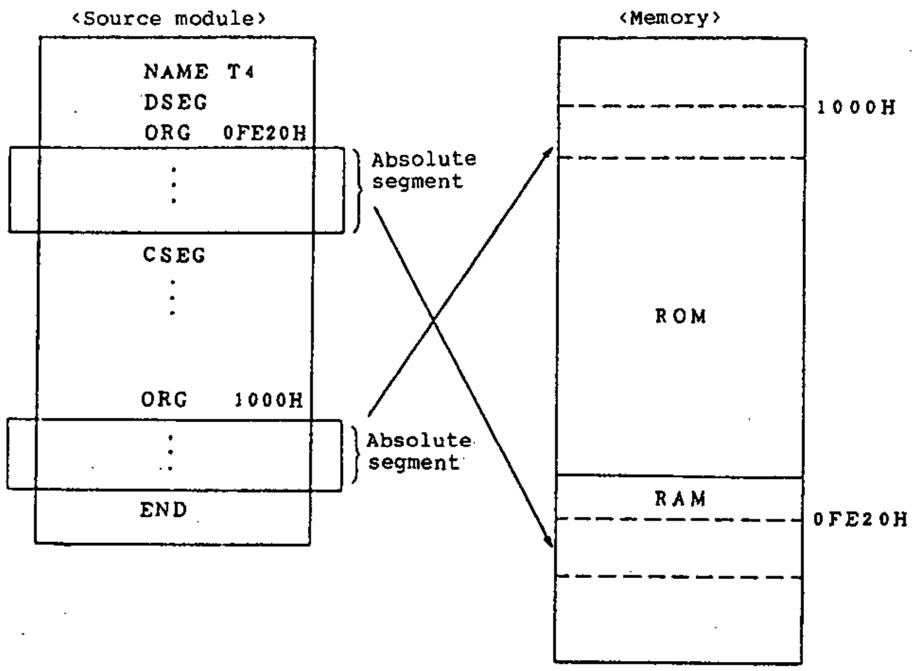
Description Format

Symbol field	Mnemonic field	Operand field	Comment field
[segment name]	ORG	expression	[;comment]

Function

- o The ORG directive sets the value of the expression specified by its operand in the location counter.
- o Instructions described or memory areas reserved after this directive until the re-appearance of any segment definition directive (CSEG, DSEG, BSEG, ORG, or ENDS) in the source module will belong to the absolute segment and will be allocated beginning with the address specified in the operand of this directive.

Fig. 3-5. Relocation of Absolute Segment



Use

- o Describe the ORG directive to start allocation to a code segment or data segment from a specific address.
- o To store an interrupt branch address in the vector table area (0H to 3FH), specify the address with the ORG directive. (See Application Example 2.)

Explanation

- o The absolute segment defined with the ORG directive belongs to the code segment or data segment defined with the CSEG or DSEG immediately before the ORG directive.
  Within an absolute segment which belongs to a data segment, no instructions can be described.
  No absolute segment can be defined following a bit segment.
- o By describing a segment name in the Symbol field of the ORG directive, the absolute segment can be named.
  If no segment name is specified for an absolute segment, the assembler will automatically give a default segment name "ASEGn" to the absolute segment (where n is a value in the range of 1 to 10 and is given in the order of segment description within the source module).
- o All absolute segments will have "ASEG" as their segment type.
- o If a name or label is to be described as the operand of the ORG directive, the name or label must be an absolute term which has already been defined in the source module.
- o Addresses 0H to 3FH are used as a vector table area, and addresses 40H to 7FH, as a table area for CALLT instruction.
  If any address lower than 7FH is specified as the operand of the ORG directive, the assembler will output a warning message to the console and an assembly list.
- o If no CSEG or DSEG directive has been described before the ORG directive, the absolute segment defined by the ORG directive is assumed to be an absolute segment in a code segment.

NOTE 3-1

The ORG directive can be described up to 10 times within a source module. An error will result if 11 or more absolute segments are described within the module.

Application Examples

Example 1

```
          NAME  SAMP1
          DSEG
SADR      ORG   0FE20H    ;(1)
SADR1:    DS    1
SADR2:    DS    1
SADR3:    DS    2
           :
MAIN0     ORG   100H
          MOV   A,SADR1   ;(2) ———— This description is
           :                          erroneous.
          CSEG            ;(3)
MAIN1     ORG   1000H     ;(4)
          MOV   A,SADR2
          MOVW  AX,SADR3
           :
          END
```

(1) An absolute segment which belongs to a data segment is defined. This absolute segment will be allocated to the short direct addressing area which starts from address "FE20H" (when the target device is uPD78112).

(2) Within an absolute segment which belongs to a data segment, no instruction can be described.

(3) This directive declares the start of a code segment.

(4) This absolute segment will be allocated to an area which starts from address "1000H".

Example 2

```
        NAME  SAMP2
        ORG   0H          ;(1)
         :
        CSEG              ;(2)
        ORG   800H        ;(3)
         :
        ORG   1000H       ;(4)
         :
        END
```

(1) This absolute segment belongs to a code segment and will be allocated to an area which starts from address "0H".

(2) This directive indicates the start of a code segment.

(3) This absolute segment will be allocated to an area which starts from address "800H".

(4) This absolute segment will be allocated to an area which starts from address "1000H".

Three absolute segments are defined in (1), (3), and (4) above, without segment name. Therefore, the assembler will automatically give names ASEG1, ASEG2, and ASEG3 to the segments defined in (1), (3), and (4), respectively.

(5) ENDS (end of segment)

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
None	ENDS	None	[;comment]

Function

The ENDS directive indicates the end of the relocatable segment defined by the CSEG, DSEG, or BSEG directive.

Use

The ENDS directive is used in pairs with the CSEG, DSEG, or BSEG directive.

Explanation

- o The ENDS directive indicates the end of each segment in source module description.
- o After the ENDS directive has been described, only comments can be described before the next segment definition directive (CSEG, DSEG, BSEG, or ORG) is described.
- o Description of the ENDS directive may be omitted.

Application Example

```
        NAME    SAMP1
        BSEG
          :
        ENDS            ;(1)
;data   segment
        DSEG            ;(2)
          :
        ENDS            ;(3)
WORK:   DS      1       ;(4)    ——— This description is
                                    erroneous.
        CSEG            ;(5)
          :
        ENDS            ;(6)
        END
```

(1) This directive indicates the end of a bit segment.

(2) This directive indicates the start of a data segment. Only comments can be described between (1) and (2).

(3) This directive indicates the end of the data segment.

(4) Only comments can be described between the ENDS directive and the CSEG directive in (5) below.

(5) This directive indicates the start of a code segment.

(6) This directive indicates the end of the code segment.

3.3 Directives for Symbol Definition

Symbol definition directives assign names to numerical data to be used for describing a source module. By these names, the meaning of each data value becomes clear and you may easily understand the contents of the source module.
Symbol definition directives inform the assembler of the value of each name to be used in the source module.
The definitions of names with symbol definition directives must be described in the module header.
Two directives are available for symbol definition: EQU and SET.

(1) EQU (equate)

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
name	EQU	expression	[;comment]

Function

The EQU directive defines a name which has the value and attributes (symbol attribute and relocation attribute) of the expression specified in the Operand field.

Use

Define a numerical data to be used in the source module as a name and describe it in the operand of the directive in place of the numerical data.
Numerical data to be frequently used in the source module should be defined as names. By so doing, if you must change a data value in the source module, all you need to do is to change the operand value of the name. (See Application Example 1.)

Explanation

- o When a name or label is to be described in the operand of the EQU directive, use the name or label which has already been defined in the source module.
- o Because the EQU directive defines data to be used in a source module, define the name for the data in the module header of the source module.
- o If an error exists in the statement in which a name has been defined with the EQU directive, the name will not be stored in memory. An error will also occur in the statement in which the name is referenced.

- o A name defined with the EQU directive cannot be defined again within the same source module.
- o Any of the bit values shown in Table 3-6 can be used as the operand of the EQU directive. A name which has defined any of the bit values can be referenced only within the same module. However, a name which has defined "saddr.bit" can be used as an external definition symbol.

Table 3-6. Representation Formats of Operands Indicating Bit Values

Representation format of operand	Range in which name can be referenced
saddr.bit	A name which has defined this bit value can be referenced from another module.
sfr.bit	A name which has defined any of these bit values can be referenced only within the same module.
A.bit	
X.bit	
PSW.bit	

- o A name which has defined a bit value with the EQU directive has a bit address as its value. See Table 3-7 for the bit address map.

Application Examples

Example 1

```
          NAME    SAMP1

DATA1     EQU     10H             ;(1)
DATA2     EQU     20H

ADRS1     EQU     0FE20H          ;(2)
ADRS2     EQU     0FE21H

          CSEG


          MOV     A,#DATA1        ;(3)


          ADD     A,ADRS1         ;(4)


          MOV     ADRS2,#DATA1    ;(5)


          END
```

(1) Name "DATA1" has value "10H", symbol attribute "NUMBER", and relocation attribute "ABSOLUTE".

(2) Name "ADRS1" has value "0FE20H", symbol attribute "NUMBER", and relocation attribute "ABSOLUTE".

(3) Name "DATA1" defined in (1) above is described as the operand of the MOV instruction with a value of 10H.

(4) Name "ADRS1" defined in (2) above is described as the operand of the ADD instruction with a value of 0FE20H.

(5) Names "ADRS2" and "DATA1" which have already been defined are described as the operands of the MOV instruction.

If the value "10H" defined as "DATA1" must be changed to 50H, you only need to change 10H to 50H in the directive description (1). Descriptions (3) and (5) need not to be changed.

Example 2

```
        NAME    SAMP2

WORK1   EQU     0FE40H          ;(1)
WORK10  EQU     WORK1.0         ;(2)
WORK11  EQU     WORK1.1         ;(2)

P12     EQU     P1.2            ;(3)
P13     EQU     P1.3            ;(3)

A4      EQU     A.4             ;(4)
X5      EQU     X.5             ;(5)

PSW5    EQU     PSW.5           ;(6)
PSW6    EQU     PSW.6           ;(7)

        CSEG


        MOV1    CY,WORK10       ;(8)


        MOV1    P12,CY          ;(9)


        OR1     CY,A4           ;(10)


        XOR1    CY,X5           ;(11)


        SET1    PSW5            ;(12)


        CLR1    PSW6            ;(13)


        END
```

(1) Name "WORK1" has value "02FE40H", symbol attribute "NUMBER", and relocation attribute "ABSOLUTE".

(2) Bit values "WORK1.0" and "WORK1.1" which are in the operand format "saddr.bit" are assigned names "WORK10" and "WORK11", respectively. Value "0FE40H" has already been defined in (1) for "WORK1" described as the operand of the EQU directive.

(3) Bit values "P1.2" and "P1.3" which are in the operand format "sfr.bit" are assigned names "P12" and "P13", respectively.

(4) Bit value "A.4" which is in the operand format "A.bit" is assigned name "A4".

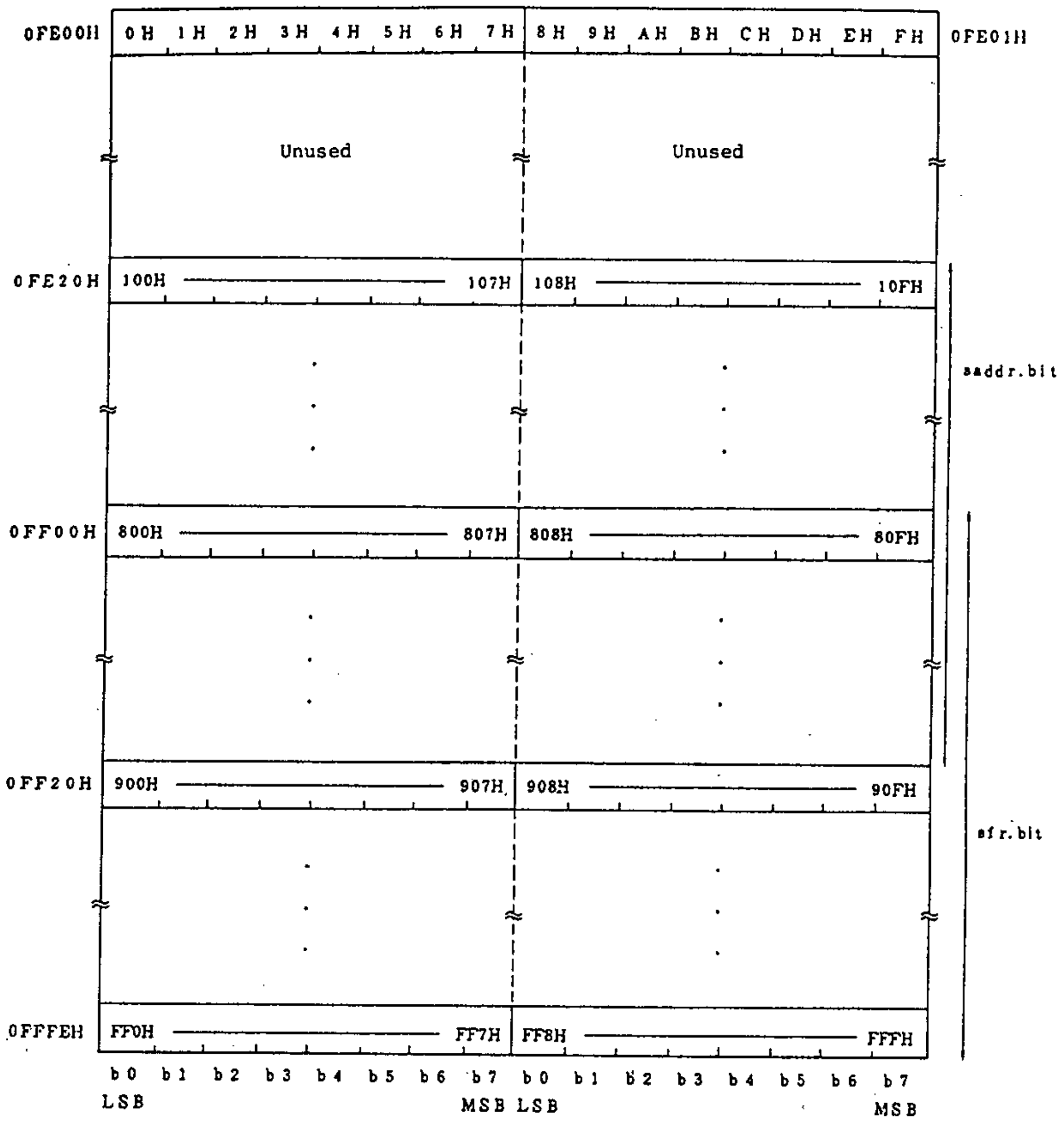
(5) Bit value "X.5" which is in the operand format "X.bit" is assigned name "X5".
(6) Bit value "PSW.5" which is in the operand format "PSW.bit" is assigned name "PSW5".
(7) Bit value "PSW.6" which is in the operand format "PSW.bit" is assigned name "PSW6".
(8) This description corresponds to "MOV1 CY, saddr.bit".
(9) This description corresponds to "MOV1 sfr.bit, CY".
(10) This description corresponds to "OR1 CY, A.bit".
(11) This description corresponds to "XOR1 CY, X.bit".
(12) This description corresponds to "SET1 PSW.bit".
(13) This description corresponds to "CLR1 PSW.bit".

Names which have defined "sfr.bit", "A.bit", "X.bit", and "PSW.bit" as in (3) through (7), can be referenced only within the same module.
A name which has defined "saddr.bit" can also be referenced as an external definition symbol from another module. (See 3.5 (2), "EXTBIT directive".)
As a result of assembling the source module in Example 2, the following assembly list is generated. See the object codes output on this list. (Also see APPENDIX C-2 for the instruction codes.)

```
ASSEMBLE LIST

STNO  ADRS R OBJECT     M I  SOURCE STATEMENT

  1                                 NAME    SAMP2

  2            FE40          WORK1  EQU     0FE40H         ;(1)
  3            0200          WORK10 EQU     WORK1.0        ;(2)
  4            0201          WORK11 EQU     WORK1.1        ;(2)

  5            080A          P12    EQU     P1.2           ;(3)
  6            080B          P13    EQU     P1.3           ;(3)

  7            000C          A4     EQU     A.4            ;(4)
  8            0005          X5     EQU     X.5            ;(5)

  9            0FF5          PSW5   EQU     PSW.5          ;(6)
 10            0FF6          PSW6   EQU     PSW.6          ;(7)

 11                                 CSEG


 12  0000      080040               MOV1    CY,WORK10      ;(8)


 13  0003      081A01               MOV1    P12,CY         ;(9)


 14  0006      034C                 OR1     CY,A4          ;(10)


 15  0008      0365                 XOR1    CY,X5          ;(11)


 16  000A      0285                 SET1    PSW5           ;(12)


 17  000C      0296                 CLR1    PSW6           ;(13)


 18                                 END
```

On lines (2) through (7) of the assembly list, bit address values for the bit values defined as names are indicated in the OBJECT (code) column. Each bit address becomes a value as shown in Table 3-7. A bit address is a value to be given for the convenience of assembly processing and the value itself has no meaning.

Table 3-7. Bit Address Map



NOTE: MSB ... Most significant bit
LSB ... Least significant bit

(2) SET (set)

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
name	SET	expression	[;comment]

Function

- o The SET directive defines a name which has the value and attributes (symbol attribute and relocation attribute) of the expression specified in the Operand field.
- o The value and attribute of a name defined with the SET directive can be re-defined within the same module.
- o The value and attribute of a name defined with the SET directive are valid until the same name is re-defined.

Use

Define a numerical data (variable) to be used in the source module as a name and describe it in the operand of the directive in place of the numerical data (variable).
If you wish to change the value of a name in the source module, a different value can be set in the same name with the SET directive.

Explanation

- o When a name is to be described in the operand of the SET directive, use the name which has already been defined in the source module.
- o The following items cannot be described as the operand of the SET directive:
  - External reference term
  - Label
  - Name which has BIT attribute and BIT value

o A name which has been defined with the SET directive can be referenced only within the same source module.

o If an error exists in the statement in which a name has been defined with the SET directive, the name will not be stored in memory. An error will also occur in the statement in which the name is referenced.

Application Example

```
        NAME   SAMP1
COUNT   SET    10H          ;(1)
        CSEG
         :
        MOV    B,#COUNT     ;(2)
LOOP:    :
        DEC    B
        BNZ    LOOP
         :
COUNT   SET    20H          ;(3)
         :
        MOV    B,#COUNT     ;(4)
         :
        END
```

(1) Name "COUNT" has value "10H", symbol attribute "NUMBER", and relocation attribute "ABSOLUTE". The value and attributes are valid until re-definition in (3).

(2) The value "10H" of name "COUNT" will be transferred to register B.

(3) The value of name "COUNT" is changed to 20H.

(4) The value "20H" of name "COUNT" will be transferred to register B.

3.4 Directives for Memory Initialization and Area Reservation

Memory initializing directives define constant data to be used in a source program. The value of the defined constant data is generated as an object code.

Area reservation directives reserve memory areas to be used by the source program.

(1) DB (define byte)

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
[label:]	DB	[(size)][initial value[,...]]	[;comment]

Function

The DB directive initializes a memory area in byte units with the initial value(s) specified in the Operand field.
The number of bytes to be initialized can be specified as "size".

Use

The DB directive is used when defining a constant (numeric constant or character constant).

Explanation

o The following three parameters can be specified as initial values:

① Constant

Numeric constant: Must be a constant consisting of not more than 8 bits. (Its value must be in the range of 0 to 255.)

Character constant: Must be one character (7-bit ASCII code).

② Name

The value of a name that can be described as the operand must be 8 bits or less. Its symbol attribute is "NUMBER" and its relocation attribute, "ABSOLUTE".

③ Expression

The value of an expression that can be described as the operand must be an absolute term consisting of not more than 8 bits.

- o If both "size" and "initial value" are specified, the assembler checks if the specified size is equal to the specified initial value in the number of bytes.
- o If a size is specified but no initial value is specified, an area equivalent to the number of bytes specified by the size is initialized with value "00H".
- o The DB directive cannot be described within a bit segment.

Application Example

```
          NAME SAMP1
          CSEG
WORK1   :DB     (1)             ;(1)
WORK2   :DB     (2)             ;(1)
          CSEG
MASSAG:DB       'ABCDEF'        ;(2)
DATA1   :DB     0AH,0BH,0CH     ;(3)
DATA2   :DB     (3)100,101,102  ;(4)
DATA3   :DB     (2)0AH          ;(5) ←—— This description is
          :                               erroneous.
          END
```

(1) By specifying (size) only, each byte area is initialized with value "00H".

(2) This directive initializes a 6-byte area with character constant "ABCDEF".

(3) This directive initializes a 3-byte area with numeric constants "0AH, 0BH, 0CH".

(4) This directive initializes a 3-byte area with numeric constants "100, 101, 102".
The assembler checks if the specified size agrees with the initial values in the number of bytes.

(5) Because the specified size does not agree with the initial value in the number of bytes, this description will result in an error.

(2) DW (define word)

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
[label:]	DW	[(size)][initial value[,...]]	[;comment]

Function

The DW directive initializes a memory area in word units (i.e., in units of 2 bytes) with the initial value(s) specified in the Operand field.
The number of words to be initialized can be specified as "size".

Use

The DW directive is used when defining a 16-bit numeric constant such as an address or data.

Explanation

o The following three parameters can be specified as initial values:
  ① Numeric constant
  Must be a constant consisting of not more than 16 bits.
  ② Name or label
  Must be a name or label with a value of not more than 16 bits.
  A name with BIT attribute cannot be described as the operand of this directive.
  ③ Expression
  Must be an expression with a value of not more than 16 bits.

o If both "size" and "initial value" are specified, the assembler checks if the specified size is equal to the specified initial value in the number of words.

o If a size is specified but no initial value is specified, an area equivalent to the number of words specified by the size is initialized with value "00H".
o The DW directive cannot be described within a bit segment.

Application Example

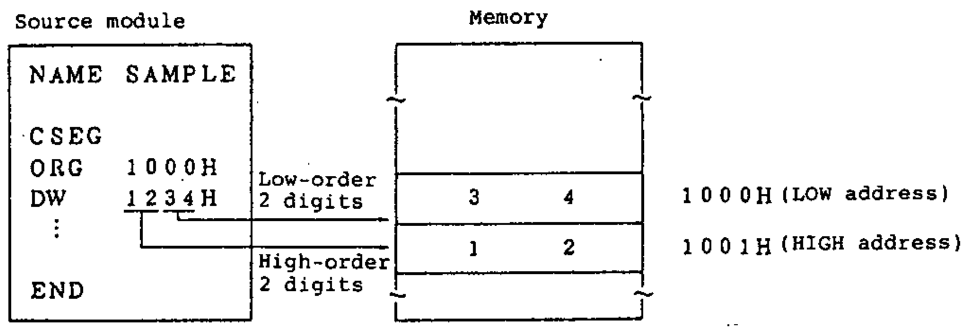
```
         NAME   SAMPLE
         CSEG
WORK1:   DW     (10)                 ;(1)
WORK2:   DW     (128)                ;(1)
         CSEG
         ORG    0H
         DW     MAIN                 ;(2)
         DW     SUB1                 ;(2)
         CSEG
MAIN  :   :
         CSEG
SUB1  :   :
DATA  :  DW     (2)1234H,5678H       ;(3)

         END
```

(1) By specifying (size) only, each word area is initialized with value "00H".
(2) Vector entry addresses are defined with the DW directive.
(3) This directive initializes a 2-word area with value "34127856".

NOTE 3-2

With a word value, the HIGH (high-order) address of memory is initialized with the high-order 2 digits of the value and the LOW (low-order) address of memory is initialized with the low-order 2 digits of the value.

Example:



(3) DS (define storage)

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
[label:]	DS	expression	[;comment]

Function

The DS directive reserves a memory area for the number of bytes specified in the Operand field.

Use

The DS directive is mainly used to reserve a memory (RAM) area to be used by a source program. If a label is specified, the value of the first address of the reserved memory area is assigned to the label. In the source module, this label is used for description to manipulate the memory.

Explanation

o The contents of an area reserved with this directive are unknown.

o When describing a name or label as the operand of this directive, the name or label must be an absolute term which has already been defined in the source module.

o If a label is specified in the Symbol field, the label will have the first address of the reserved area as its value.

o If a value of 0 is given to the operand of this directive, no memory area will be reserved.

o The DS directive cannot be described within a bit segment.

Application Example

```
           NAME  SAMPLE
           DSEG
TABLE1 :DS       10         ;(1)
WORK1  :DS       1          ;(2)
WORK2  :DS       2          ;(3)
           CSEG
           MOVW  HL,#TABLE1
             :
           MOV   A,WORK1
             :
           MOVW  BC,#WORK2
             :
           END
```

(1) This directive reserves a 10-byte work area, but the contents of the area are unknown. Label "TABLE1" is assigned to the first address of the area.

(2) This directive reserves a 1-byte work area.

(3) This directive reserves a 2-byte work area.

(4) DBIT (define bit)

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
[name]	DBIT	None	[;comment]

Function

The DBIT directive reserves a 1-bit memory area within a bit segment.

Use

The DBIT directive is used to reserve a bit area within a bit segment.

Explanation

- o The DBIT directive is described only in a bit segment.
- o The contents of a 1-bit area reserved with this directive are unknown.
- o If a name is specified in the Symbol field, the name will have a bit address as its value. See Table 3-7 for the bit address map.

Application Example

```
        NAME  SAMPLE
        BSEG
BIT1    DBIT              ;(1)
BIT2    DBIT              ;(1)
BIT3    DBIT              ;(1)
        CSEG
        MOV1  CY,BIT1     ;(2)
          ⋮
        OR1   CY,BIT2     ;(3)
          ⋮
        END
```

(1) These three DBIT directives reserve a 1-bit area and define names (BIT1, BIT2, and BIT3) each having a bit address.

(2) This instruction corresponds to "MOV1 CY,saddr.bit" and describes name "BIT1" of the bit area reserved in (1) above as operand "saddr.bit".

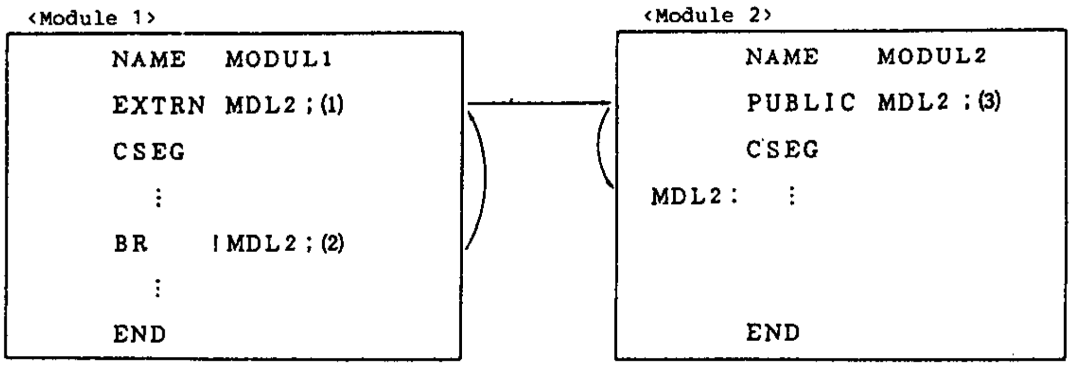
(3) This instruction corresponds to "OR1 CY,saddr.bit" and describes name BIT2 as "saddr.bit".

3.5 Directives for Linkage

Linkage directives function to make clear the relation between the external definition of a symbol and its external reference. Let's consider a case where a program is created by being divided into two modules: Module 1 and Module 2. In Module 1, if you wish to reference a symbol defined in Module 2, the symbol cannot be use without declaration in each module. For this reason, some sort of signal or indication as "I want to use the symbol" and "You may use the symbol" is required between the two modules.
In Module 1, the external reference declaration of a symbol must be made to indicate that a symbol defined in another module must be referenced. On the other hand, in Module 2, the external definition declaration of a symbol must be made to indicate that the symbol may be referenced in another module.
The symbol can be referenced for the first time when the two external reference and external definition declarations are effectively made.
Linkage directives function to establish this interrelationship and are available in the following types:

- o To declare external reference of symbol: EXTRN and EXTBIT directives
- o To declare external definition of symbol: PUBLIC directive

Fig. 3-6. Relationship of Symbols between Two Modules



<Module 1>

```
        NAME   MODUL1
        EXTRN  MDL2 ;(1)
        CSEG
          :
        BR     !MDL2 ;(2)
          :
        END
```

<Module 2>

```
        NAME    MODUL2
        PUBLIC  MDL2 ;(3)
        CSEG
MDL2 :    :


        END
```

In Module 1 in Fig. 3-6, the symbol "MDL2" defined in Module 2 is referenced in (2). Therefore, the symbol is declared as an external reference with the EXTRN directive in (1).
In Module 2, the symbol "MDL2" to be referenced from Module 1 is declared as an external definition in (3).
The linker checks whether or not this external reference of the symbol corresponds to the external definition of the symbol.

(1) EXTRN (external)

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
[label:]	EXTRN	symbol name[,...]	[;comment]

Function

The EXTRN directive declares that the symbol described in the Operand field is to be referenced in this module.
The specified symbol has been defined in another module.

Use

When referencing a symbol defined in another module, the EXTRN directive must be used to declare the symbol as an external reference.

Explanation

- o The EXTRN directive must be described in the module header of a source module. (See Section 2.1, "Basic Configuration of Source Program".)
- o Two or more symbols may be specified in the Operand field by delimiting each symbol name with a comma (,).
- o When referencing a symbol having a bit value, the symbol must be declared as an external reference with the EXTBIT directive.
- o No macro name can be described as the operand of the EXTRN directive. (See Chapter 5, Macro for the macro name.)

Application Example

<Module 1>

```
        NAME  SAMP1
        EXTRN SADR1,SADR2 ;(1)
        CSEG
          :
        MOV   A,SADR1     ;(2)
          :
        MOVW  DE,#SADR2   ;(3)
        MOVW  AX,[DE]
          :
        END
```

<Module 2>

```
        NAME    SAMP2
        PUBLIC  SADR1,SADR2 ;(4)
        DSEG
        ORG     0FE20H
SADR1:  DS      1           ;(5)
SADR2:  DS      2           ;(6)
          :


        END
```

(1) This directive declares symbols "SADR1" and "SADR2" to be referenced in (2) and (3), respectively, as external references.
Two or more symbols may be described in the Operand field.

(2) This instruction references symbol "SADR1".

(3) This instruction references symbol "SADR2".

(4) This directive declares symbols "SADR1" and "SADR2" as external definitions.

(5) This directive defines symbol "SADR1".

(6) This directive defines symbol "SADR2".

(2) EXTBIT (external bit)

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
[label:]	EXTBIT	symbol name[,...]	[;comment]

Function

The EXTBIT directive declares that the symbol having a bit value, described in the Operand field is to be referenced in this module. The specified symbol has been defined in another module.

Use

When referencing a symbol having a bit value (saddr.bit) defined in another module, the EXTBIT directive must be used to declare the symbol as an external reference.

Explanation

- o The EXTBIT directive must be described in the module header of a source module.
- o Two or more symbols may be specified in the Operand field by delimiting each symbol name with a comma (,).
- o Only the bit value of a symbol which can be represented in the operand format "saddr.bit" can be described as the operand of the EXTBIT directive.

Table 3-8. Bit Value That Can Be Described as Operand of EXTBIT

Representation Format of operand	Description as operand of EXTBIT directive
saddr.bit	Allowed
sfr.bit	Not allowed
A.bit	
X.bit	
PSW.bit	

Application Example

<Module 1>

```
        NAME    SAMP1
        EXTBIT  FLAG1,FLAG2;(1)
        CSEG
          :
        MOV1    CY,FLAG1;(2)
          :
        OR1     CY,FLAG2;(3)
          :
        END
```

<Module 2>

```
        NAME    SAMP2
        PUBLIC  FLAG1, FLAG2;(4)
FLAG1   EQU     0FE20H.0    ;(5)
FLAG2   EQU     0FE20H.1    ;(6)
        CSEG
          :


        END
```

(1) This directive declares symbols "FLAG1" and "FLAG2" to be referenced in (2) and (3), respectively, as external references.
Two or more symbols may be described in the Operand field.
(2) This instruction references symbol "FLAG1". This description corresponds to "MOV1 CY,saddr.bit".
(3) This instruction references symbol "FLAG2". This description corresponds to "OR1 CY,saddr.bit".
(4) This directive declares symbols "FLAG1" and "FLAG2" as external definitions.
(5) This directive defines symbol "FLAG1".
(6) This directive defines symbol "FLAG2".

(3) PUBLIC (public)

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
[label:]	PUBLIC	symbol name[,...]	[;comment]

Function

The PUBLIC directive declares that the symbol described in the Operand field is a symbol to be referenced from another module.

Use

When defining a symbol to be referenced from another module, the PUBLIC directive must be used to declare the symbol as an external definition.

Explanation

- o The PUBLIC directive must be described in the module header of a source module.
- o Two or more symbols may be specified in the Operand field by delimiting each symbol name with a comma (,).
- o Symbol(s) to be described in the Operand field must have been defined within the same source module.
- o The following symbols cannot be used as the operand of the PUBLIC directive:
  - . Name defined with the SET directive
  - . Symbol defined with the EXTRN directive within the same module
  - . Name with a bit value other than "saddr.bit"
  - . Segment name
  - . Module name
  - . Macro name

NOTE 3-3

External declaration of segment name or module name:
Even if a segment name or module name is declared with the PUBLIC or EXTRN directive, no error will occur. However, this declaration becomes invalid and if these symbols are referenced, an error will result at linkage time.

Application Example

Example of program consisting of three modules

<Module 1>

```
        NAME    SAMP1
        PUBLIC  A1,A2    ;(1)
        EXTRN   B1,C1

A1      EQU     10H
A2      EQU     0FE20H.1

        CSEG
          :
        BR      !B1
          :
        XOR1    CY,C1
          :
        END
```

<Module 2>

```
        NAME    SAMP2
        PUBLIC  B1       ;(2)
        EXTRN   A1
        CSEG
B1:       :
        MOV     C,#A1
          :
        END
```

<Module 3>

```
        NAME    SAMP3
        PUBLIC  C1       ;(3)
        EXTRN   A2
C1      EQU     0FE21H.0

        CSEG
          :
        MOV1    CY,A2
          :
        END
```

(1) This directive declares that symbols "A1" and "A2" are to be referenced from other modules.

(2) This directive declares that symbol "B1" is to be referenced from another module.

(3) This directive declares that symbol "C1" is to be referenced from another module.

(4) NAME (name)

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
[label:]	NAME	module name[,...]	[;comment]

Function

The NAME directive gives (assigns) the module name described in the Operand field to an object module to be output by the assembler.

Use

A module name is required for symbolic debugging of an object module with a debugger.

Explanation

- o The NAME directive must be described in the module header of a source module. (See Section 2.1, "Basic Configuration of Source program" in Chapter 2.)
- o For the conventions of module name description, see Subsection 2.2.3, "Symbol field" in Chapter 2.
- o No module name can be described as the operand of any directive other than NAME or of any instruction.
- o The NAME directive must be described for each source module. If this directive is omitted, an error will result.

Application Example

```
NAME SAMPLE ;(1)
DSEG
  :
CSEG
  :
END
```

(1) This directive declares module name "SAMPLE".

3.6 Directive for Automatic Selection of BR Instruction

As the unconditional branch instructions of the uCOM-78K/I, which directly describe a branch destination address as their operand, two instructions "BR !addr16(addr13)" and "BR $addr16(addr13)" are available.
The BR !addr16(addr13) instruction is a three-byte instruction which allows branching to any address, whereas the BR $addr16 (addr13) instruction is a two-byte instruction which allows branching to an address within the range of -126 to +129 bytes from the current location counter value.
Therefore, to create a program with high memory utilization efficiency, the 2-byte instruction "BR $addr16(addr13)" must be described according to the address range of the branch destination. However, it is quite troublesome to take this address range into account when you describe the branch instruction.
For this reason, there was a need for a directive which directs the assembler to automatically select the two-byte or three-byte branch instruction according to the address range of the branch destination. The BR directive is provided for this purpose.
However, note that this directive is valid only when the assembler option "OPTIMIZE" is specified. (For the OPTIMIZE option, see Subsection 4.4.4, "Description of each assembler option" in Chapter 4 of the RA78K/I Assembler Package User's Manual for Operation.)

(1) BR (branch)

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
[label:]	BR	expression	[;comment]

Function

The BR directive causes the assembler to automatically select the 2-byte or 3-byte branch instruction according to the value range of the expression specified in the Operand field and to generate the object code applicable to the selected instruction. This function is referred to as "optimization of branch instructions".
The Optimize function is valid only when the assembler option "OPTIMIZE" is specified.

Use

o If the branch destination is within the range of -126 to +129 bytes from the current location counter, you can describe the 2-byte branch instruction "BR $addr16(addr13)". With this instruction, required memory space can be reduced by one byte as compared with that when using the 3-byte branch instruction. To create a program with high memory utilization efficiency, the 2-byte branch instruction should be used positively. However, each time you describe a branch instruction, it is troublesome for you to take into account the address range of the branch destination. So, use the BR directive when you are not sure of whether or not the the 2-byte branch instruction can be described.

- o If it is definite that you can describe the 2-byte or 3-byte branch instruction, describe the applicable branch instruction. In this case, the assembly time can be shortened as compared with that when the BR directive is described.

Explanation

- o If the assembler option "OPTIMIZE" is specified, the assembler performs the following processes for the BR directive. In the following explanation, the term "expression" refers to the expression described in the Operand field.
  - (a) When BR directive is described within a relocatable segment
    - If the expression is a label within the same segment, the optimization phase is executed.
    - If the expression is a label within another segment or an externally referenced term, the object code of the 3-byte branch instruction is generated.
    - If the relocation attribute of the expression is an absolute term, the object code of the 3-byte branch instruction is generated.
  - (b) When BR directive is described within an absolute segment
    - If the relocation attribute of the expression is an absolute term, the optimize phase is executed.
    - If the relocation attribute of the expression is a relocatable term or externally referenced term, the object code of the 3-byte branch instruction is generated.

Table 3-9. Optimization Conditions of BR Directive

Jump condition (source) / Jump condition (destination)		Absolute segment		Relocatable segment	
Reference		Backward	Forward	Backward	Forward
Numeric value		Optimize	Optimize	3-byte BR	3-byte BR
Name (symbol attribute: NUMBER)		Optimize	Optimize	3-byte BR	3-byte BR
Label	Same segment	Optimize	Optimize	Optimize	Optimize
	Same named segment	Optimize	Optimize	Optimize	Optimize
	Other segment (same type)	Optimize	Optimize	-	-
	Other segment (Other type)	3-byte BR	3-byte BR	3-byte BR	3-byte BR
External reference name		3-byte BR	-	3-byte BR	-
Location counter ($)		Optimize	-	Optimize	

NOTE: o "-" in the table indicates that the combination is prohibited.

o "Backward" reference denotes the reference of a symbol which has already been defined in the source module.

o "Forward" reference denotes the reference of a symbol which is to be defined in a subsequent line.

o If the assembler option "OPTIMIZE" is omitted, the assembler generates the object code of the 3-byte branch instruction for all BR directives in the source module.

o "$" indicating the current location counter cannot be specified as the operand of the BR directive.

Application Example

```
        NAME  SAMPLE
        CSEG                      ┐
          :                       │
L1:     MOV   A, #10H             │  Relocatable
          :                       │  segment
        BR    L1        ; (1)     │
        BR    L2        ; (2)     │
          :                       ┘
        ORG   1000H               ┐
L2:       :                       │
        BR    L2        ; (3)     │  Absolute
        BR    L1          (4)     │  segment
          :                       ┘
        END
```

(1) This BR directive will be optimized.
If displacement between the line (1) and the "L1:" label definition is within -126 bytes, the object code of the 2-byte branch instruction will be generated.

(2) This BR directive will be substituted with the 3-byte branch instruction, because it branches to a label in another segment.

(3) This BR directive will be optimized.
If displacement between the line (3) and the "L2:" label definition is within -126 bytes, the object code of the 2-byte branch instruction will generated.

(4) Because the relocation attribute of "L1" described as the operand of this BR directive is a relocatable term, the object code of the 3-byte branch instruction will be generated.

3.7 Macro Directives

When you describe a source program, it is troublesome for you to describe a series of frequently used instruction groups over and over again, and this may cause an increase in the number of description or coding errors.
By using the macro function with macro directives, the need to repeatedly describe the same group of instructions can be eliminated, thereby increasing coding efficiency of the program.
The basic function of a macro is the substitution of a series of statements with a name. For details of the macro function, see Chapter 5, Macros.
Macro directives include MACRO, LOCAL, REPT, IRP, EXITM, and ENDM.
In this section, each of these directives is detailed.

(1) MACRO (macro)

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
macro name	MACRO	[formal parameter[,...]]	[;comment]
	≀		
	ENDM		

Function

The MACRO directive executes a macrodefinition by assigning the macro name specified in the Symbol field to a series of statements (called a macro body) described between this directive and the ENDM directive.

Use

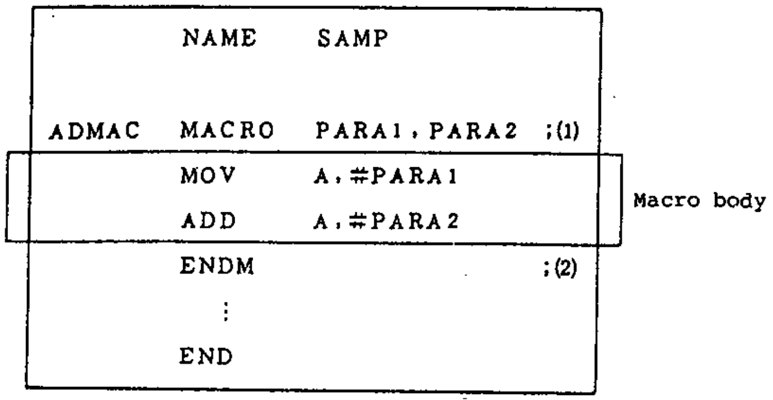
Define a series of frequently used statements in the source program with a macro name. After the macrodefinition, you only need to describe the defined macro name (for macro reference) and the macro body corresponding to the macro name will be expanded.

Explanation

- For the macro name to be described in the Symbol field, see the conventions of symbol description in Subsection 2.2.3, "Symbol field" in Chapter 2.
- To reference a macro, describe the defined macro name in the Mnemonic field. (See Application Example.)
- For the formal parameter(s) to be described in the Operand field, the same rules as the conventions of symbol description will apply.
- Description of formal parameter(s) within a macro body is limited to the part of the Operand field where constant(s) are to be described.
- Formal parameters are valid only within a macro body.

- o Because formal parameters are handled as symbols, the same named parameters cannot be described within the same source module.
- o A name or label defined within a macro body must be declared as a local symbol with the LOCAL directive.
- o Nesting of macros (i.e., to refer to other macros within a macro body) is allowed up to eight levels.
- o Up to 10 macros can be defined within a single source module.

<u>Application Example</u>



```
        NAME     SAMP

ADMAC   MACRO    PARA1,PARA2   ;(1)
        MOV      A,#PARA1
        ADD      A,#PARA2
        ENDM                   ;(2)
          :
        END
```

(1) A macro is defined with macro name "ADMAC" and two formal parameters "PARA1" and "PARA2" specified.

(2) This directive indicates the end of the macrodefinition.

(2) LOCAL (local)

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
[label:]	LOCAL	symbol name[,...]	[;comment]

Function

The LOCAL directive declares that the symbol name specified in the Operand field is a local symbol which is valid only within the macro body.

Use

Use the LOCAL directive if you wish to reference a macro defining symbol(s) within its body more than once.

Explanation

- o If a symbol is defined within a macro body and the macro is referenced more than once, it means that the symbol would be defined more than once in the source module. For this reason, it is necessary to declare that the symbol is a local symbol which is valid only within the macro.
- o The LOCAL directive can be used only within a macro-definition.
- o The LOCAL directive must be described before the symbol specified in the Symbol field is defined. (In other words, the directive must be described at the beginning of the macro body.)
- o Symbol names to be defined with the LOCAL directive within a single source module must be all different. (In other words, the same name cannot be used for local symbols to be used in each source module.)
- o Up to 15 symbol names can be specified in the Operand field.

o Symbols defined with the LOCAL directive cannot be called (referenced) from outside the macro.

Application Example

<Source program>

```
         NAME     SAMPLE
MAC1     MACRO
         LOCAL    LLAB      ;(1)
LLAB:      :                         Macrodefinition
         BR       $LLAB     ;(2)
         ENDM
           :
REF1:    MAC1               ;(3)
           :
         BR       !LLAB     ;(4)  ----This description is
           :                          erroneous.
REF2:    MAC1               ;(5)
           :
         END
```

(1) This directive defines symbol name "LLAB" as a local symbol.

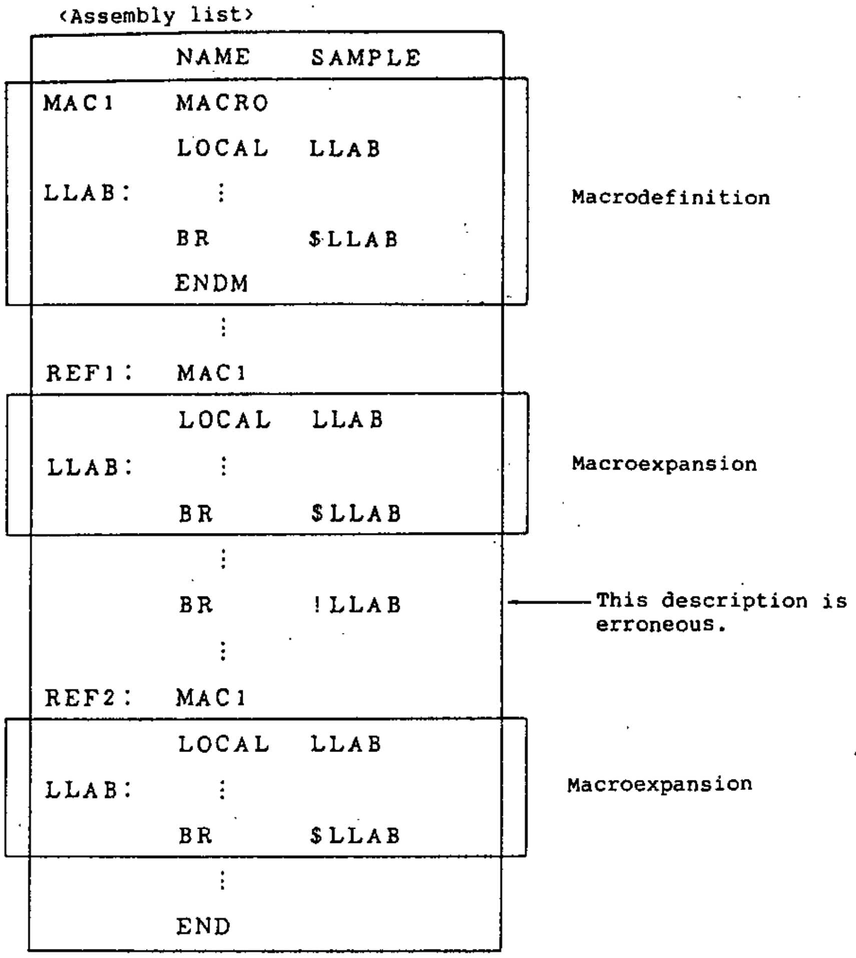
(2) This instruction references local symbol "LLAB" within macro MAC1.

(3) This directive references macro MAC1.

(4) Because local symbol "LLAB" is referenced outside the definition of macro MAC1, this description causes an error.

(5) This directive references macro MAC1.

If the source program in the above example is assembled, macroexpansion (replacement of a macrocall with the body itself) occurs as shown below.



‹Assembly list›

```
        NAME    SAMPLE
MAC1    MACRO
        LOCAL   LLAB
LLAB:     :
        BR      $LLAB
        ENDM
          :
REF1:   MAC1
        LOCAL   LLAB
LLAB:     :
        BR      $LLAB
          :
        BR      !LLAB
          :
REF2:   MAC1
        LOCAL   LLAB
LLAB:     :
        BR      $LLAB
          :
        END
```

(3) REPT (repeat)

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
[label:]	REPT	expression	[;comment]
	≀		
	ENDM		

Function

The REPT directive causes the assembler to repeatedly expand a series of statements described between this directive and the ENDM directive (called the REPT-ENDM block) the number of times equivalent to the value of the expression specified in the Operand field.

Use

If a series of statements is to be described repeatedly in a source program, use the REPT and ENDM directives.

Explanation

o If a name or label is to be described as the operand of the REPT directive, the name or label must be an absolute term which has already been defined in the source module.
o In the REPT-ENDM block, none of the MACRO, REPT, IRP, and ENDM directives can be described. In other words, nesting of macros is not allowed within the REPT-ENDM block.
o If the EXITM directive is described in the REPT-ENDM block, subsequent expansion of the REPT-ENDM block by the assembler will be terminated. (See (5), EXITM directive.)

Application Examples

Example 1

<Source program>

```
        NAME    SAMP1

        CSEG
          :
        REPT    3       ;(1)
        INC     B                  REPT-ENDM block
        DEC     C
        ENDM            (2)
          :
        END
```

(1) This directive instructs the assembler to expand the REPT-ENDM block three consecutive times.
(2) This directive indicates the end of the REPT-ENDM block.

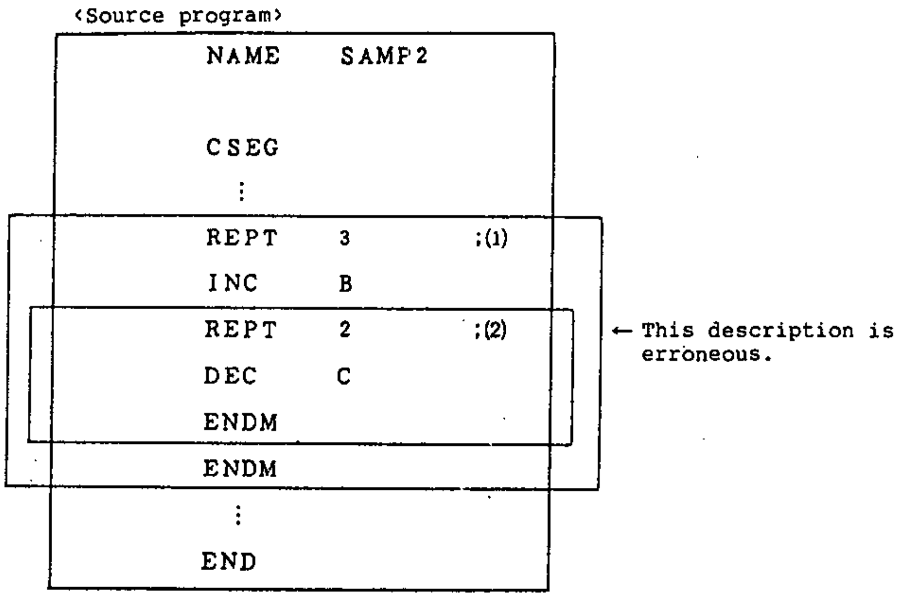
When the above source program is assembled, the REPT-ENDM block is expanded as shown in the following assembly list:

<Assembly list>

```
        NAME    SAMP1

        CSEG
          :
        INC     B
        DEC     C
        INC     B
        DEC     C
        INC     B
        DEC     C
          :
        END
```

You can see that the REPT-ENDM block defined by statements (1) and (2) has been expanded three times. On the assembly list, the definition statements (1) and (2) by the REPT directive in the source module will not be displayed.

Example 2



```
<Source program>
        NAME    SAMP2

        CSEG
          :
        REPT    3        ;(1)
        INC     B
        REPT    2        ;(2)    ← This description is
        DEC     C                  erroneous.
        ENDM
        ENDM
          :
        END
```

(1) This directive instructs the assembler to expand the REPT-ENDM block three consecutive times.

(2) This directive instructs the expansion of the REPT-ENDM block again within the REPT-ENDM block. This description is incorrect, because nesting of macros is not allowed within the REPT-ENDM block.

(4) IRP (indefinite repeat)

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
[label:]	IRP	formal parameter, <actual parameter[,...]>	[;comment]
	⟆		
	ENDM		

Function

The IRP directive causes the assembler to repeatedly expand a series of statements described between this directive and the ENDM directive (called the IRP-ENDM block) the number of times equivalent to the number of actual parameters while replacing the formal parameter with the actual parameters specified in the Operand field (in sequence from left to right).

Use

If a series of statements, only part of which becomes variables is to be described repeatedly in a source program, use the IRP and ENDM directives.

Explanation

- o Up to 13 actual parameters may be described in the Operand field.
- o A numeric value, name, or label can be described as an actual parameter. However, no expression can be described.
- o In the IRP-ENDM block, none of the MACRO, REPT, IRP, and ENDM directives can be described. In other words, nesting of macros is not allowed within the IRP-ENDM block.
  If the EXITM directive is described within the IRP-ENDM block, the expansion of the IRP-ENDM block subsequent to the EXITM directive will be terminated.

NOTE 3-4

If a symbol which has the same name as the formal parameter of the IRP directive is described as a name or label within the same IRP-ENDM block, no error will occur. However, its object code will become invalid. Therefore, take care not to define any symbol having the same name as the formal parameter.

Application Example

‹Source program›

```
        NAME    SAMP1

        CSEG
          :
        IRP     PARA,<0AH,0BH,0CH>  ;(1)
        ADD     A,#PARA                    IRP-ENDM block
        MOV     [DE+],A
        ENDM                        ;(2)
          :
        END
```

(1) The formal parameter is "PARA" and the actual parameters are the following three: "0AH", "0BH", and "0CH".
This directive instructs the assembler to expand the IRP-ENDM block three times (i.e., the number of actual parameters) while replacing the formal parameter "PARA" with the actual parameters "0AH", "0BH" and "0CH".

(2) This directive indicates the end of the IRP-ENDM block.

When the above source program is assembled, the IRP-ENDM block is expanded as shown in the following assembly list:

<Assembly list>

```
    NAME    SAMP1

    CSEG
     :
    ADD     A, #PARA    ; (3)
    MOV     [DE+], A
    ADD     A, #PARA    ; (4)
    MOV     [DE+], A
    ADD     A, #PARA    ; (5)
    MOV     [DE+], A
     :
    END
```

You can see that the IRP-ENDM block defined by statements (1) and (2) has been expanded three times (equivalent to the number of actual parameters).
On the assembly list in this case, the formal parameters within the IRP-ENDM block are not replaced with the actual parameters. However, in object code generation, the formal parameters are replaced with the actual parameters in sequence from left to right.

```
                         (Object code)
(3) ADD A, #PARA    -->  A80A
(4) ADD A, #PARA    -->  A80B
(5) ADD A, #PARA    -->  A80C
```

(5) EXITM (exit from macro)

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
[label:]	EXIT	None	[;comment]

Function

- o The EXITM directive terminates by force the expansion of the macro body defined by the MACRO directive.
- o If the EXITM directive is described in the REPT-ENDM or IRP-ENDM block, the directive terminates by force the definition (expansion) of the block. (The REPT or IRP directive becomes invalid.)

Use

- o This function is mainly used when a conditional assembly function (see Section 4.4, Conditional Assembly Control Instructions) is used in the macro body defined with the MACRO directive.
- o If conditional assembly functions are used in combination within the macro body, part of the source program which must not be assembled is likely to be assembled unless control is returned from the macro by force with the EXITM directive. In such a case, the EXITM directive must be used.

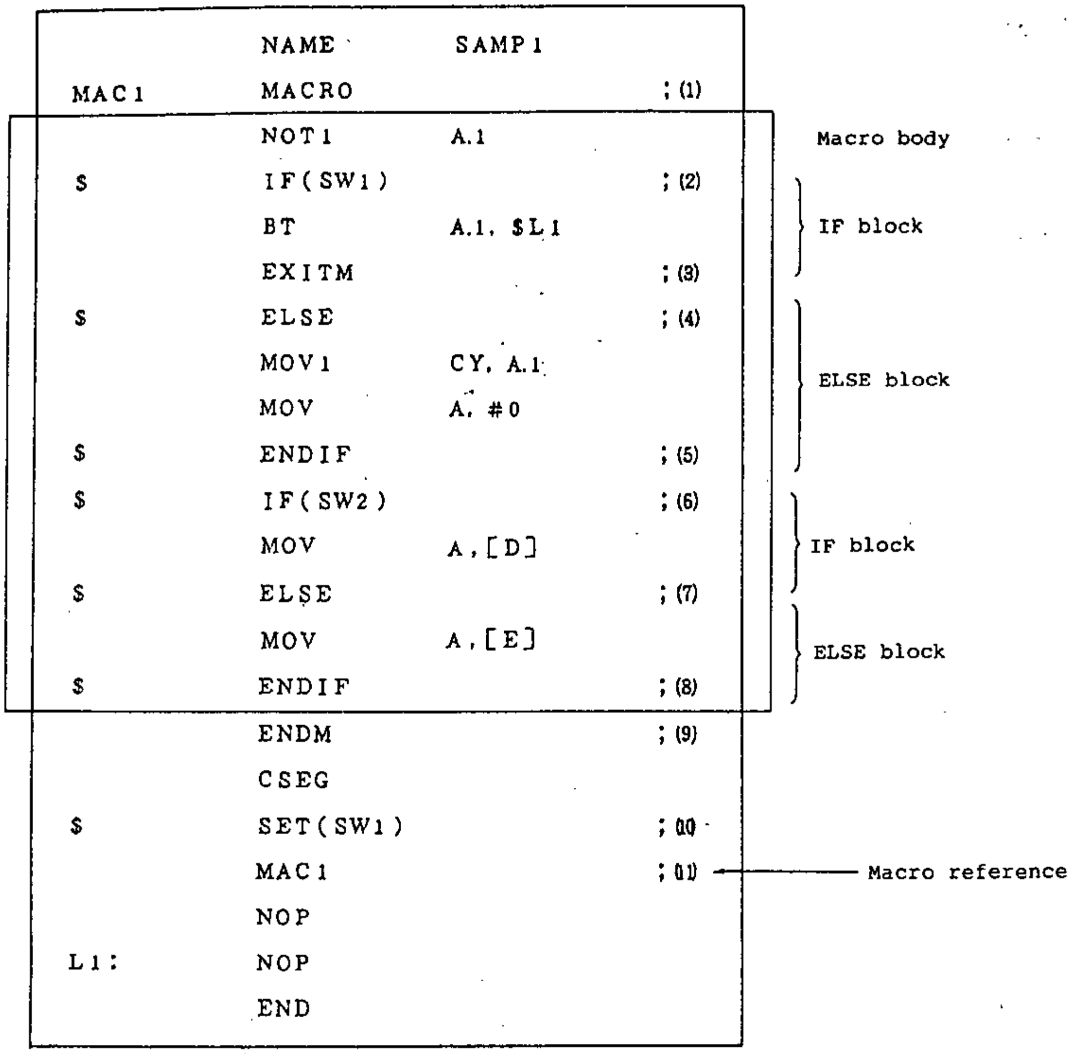
Explanation

If the EXITM directive is described in a macro body, instructions up to the ENDM directive are stored as the macro body.
The EXITM directive indicates the end of a macro only during the macroexpansion.

Application Example

- o In the example here, conditional assembly control instructions are used. See Section 4.4, Chapter 4 for the conditional assembly control instructions.
- o See Chapter 5, Macros for the macro body and macro-expansion.

<Source program>



```
            NAME        SAMP1
MAC1        MACRO                   ; (1)
            NOT1        A.1
$           IF(SW1)                 ; (2)
            BT          A.1, $L1
            EXITM                   ; (3)
$           ELSE                    ; (4)
            MOV1        CY, A.1
            MOV         A, #0
$           ENDIF                   ; (5)
$           IF(SW2)                 ; (6)
            MOV         A,[D]
$           ELSE                    ; (7)
            MOV         A,[E]
$           ENDIF                   ; (8)
            ENDM                    ; (9)
            CSEG
$           SET(SW1)                ; (10)
            MAC1                    ; (11)
            NOP
L1:         NOP
            END
```

(1) The macro "MAC1" uses conditional assembly functions (2) and (4) through (8) within the macro body.

(2) This instruction defines an IF block for conditional assembly. If switch name "SW1" is true (0FFH), the IF block will be assembled.

(3) This directive terminates by force the expansion of the macro body in (4) and thereafter.
If this EXITM directive is omitted, the assembler proceeds to the assembly process in (6) and thereafter when the macro is expanded.

(4) This instruction defines an ELSE block for conditional assembly. If switch name "SW1" is false (00H), the ELSE block will be assembled.

(5) This instruction indicates the end of the conditional assembly.

(6) This instruction defines another IF block for conditional assembly. If switch name "SW2" is true (0FFH), the IF block following this will be assembled.

(7) This instruction defines another ELSE block for conditional assembly. If switch name "SW2" is false (00H), the ELSE block will be assembled.

(8) This instruction indicates the end of the conditional assembly processes in (6) and (7).

(9) This directive indicates the end of the macro body.

(10) This SET control instruction gives true value (0FFH) to switch name "SW1" and sets the condition of the conditional assembly.

(11) This instruction references macro "MAC1".

When the source program in the above example is assembled, macroexpansion occurs as shown below.

```
         NAME        SAMP1
MAC1     MACRO                       ;(1)
          (
          )
         ENDM                        ;(9)
         CSEG
$        SET(SW1)                    ;(10)
         MAC1                        ;(11)
$        IF(SW1)                          Macro-expanded part
         BT          A.1, $L1
         NOP
L1:      NOP
         END
```

By the macro reference in (11), the macro body of macro "MAC1" has been expanded. Because true value (0FFH) is set in switch name "SW1" in (10), the first IF block in the macro body is assembled. Because the EXITM directive is described at the end of the IF block, the subsequent macroexpansion is not executed.

(6) ENDM (end macro)

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
None	ENDM	None	[;comment]

Function

The ENDM directive instructs the assembler to terminate the execution of a series of statements defined as the functions of the macro.

Use

The ENDM directive must always be described at the end of a series of statements following the MACRO, REPT, or IRP directive.

Explanation

o A series of statements described between the MACRO directive and ENDM directive becomes a macro body.
o A series of statements described between the REPT directive and ENDM directive becomes an REPT-ENDM block.
o A series of statements described between the IRP directive and ENDM directive becomes an IRP-ENDM block.

Application Examples

Example 1

<MACRO—ENDM>

```
        NAME    SAMP1
ADMAC   MACRO   PARA1,PARA2
        MOV     A,#PARA1
        ADD     A,#PARA2
        ENDM
          :
        END
```

Example 2

<REPT—ENDM>

```
        NAME    SAMP2
        CSEG
          :
        REPT    3
        INC     B
        DEC     C
        ENDM
          :
        END
```

Example 3

<IRP—ENDM>

```
        NAME    SAMP3
        CSEG
          :
        IRP     PARA,<1,2,3>
        ADD     A,#PARA
        MOV     [E+],A
        ENDM
          :
        END
```

3.8 Directive for Assembly Termination

The assembly termination directive (END) informs the assembler of the end of a source module. This assembly termination directive must always be described at the end of each source module.
The assembler processes a series of statements up to the assembly termination directive as a source module. Therefore, if the END directive is described before the ENDM directive in the REPT-ENDM or IRP-ENDM block, the REPT or IRP block becomes invalid.

(1) END (end)

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
None	END	None	[;comment]

Function

The END directive indicates to the assembler the end of a source module.

Use

The END directive must always be described at the end of each source module.

Explanation

- o The assembler continues to assemble a source module until the END directive appears in the source module. Therefore, the END directive is required at the end of each source module.
- o Always input a carriage return (CR) or line feed code (LF) code immediately after the END directive.

Application Example

```
NAME   SAMPLE
DSEG
  :
CSEG
  :
END              ;(1)
```

(1) Always describe the END directive at the end of each source module.

CHAPTER 4. CONTROL INSTRUCTIONS

4.1 Overview of Control Instructions

Control instructions are described in a source program and are used to provide particular instructions on the assembler operation. These instructions are not subject to object code generation.

Control instructions are available in the following three types:

Table 4-1 List of Control Instructions

No.	Type of control instruction	Control instruction
1	Instruction to control INCLUDE file	INCLUDE
2	Instructions to control assembly list	EJECT, NOLIST, LIST, SUBTITLE

3	Instructions to control conditional assembly	SET, RESET
		IF, ELSEIF, ELSE, ENDIF

Control instructions are described in a source program just the same as directives. However, "$" (dollar sign) must be described in the 1st column of the source statement.

```
$ [one blank or TAB] Control instruction
↑
|
1st column
```

Of the control instructions listed in Table 4-1, the following instructions can also be specified as assembler options in the start-up command line of the assembler:

- o NOLIST, LIST, SUBTITLE
- o SET, RESET

For the method of specifying assembler options in the command line, see Subsection 4.3.1, "Starting up the assembler", Chapter 4 in the RA78K/I Assembler Package User's Manual for Operation.

4.2 INCLUDE control instruction

The INCLUDE control instruction is used to include another module file in a source module.
By making the most of this control instruction, you may save your time and labor in describing a source program.
This control instruction is detailed on the next page.

(1) INCLUDE (include)

Description Format

```
$[△]INCLUDE (filename)
$[△]IC (filename)                 ; Abbreviated format
↑
1st column
```

Function

The INCLUDE control instruction inserts the contents of the file specified by "filename" into the source program for assembly.

Use

A relative large group of statements which may be shared by two or more source modules should be combined into a single file as an INCLUDE file. If the group of statements must be used in each source module, specify the filename of the required INCLUDE file with the INCLUDE control instruction. With this instruction, your time and effort in describing the source modules can be greatly reduced.

Explanation

o When describing the INCLUDE control instruction, "$" must be described in the 1st column of the source statement.
  If you wish to provide a space between "$" and "INCLUDE", input only one BLANK or TAB character.
o When specifying a filename, the drive name or directory name in which the INCLUDE file is stored can be specified.
  If neither drive name nor directory name is specified, the drive or directory in which the source module file is stored is assumed to have been specified.
o Nesting of INCLUDE files is allowed only for one level.
  (The term "nesting" refers to the specification of one or more other INCLUDE files in an INCLUDE file.)

o The END directive need not be described in an INCLUDE file.

Application Example

<Source program>

```
        NAME     SAMPLE
        EXTRN    L1, L2
        PUBLIC   L3
$       INCLUDE(EQU.INC)  ;(1)
        CSEG
          :
        END
```

<INCLUDE file EQU.INC>

```
SYMA    EQU     10H
SYMB    EQU     20H
                 :
SYMZ    EQU    100H
```

(1) This control instruction specifies "EQU.INC" as the INCLUDE file. When this source program is assembled, the contents of the INCLUDE file will be expanded as follows:

```
        NAME     SAMPLE
        EXTRN    L1, L2
        PUBLIC   L3
$       INCLUDE(EQU.INC)  ;(1)
SYMA    EQU      10H
SYMB    EQU      20H
          :
SYMZ    EQU     100H

        CSEG
          :
        END
```

The contents of INCLUDE file "EQU.INC" have been expanded.

4.3 Assembly List Control Instructions

These control instructions are used to control the output format of an assembly list such as page ejection, suppression of list output, and subtitle output.

Assembly list control instructions include EJECT, NOLIST, LIST, and SUBTITLE. Each of these control instructions is explained on the following pages.

(1) EJECT (eject)

Description Format

```
$[△]EJECT
$[△]EJ                          ; Abbreviated format
↑
1st column
```

Function

The EJECT control instruction causes the assembler to execute page ejection of an assembly list.

Use

Describe the EJECT control instruction in a line of the source module at which the page ejection of the assembly list is required.

Explanation

- o When describing the EJECT control instruction, "$" must be described in the 1st column of the source statement. If you wish to provide a space between "$" and "EJECT", input only one BLANK or TAB character.
- o The image (i.e., $ EJECT) of the EJECT control instruction itself will be printed at the top of the page after the page ejection.
- o If any of the following constructions or assembler options has already been specified, the EJECT control instruction will become invalid.

  NOLIST, NOPRINT, NOPAGING

  (For the NOPRINT and NOPAGING options, see Subsection 4.4.4, "Description of each assembler option", in Chapter 4 of the RA78K/I Assembler Package User's Manual for Operation.)

Application Example

<Source module>

```
          :
        MOV        [DE+], A
        BR         $$
$       EJECT                    ; (1)
        CSEG
          :
        END
```

(1) When page ejection is executed with the EJECT control instruction, the assembly list will look like this.

```
          :
        MOV     [E+],A
        BR      $$
--------------------------------- ← Page ejection
$       EJECT
        CSEG
          :
        END
```

(2) NOLIST (no list)

Description Format

```
$[△]NOLIST
$[△]NOLI                              ; Abbreviated format
 ↑
1st column
```

Function

The NOLIST control instruction indicates to the assembler the line at which assembly list output must be suppressed. All source statements described after the NOLIST control instruction specification until the LIST control instruction appears in the source program will be assembled but will not be output on the assembly list.

Use

Use the NOLIST control instruction to merely control the amount of list output.

Explanation

o When describing the NOLIST control instruction, "$" must be described in the 1st column of the source statement. If you wish to provide a space between "$" and "NOLIST", input only one BLANK or TAB character.

o The NOLIST control instruction functions to suppress assembly list output and is not intended to stop the assembly process.

o If the LIST control instruction is specified after the NOLIST control instruction, statements described after the LIST control instruction will be output again on the assembly list.

o The NOLIST control instruction can also be described as an assembler option in the start-up command line of the assembler.

Application Example

```
         NAME     SAMP1
$        NOLIST            ;(1)
DATA1    EQU      10H
DATA2    EQU      11H
           :
DATAX    EQU      20H
$        LIST              ;(2)
         CSEG
           :
         END
```

Statements in this part will not be output on the assembly list.

(1) Because the NOLIST control instruction is specified here, statements after "$ NOLIST" and up to the LIST control instruction in (2) will not be output on the assembly list. The image of the NOLIST control instruction itself will not be output on the list too.

(2) Because the LIST control instruction is specified here, statements after this control instruction will be output again on the assembly list. The image of the LIST control instruction will also be output on the list.

(3) LIST (list)

Description Format

```
$[△]LIST
$[△]LI                              ; Abbreviated format
 ↑
 1st column
```

Function

The LIST control instruction indicates to the assembler the line at which assembly list output must be started.

Use

Use the LIST control instruction to release the suppression of the assembly list output specified by the NOLIST control instruction and to output the assembly list again.
By using the LIST control instruction in combination with the NOLIST control instruction in a source program, the output amount or contents of the assembly list can be controlled.

Explanation

o When describing the LIST control instruction, "$" must be described in the 1st column of the source statement.
If you wish to provide a space between "$" and "LIST", input only one BLANK or TAB character.

o If the LIST control instruction is specified after the NOLIST control instruction, statements described after the LIST control instruction will be output again on the assembly list. The image of the LIST control instruction itself will also be printed on the assembly list.

o The LIST control instruction can also be described as an assembler option in the start-up command line of the assembler.

Application Example

See the Application Example of the NOLIST control instruction.

(4) SUBTITLE (subtitle)

Description Format

```
$[△]SUBTITLE('character string')
$[△]ST('character string')          ; Abbreviated format
 ↑
 |
1st column
```

Function

The SUBTITLE control instruction specifies the character string to be printed on the SUBTITLE section at each page header of an assembly list.

Use

Use the SUBTITLE control instruction to print a subtitle on each page of an assembly so that the contents of the assembly list can be readily identified. The character string of a subtitle may be changed for each page.

Explanation

- o When describing the SUBTITLE control instruction, "$" must be described in the 1st column of the source statement. If you wish to provide a space between "$" and "SUBTITLE", input only one BLANK or TAB character.
- o Up to 60 characters can be specified as the character string of a subtitle.
- o Description of a character string in excess of 61 characters as a subtitle will result in an error. However, the first 60 characters of the character string will be accepted as valid.
- o The character string specified with the SUBTITLE control instruction will be printed in the SUBTITLE section on the 2nd page of the assembly list.

- o If the SUBTITLE control instruction is omitted, the SUBTITLE section on each page will be left blank.
- o This control instruction can also be specified as an assembler option in the start-up command line of the assembler.

Note

When the SUBTITLE control instruction is to be specified as an assembler option in the start-up command line of the assembler, pay attention to the following points:

- o The character string to be specified as a subtitle need not be enclosed in a pair of single quotation marks. With the MS-DOS or PC-DOS based system, an error will result if the character string is enclosed in a pair of single quotation marks.
- o No Blank characters can be used as the character string.
- o If more than 60 characters are specified as the character string, an error will result and the program execution will be aborted.

Application Example

<Source module>

```
        NAME    SAMP

        CSEG


$       SUBTITLE('THIS IS SUBTITLE 1')  ;(1)
$       EJECT                           ;(2)


        CSEG


$       SUBTITLE('THIS IS SUBTITLE 2')  ;(3)
$       EJECT                           ;(4)


        CSEG


        END
```

(1) This control instruction specifies character string "THIS IS SUBTITLE 1".

(2) This control instruction indicates page ejection.

(3) This control instruction specifies character string "THIS IS SUBTITLE 2".

(4) This control instruction indicates page ejection.

When the source program in the above example is assembled, the assembly list will look like this.

```
Copyright (C) 1986 NEC Corporation
UCOM-78K/I ASSEMBLER VX.X                                        DATE        PAGE    1

SOURCE FILE: TEXT.ASM
OBJECT FILE: TEXT.REL
COMMAND     : RA78K1 TEXT.ASM PROCESSOR(112)


    ASSEMBLE LIST

 STNO  ADRS R OBJECT     M I  SOURCE STATEMENT

    1                                NAME    SAMP

    2                                CSEG

                              $      SUBTITLE('THIS IS SUBTITLE 1')  ;(1)




-------------------------------------------------------------------------------------
UCOM-78K1 ASSEMBLER VX.X                                         DATE        PAGE    2
THIS IS SUBTITLE 1

                              $      EJECT                           ;(2)

    3                                CSEG

                              $      SUBTITLE('THIS IS SUBTITLE 2')  ;(3)


-------------------------------------------------------------------------------------
UCOM-78K1 ASSEMBLER VX.X                                         DATE        PAGE    3
THIS IS SUBTITLE 2

                              $      EJECT                           ;(4)

    4                                CSEG

    5                                END
```

← Page ejection by instruction in (2)

← Subtitle printing by instruction in (1)

← Page ejection by instruction in (4)

← Subtitle printing by instruction in (3)

4.4 Control Instructions for Conditional Assembly

Conditional assembly control instructions select a series of statements in a source module as those subject to assembly or not subject to assembly by setting switches for conditional assembly.
Conditional assembly control instructions are available in two groups: one group to set the condition for limiting source statements subject to assembly (IF, ELSEIF, ELSE, and ENDIF) and the other, to give a true or false value to a specified switch name (SET and RESET).
Each group of these control instructions is detailed on the following pages.
By making the best of these control instructions, assembly of a source module by excluding unwanted statements can be executed with little or no change to the source module.

(1) IF, ELSEIF, ELSE, ENDIF

Description Format

```
$[△]IF(switch name[:switch name[...]])
$[△]ELSEIF(switch name[:switch name[...]])
$[△]ELSE
$[△]ENDIF
↑
|
1st column
```

Function

- o These control instructions set the conditions to limit source statements subject to conditional assembly and those not subject to conditional assembly.
  Source statements described between the IF control instruction and the ENDIF control instruction are subject to conditional assembly.
- o If the value of the switch name specified by the IF control instruction (i.e., IF condition) is true (0FFH), source statements described after this IF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF, ELSE, or ENDIF) in the source program will be assembled. For subsequent assembly processing, the assembler will proceed to the statement next to the ENDIF control instruction.
  If the IF condition is false (00H), source statements described after this IF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF, ELSE, or ENDIF) in the source program will be not assembled.
- o The ELSEIF control instruction is checked for true/false value of its switch name(s) only when the conditions of the IF control instruction described before this ELSEIF control instruction are not satisfied (i.e., all the switch name values are false).

If the value of the switch name specified by the ELSEIF control instruction (i.e., ELSEIF condition) is true (0FFH), source statements described after this ELSEIF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF, ELSE, or ENDIF) in the source program will be assembled. For subsequent assembly processing, the assembler will proceed to the statement next to the ENDIF control instruction.

If the ELSEIF condition is false (00H), source statements described after this ELSEIF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF, ELSE, or ENDIF) in the source program will be not assembled.

o If all the conditions of the IF and ELSEIF control instructions described before the ELSE control instruction are not satisfied (i.e., the switch name values are all false), source statements described after this ELSE control instruction until the appearance of the ENDIF control instruction in the source program will be assembled.

o The ENDIF control instruction indicates to the assembler the termination of source statements subject to conditional assembly.

<u>Use</u>

o With these conditional assembly control instructions, source statements subject to assembly can be changed without major modifications to the source program.

o If a statement for debugging necessary only during the program development is described in a source program, whether or not the debugging statement should be assembled (translated into machine language) can be specified by setting switches for conditional assembly.

<u>Explanation</u>

o When describing any of the IF, ELSEIF, ELSE, and ENDIF control instructions, "$" must be described in the 1st column of the source statement.

If you wish to provide a space between "$" and each of these control instructions, input only one BLANK or TAB character.

o With the IF and ELSEIF control instructions, at least one switch name must be described.
  The rules of describing switch names are the same as the conventions of symbol description, for which see Subsection 2.2.3, "Symbol field" in Chapter 2. However, note that underscore (_) cannot be used to describe any switch name.
  Up to five switch names can be used per module.

o If two or more switch names are to be specified with the IF or ELSEIF control instruction, delimit each switch name with a colon (:).

o When two or more switch names have been specified with the IF or ELSEIF control instruction, the IF or ELSEIF condition is judged as satisfied if one of the switch name values is true (0FFH).

o The value of each switch name to be specified with the IF or ELSEIF control instruction must be defined with the SET or RESET control instruction. (See (2), "SET, RESET" in this section.) Therefore, the value of the switch name specified with the IF or ELSEIF control instruction must have been set in the source module with the SET or RESET control instruction.

o Within an IF-ENDIF block, another IF-ENDIF block cannot be described. (Nesting of IF statements is not allowed.)

o In conditional assembly, object codes will not be generated for statements not assembled, but these statements will be output as is on the assembly list.

Application Example

Example 1

```
            text0

$           IF(SW1)        ;(1)

            text1

$           ENDIF          ;(2)
              ⋮
            END
```

(1) If the value of switch name "SW1" is true (0FFH), statements in "text1" will be assembled.
If the value of switch name "SW1" is false (00H), statements in "text1" will not be assembled.
The value of switch name "SW1" has been set to true (0FFH) or false (00H) with the SET or RESET control instruction described in "text0".

(2) This instruction indicates the end of the source statement range for conditional assembly.

Example 2

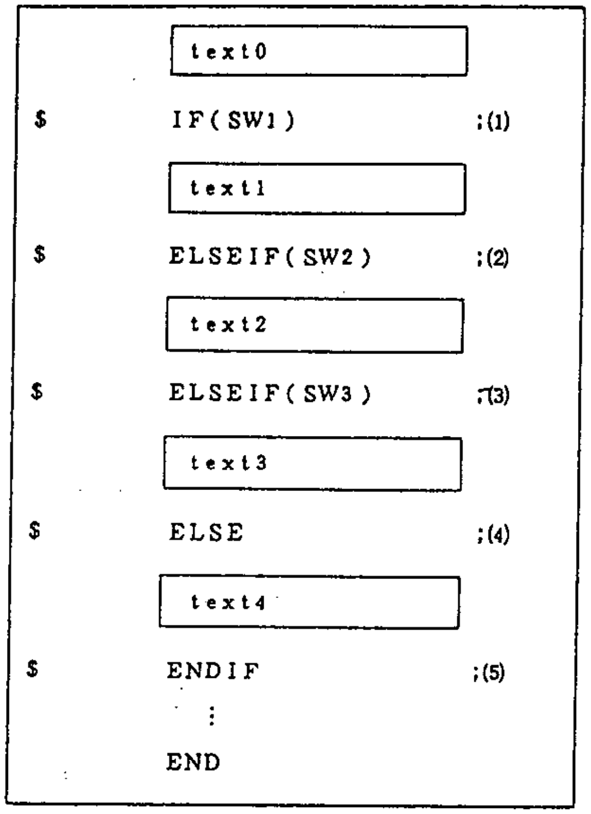
```
            text0

$           IF(SW1)        ;(1)

            text1

$           ELSE           ;(2)

            text2

$           ENDIF          ;(3)
              ⋮
            END
```

(1) The value of switch name "SW1" has been set to true (0FFH) or false (00H) with the SET or RESET control instruction described in "text0".
If the value of switch name "SW1" is true (0FFH), statements in "text1" will be assembled and statements in "text2" will not be assembled.

(2) If the value of switch name "SW1" is false (00H), statements in "text1" will not be assembled and statements in "text2" will be assembled.

(3) This instruction indicates the end of the source statement range for conditional assembly.

Example 3



```
            text0
$           IF(SW1)          ;(1)
            text1
$           ELSEIF(SW2)      ;(2)
            text2
$           ELSEIF(SW3)      ;(3)
            text3
$           ELSE             ;(4)
            text4
$           ENDIF            ;(5)
              :
            END
```

(1) The values of switch names "SW1", "SW2", and "SW3" have been set to true (0FFH) or false "00H" with the SET or RESET control instruction described in "text0".
If the value of switch name "SW1" is true, statements in "text1" will be assembled and statements in "text2", "text3", and "text4" will not be assembled.
If the value of switch name "SW1" is false, statements in "text1" will not be assembled and conditional assembly of statements in "text2" and thereafter will be executed.

(2) If the value of switch name "SW1" in (1) is false and the value of switch name "SW2" is true, statements in "text2" will be assembled and statements in "text1", "text3", and "text4" will not be assembled.

(3) If the values of both switch names "SW1" in (1) and "SW2" in (2) are false and the value of switch name "SW3" is true, statements in "text3" will be assembled and statements in "text1", "text2" and "text4" will not be assembled.

(4) If the values of switch names "SW1" in (1), "SW2" in (2), and "SW3" in (3) are all false, statements in "text4" will be assembled and statements in "text1", "text2" and "text3" will not be assembled.

(5) This instruction indicates the end of the source statement range for conditional assembly.

Example 4

```
        ┌──────────────────┐
        │ text0            │
        └──────────────────┘
$       IF(SWA:SWB)          ;(1)
        ┌──────────────────┐
        │ text1            │
        └──────────────────┘
$       ENDIF                ;(2)
           :
        END
```

(1) The values of switch names "SWA" and "SWB" have been set to true (0FFH) or false "00H" with the SET or RESET control instruction described in "text0".
If the value of switch name "SWA" or "SWB" is true (0FFH), statements in "text1" will be assembled.
If the values of both switch names "SWA" and "SWB" are false, statements in "text1" will not be assembled.

(2) This instruction indicates the end of the source statement range for conditional assembly.

(2) SET, RESET (set, reset)

Description Format

```
$[△]SET(switch name[:switch name[...]])
$[△]RESET(switch name[:switch name[...]])
↑
|
1st column
```

Function

- o The SET and RESET control instructions give a value (true or false) to each switch name to be specified with the IF or ELSE control instruction.
- o The SET control instruction gives a true value (0FFH) to the switch name specified in the operand.
- o The RESET control instruction gives a false value (00H) to the switch name(s) specified in the operand.

Use

- o Describe the SET control instruction to give a true value (0FFH) to a switch name to be specified with the IF or ELSEIF control instruction.
- o Describe the RESET control instruction to give a false value (00H) to a switch name to be specified with the IF or ELSEIF control instruction.

Explanation

- o When describing the SET and RESET control instructions, "$" must be described in the 1st column of the source statement. If you wish to provide a space between "$" and each of these control instructions, input only one BLANK or TAB character.
- o With the SET and RESET control instructions, at least one switch name must be described.

The rules of describing switch names are the same as the conventions of symbol description, for which see Subsection 2.2.3, "Symbol field" in Chapter 2. However, note that underscore (_) cannot be used to describe any switch name. Up to five switch names can be used per module.

o If two or more switch names are to be specified with the SET or RESET control instruction, delimit each switch name with a colon (:).

o The switch name once set to "true" with the SET control instruction can be changed to "false" with the RESET control instruction, and vice versa.

o A switch name to be specified with the IF or ELSEIF control instruction must be defined at least once with the SET or RESET control instruction in the source module before describing the IF or ELSEIF control instruction.

o The SET and RESET control instructions can also be specified as assembler options in the start-up command line of the assembler.

Application Example

Example 1

```
              :
$         SET(SW1)              ;(1)
              :
$         IF(SW1)                (2)
          text1
$         ENDIF                 ;(3)
              :
$         RESET(SW1:SW2)        ;(4)
              :
$         IF(SW1)               ;(5)
          text2
$         ELSEIF(SW2)           ;(6)
          text3
$         ELSE                  ;(7)
          text4
$         ENDIF                 ;(8)
              :
          END
```

(1) This instruction gives a true value (0FFH) to switch name "SW1".

(2) Because the true value has been given to switch name "SW1" in (1) above, statements in "text1" will be assembled.

(3) This instruction indicates the end of the source statement range for conditional assembly, which starts from (2).

(4) This instruction gives a false value (00H) to switch names "SW1" and "SW2".

(5) Because the false value has been given to switch name "SW1" in (4) above, statements in "text2" will not be assembled.

(6) Because the false value has also been given to switch name "SW2" in (4) above, statements in "text3" will not be assembled.

(7) Because both switch names "SW1" and "SW2" are false in (5) and (6) above, statements in "text4" will be assembled.

(8) This instruction indicates the end of the source statement range for conditional assembly, which starts from (5).

CHAPTER 5. MACROS

5.1 Overview of Macro

When you must describe a series of instruction groups over and over again in a source program, use of a macro function is very useful for program description.

The macro function refers to the expansion of a series of instruction groups defined as a macro body with MACRO and ENDM directives into the location where the macro name is referenced.

A macro is used to increase coding efficiency of a source program and is different from a subroutine.

A macro and a subroutine each have the following features and should be used selectively according to the specific purpose.

(1) Subroutine

- o Describe a process (or the same sequence of instructions) which has to be repeated over and over again in a program as a subroutine. The subroutine will be converted into machine language just once by the assembler.
- o To call the subroutine, you only need to describe a subroutine call instruction. (Generally, instructions to set arguments are also described before and after the subroutine.)
  Therefore, by making the best of subroutines, the program memory can be used with high efficiency.
- o By coding a series of processes in a program as subroutines, the program can be structurized. (By this structurization, the programmer can easily understand the overall structure of the program, thus making the program design easy.)

(2) Macro

- o The basic function of a macro is the replacement of a group of instructions with a name.
  A series of instruction groups defined as a macro body with MACRO and ENDM directives will be expanded into the location where the macro name is referenced.
- o When the assembler detects a macro reference, the assembler expands the macro body and converts the group of instructions into machine language while replacing the formal parameter(s) of the macro body with the actual parameters at the time of the macro reference.

o Parameters can be described for a macro.
  For example, if there are instruction groups which are the same in processing procedure but are different in data to be described in the operand, define a macro by assigning formal parameter(s) to the data. By describing the macro name and the actual parameter(s) at macro reference time, the assembler can cope with various instruction groups which differ only in part of the statement description.

The programming technique with subroutines is mainly used for memory size reduction and program structurization, whereas macros are used to increase coding efficiency of the program.

5.2 Utilization of Macros

5.2.1 Macrodefinition

A macro is defined with the MACRO and ENDM directives.

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
macro name	MACRO	[formal parameter[,...]]	[;comment]
	∫		
	ENDM		

Function

The MACRO directive executes a macrodefinition by assigning the macro name specified in the Symbol field to a series of statements (called a macro body) described between this directive and the ENDM directive.

Application Example

```
ADMAC   MACRO   PARA1,PARA2
        MOV     A,#PARA1
        ADD     A,#PARA2
        ENDM
```

The above example shows a simple macrodefinition which specifies the addition of two values PARA1 and PARA2 and the storage of the result in register A. The macro is given a name "ADMAC" and "PARA1" and "PARA2" are formal parameters. For details, see (1) MACRO in Section 3.7, Macro Directives, Chapter 3.

5.2.2 Macro reference

To call a macro, the already defined macro name must be described in the Mnemonic field of the source program.

Description Format

Symbol field	Mnemonic field	Operand field	Comment field
[label:]	macro name	[actual parameter[,...]]	[;comment]

Function

This directive calls the macro body assigned to the macro name specified in the Mnemonic field.

Use

Use this directive description to call a macro body.

Explanation

- o The macro name to be specified in the Mnemonic field must have been defined before the macro reference.
- o A constant, expression, name, or label can be described as the operand of this directive.
- o Formal parameters are replaced with their corresponding actual parameters in sequence from left to right.
- o An error will result if the number of formal parameters is not equal to the number of actual parameters.

Application Example

```
        NAME     SAMPLE
ADMAC   MACRO    PARA1,PARA2

        MOV      A,#PARA1
        ADD      A,#PARA2

        ENDM

        CSEG
         :
        ADMAC    10H,20H
         :
        END
```

This directive calls the already defined macro name "ADMAC". 10H and 20H are actual parameters.

5.2.3 Macroexpansion

The assembler processes a macro as follows:

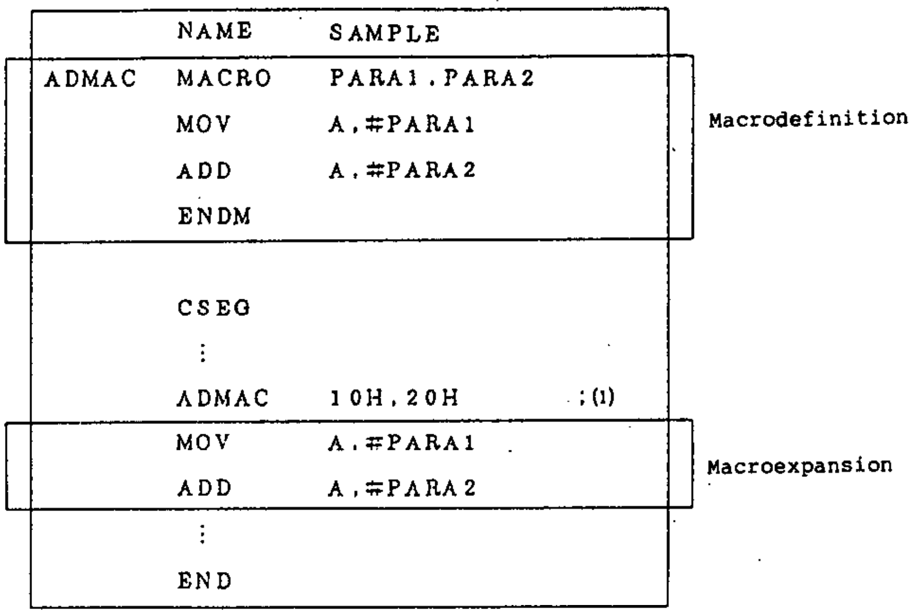
o Stores the macro body of the defined macro name in the symbol table area (an area within the assembler).

o Searches the macro body corresponding to the referenced macro name from the symbol table area and expands it to the location where the macro name is referenced.

o Assembles statements in the expanded macro body just the same as other statements.

Application Example

When the macro referenced in Subsection 5.2.2, "Macro reference" is assembled, the macro body will be expanded as shown below.



```
        NAME     SAMPLE
ADMAC   MACRO    PARA1.PARA2
        MOV      A,#PARA1
        ADD      A,#PARA2
        ENDM

        CSEG
         :
        ADMAC    10H,20H        ;(1)
        MOV      A,#PARA1
        ADD      A,#PARA2
         :
        END
```

By the macro reference in (1), the macro body will be expanded. On the assembly list in this case, the formal parameters within the macro body are not replaced with the actual parameters. However, in object code generation, the formal parameters will be replaced with the actual parameters as shown below.

(Object code)

MOV A, #PARA1 $\rightarrow$ B9$\underline{10}$

ADD A, #PARA2 $\rightarrow$ A8$\underline{20}$

5.3 Symbols within Macro

Symbols that can be defined in a macro are divided into two types: global symbols and local symbols.

(1) Global symbols

- o A global symbol is a symbol that can be referenced from any statement within a source program.
  Therefore, if a series of statements are expanded by referencing a macro in which the global symbol has been defined, the symbol will cause a double definition error.
- o Symbols not defined with the LOCAL directive are global.

(2) Local symbols

- o A local symbol is a symbol defined with the LOCAL directive. (See (2) LOCAL in Section 3.7, "Macro directives".)
- o A local symbol is valid within the macro in which the local symbol has been defined.
  Therefore, if a series of statements are expanded by referencing a macro in which the local symbol has been defined, the symbol will not cause a double definition error.
- o No local symbol can be referenced from outside the macro.

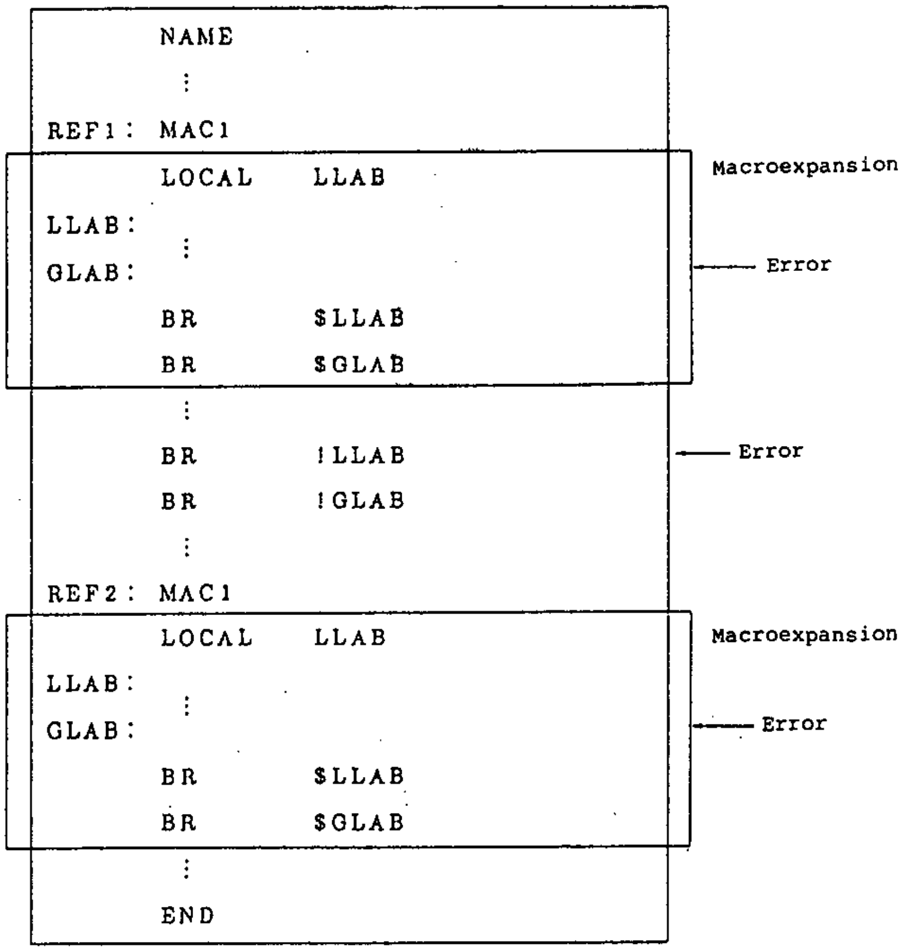
Application Example

<Source program>

```
        NAME     SAMPLE
MAC1    MACRO
        LOCAL    LLAB            ;(1)
LLAB:                                      Macrodefinition
          :
GLAB:
        BR       $LLAB           ;(2)
        BR       $GLAB           ;(3)
        ENDM
          :
REF1:   MAC1                     ;(4)  ←── Macro reference
          :
        BR       !LLAB           ;(5)
        BR       !GLAB           ;(6)
          :
REF2:   MAC1                     ;(7)  ←── Macro reference
          :
        END
```

(1) This directive defines label "LLAB" as a local symbol.
(2) This instruction references local symbol "LLAB" in macro MAC1.
(3) This instruction references global symbol "GLAB" in macro MAC1.
(4) This directive references macro MAC1.
(5) This instruction references local symbol "LLAB" from outside the definition of macro MAC1. This description cause an error when the source program is assembled.
(6) This instruction references global symbol "GLAB" from outside the definition of macro MAC1.
(7) This directive references macro MAC1. The same macro is referenced twice.

When the source program in the above program is assembled, the macro body will be expanded as shown below.



```
        NAME
         :
REF1:   MAC1
        LOCAL    LLAB
LLAB:
         :
GLAB:
        BR       $LLAB
        BR       $GLAB
         :
        BR       !LLAB
        BR       !GLAB
         :
REF2:   MAC1
        LOCAL    LLAB
LLAB:
         :
GLAB:
        BR       $LLAB
        BR       $GLAB
         :
        END
```

Global symbol "GLAB" has been defined in macro MAC1. Because macro MAC1 is referenced twice, global symbol "GLAB" causes a double definition error as a result of expanding a series of statements in the macro body.

CHAPTER 6. PRODUCT UTILIZATION

There are several ways to effective use this package for assembly of source modules. Only a few of these techniques are introduced in this section.

(1) Specification of assembler options

It is better to specify assembler options you always use at the beginning of each source module. Especially, the PROCESSOR option which cannot be omitted from specification should be specified in the module header. Then, you do not need to specify the option in the command line each time you start up the assembler program. An error will result if you forget to specify this option in the command line and you must start up the assembler again from the beginning with the correct assembler options.

The DEBUG and XREF options should also be specified in the module header.

Example

```
$       PROCESSOR(112)
$       DEBUG
$       XREF

        NAME        TEST

        CSEG
          :
```

(2) Definition of short direct addressing area

The short direct addressing area (addresses 0FE20H to 0FF1FH with uPD78122/78124 or 0FE40H to 0FF1FH with uPD78112) is an area which can be accessed with instructions of short byte length as compared with other data memory areas.
Therefore, by making the best of this area, a program with high memory utilization efficiency can be developed.
So, you must determine the memory map of this short direct addressing area before you start developing any program.
Then, describe the data definition of the short direct addressing area with absolute addresses at the beginning of each source module.

Example

```
        NAME    TEST
WORK1   EQU     0FE20H
WORK2   EQU     0FE21H
          :
WORKX   EQU     0FEFFH
        CSEG
          :
        MOV     WORK1, #00H
          :
        END
```

Describe the data definition of each work area with absolute address(es) at the beginning of the source module.

APPENDIX A. LIST OF RESERVED WORDS

Reserved words are available in five types: machine language instructions, directives, operators, register names, and sfr symbols.
A reserved word is a word or sequence of letters and/or symbols (a character string) which has unique meaning. These words have been reserved beforehand by the assembler and cannot be used for other than the intended purposes.
Types of reserved words that can be described in each field of a source program and a list of reserved words are shown in Tables A-1 and A-2, respectively.

Table A-1. Types of Reserved Words That Can Be Described in Respective Fields of Source Program

Symbol field	All reserved words cannot be described in this field.
Mnemonic field	Only machine language instructions and directives can be described in this field.
Operand field	Only operators, sfr symbols, and register names can be described in this field.
Comment field	All reserved words can be described in this field.

Table A-2. List of Reserved Words

Instructions	ADD	ADDC	ADJBA	ADJBS	AND	AND1
	ADDW	BC	BE	BF	BL	BNC
	BNE	BNL	BNZ	BR	BT	BTCLR
	BZ	CALL	CALLF	CALLT	CLR1	CMP
	CMPW	DBNZ	DEC	DECW	DI	DIVUW
	EI	INC	INCW	MOV	MOVW	MOV1
	MULUW	NOP	NOT1	OR	OR1	POP
	PUSH	RET	RETI	ROL	ROLC	ROR
	RORC	ROL4	ROR4	SEL	SET1	SHL
	SHR	SHLW	SHRW	SUB	SUBC	SUBW
	XCH	XOR	XOR1			
Operators	AND	EQ	GE	GT	HIGH	LE
	LOW	LT	MOD	NE	NOT	OR
	SHL	SHR	XOR			
Special reserved words	$	STKBEG	STKEND			
Directives	BR	CSEG	DB	DBIT	DS	DSEG
	DW	END	ENDM	EQU	EXITM	EXTBIT
	EXTRN	IRP	LOCAL	MACRO	NAME	ORG
	PUBLIC	REPT	SET	SFR	SFRP	

Table A-2. List of Reserved Words (contd)

sfr symbols	ADM	CLOM	CPT0	CPT1	CPT2	CPT3
	CPTM	CR00	CR01	CR02	CR10	CR11
	CR12	CR20	CR21	CR22	CR30	CRC0
	CRC1	CRC2	CSIM	EDVC	FRC	ICR
	IFO	IFOL	IFOH	INTM0	INTM1	ISM0
	ISM0H	ISM0L	IS0H	IST	MK0	MK0L
	MKD0H	MM	P0	P0H	P0L	P1
	P2	P3	P4	P5	P6	PM0
	PM1	PM3	PM5	PM6	PMC3	PR0
	PR0L	PR0H	PRM0	PRM1	PU0	PWM0
	PWM1	PWMC	RTPC	SA	SI0	STBC
	TM0	TM1	TM2	TM3	TMC0	TMC1
	TOC					
Register names	A	AX	B	C	CY	D
	E	H	HL	L	PSW	R0
	R1	R2	R3	R4	R5	R6
	R7	RP0	RP1	RP2	RP3	RB0
	RB1	RB2	RB3	RB4	RB5	RB6
	RB7	SP	STBC			
Segment attributes	CALLT0	FIXED				

APPENDIX B. LIST OF DIRECTIVES

Table B-1. List of Directives

No.	Directive				Function/ classification
	Symbol field	Mnemonic field	Operand field	Comment field	
1	[segment name]	CSEG	[reloc. attr.]	[;comment]	Declares the start of a code segment.
2	[segment name]	DSEG	None	[;comment]	Declares the start of a data segment.
3	[segment name]	BSEG	None	[;comment]	Declares the start of a bit segment.
4	[segment name]	ORG	expres- sion	[;comment]	Declares the start of an absolute segment. (See Note 1.)
5	None	ENDS	None	[;comment]	Indicates the end of the segment.
6	name	EQU	expres- sion	[;comment]	Defines a name. (See Note 2.) name:symbol
7	name	SET	expres- sion	[;comment]	Defines a relocat- able name. (See Note 1.) name:symbol
8	[label:]	DB	[(size)] [initial value [,...]]	[;comment]	Initializes or reserves a byte data area. (See Note 3.) label:symbol
9	[label:]	DW	[(size)] [initial value [,...]]	[;comment]	Initializes or reserves a word data area. label:symbol
10	[label:]	DS	expres- sion	[;comment]	Reserves a byte data area. (See Note 1.) label:symbol

Table B-1. List of Directives (contd)

No.	Directive				Function/ classification
	Symbol field	Mnemonic field	Operand field	Comment field	
11	[name]	DBIT	None	[;comment]	Reserves a bit data area. (See Note 1.) name:symbol
12	[label:]	PUBLIC	symbol name [,...]	[;comment]	Declares an external defini-tion name.
13	[label:]	EXTRN	symbol name [,...]	[;comment]	Declares an external reference name.
14	[label:]	EXTBIT	symbol name [,...]	[;comment]	Declares an external reference name. Symbol names are limited to those having a bit value.
15	[label:]	NAME	module name [,...]	[;comment]	Defines a module name. module name:symbol
16	[label:]	BR	expres-sion [,...]	[;comment]	Automatically selects a Branch instruction. label:symbol
17	macro name	MACRO	[formal parameter [,...]]	[;comment]	Defines a macro. macro name:symbol
18	[label:]	LOCAL	symbol name [,...]	[;comment]	Defines a symbol valid only within the macro. (See Note 4.) label: symbol

Table B-1. List of Directives (contd)

No.	Directive				Function/ classification
	Symbol field	Mnemonic field	Operand field	Comment field	
19	[label:]	REPT	expres- sion	[;comment]	Defines the repeat count in macro- expansion. label: symbol
20	[label:]	IRP	formal parameter, <actual parameter [,...]>	[;comment]	Expands the macro body by replacing formal parameters with actual para- meters. label:symbol
21	None	EXITM	None	[;comment]	Interrupts the macroexpansion. (See Note 4.)
22	None	ENDM	None	[;comment]	Indicates the end of macrodefinition. (See Note 4.)
23	None	END	None	[;comment]	Indicates the end of th source module

NOTE: 1. Forward reference of a symbol is not allowed in the expression described in the Operand field.
2. Neither forward reference of a symbol nor reference of an external reference name is allowed in the expression described in the Operand field.
3. A character string may be described in place of an initial value.
4. This directive can be used only in the macro- definition.

APPENDIX C. INSTRUCTION SET OF uCOM-78K/I

APP C-1. Instruction Set and Its Operation

(1) Representation formats and description methods of operands

Describe an operand in the Operand field of each instruction according to the description method for the operand representation format of the instruction. (For details, see the assembler specifications.) If two or more elements exist in the description method, select one of the elements. Elements written in uppercase letters and symbols +, -, #, $, !, and [ ] are keywords and must be described as is in the Operand field. The meanings of these symbols are as follows:

+ :	Autoincrement
# :	Immediate data
! :	Address by an immediate addressing method
$:	Address by a relative addressing method
/ :	Bit inversion
[ ]:	Indirect addressing

With an immediate data, describe an appropriate numeric value or label as the data. When describing it with a label, symbol #, $, !, or [ ] must also be described.

Table C-1. Representation Formats and Description Methods of Operands

Representation format	Description method	
r	X(R0),A(R1),C(R2),B(R3),E(R4),D(R5),L(R6),H(R7)	
r1	A,B	
r2	B,C	
r3	D,E,E+	
r4	D,E	
rp	AX(RP0),BC(RP1),DE(RP2),HL(RP3)	
sfr	Special function register symbol (see Table C-2.)	
sfrp	Special function register symbol (register capable of 16-bit manipulation; see Table C-2.)	
saddr	FE40H-FF1FH	Immediate data or label (with uPD78112)
	FE20H-FF1FH	Immediate data or label (with uPD78122/78124)
saddrp	FE40H-FF1EH	Immediate data (with bit 0 = 0) or label (in 16-bit manipulation) (with uPD78112)
	FE20H-FF1EH	Immediate data (with bit 0 = 0) or label (in 16-bit manipulation) (with uPD78122/78124)
!addr13	0000H-1FFFH	Immediate data or label (with uPD78112) ... Immediate addressing
!addr16	0000H-3FFFH	Immediate data or label (with uPD78122/78124) ... Immediate addressing
$addr13	0000H-1FFFH	Immediate data or label (with uPD78112) ... Relative addressing
$addr16	0000H-3FFFH	Immediate data or label (with uPD78122/78124) ... Relative addressing
addr11	800H-FFFH	Immediate data or label
addr5	40H-7EH	Immediate data (with bit 0 = 0) or label

Table C-1. Representation Formats and Description Methods of Operands (contd)

Representation format	Description method
word	16-bit immediate data or label
byte	8-bit immediate data or label
bit	3-bit immediate data or label
n	3-bit immediate data (0 - 7)
RBn	RB0 - RB3

NOTE: 1. In register representation formats "r" and "rp", registers may also be described with absolute names (R0 to R7, RP0 to RP3) in addition to absolute names (X, A, C, B, E, D, L, H, AX, BC, DE, HL).

2. In the immediate addressing method, the operand(s) of an instruction can be addressed to any memory space, whereas in the relative addressing method, the operand(s) of an instruction can be addressed only within the range of -128 to +127 bytes from the first address of the next instruction.

(2) Legend of symbols in "Operation" column

A	: A register; 8-bit accumulator
X	: X register
B	: B register
C	: C register
D	: D register
E	: E register
H	: H register
L	: L register
R0-R7	: Register 0 to register 7 (absolute names)
AX	: Register pair AX; 16-bit accumulator
BC	: Register pair BC
DE	: Register pair DE
HL	: Register pair HL
RP0-RP3	: Register pair 0 to register pair 3 (absolute names)
PC	: Program counter
SP	: Stack pointer
PSW	: Program status word
CY	: Carry flag
AC	: Auxiliary carry flag
Z	: Zero flag
RBS0-RBS1	: Register bank select flags
IE	: Interrupt enable flag
STBC	: Standby control register
( )	: Refers to the memory contents indicated by the address or register in parentheses ( ).
xxH	: Hexadecimal number
xH	: High-order 8 bits of 16-bit register
xL	: Low-order 8 bits of 16-bit register

(3) Symbols in "Clocks" column

(a) If "n" is indicated in the "Clocks" column for a Shift or Rotate instruction, the value of "n" refers to the number of bits to be shifted.

(b) The number indicated in ( ) in the "Clocks" column for a conditional branch instruction indicates the number of clocks of the instruction when branching is not executed.

(c) When accessing an SFR by register deferred addressing ([HL]) or indexed addressing (word[r1]), the latter of the two values separated by a slash in the "Clocks" column becomes the number of clocks for the instruction.

(d) In indexed addressing, if an overflow occurs in the result of "word + r1", the number of clocks of the instruction increases to the number indicated in ( ) in the "Clocks" column for the instruction.

(4) Legend of symbols in "Flag" column

Symbol	Description
(blank)	Flag contents will remain unchanged.
0	Flag contents are cleared to 0.
1	Flag contents are set to 1.
x	Flag contents are set or cleared according to the result.
R	Previously saved contents are restored.

(5) Special function registers (SFR)

The SFR refers to a group of registers to which special functions have been assigned, such as mode registers for various peripheral devices and control register.

The special function register group is mapped to an 256-byte area (addresses FF00H to FFFFH).

These SFR registers can be manipulated in various ways with arithmetic operation instructions, move (transfer) instructions, and bit manipulation instructions.

Table C-1 shows a list of special function registers (SFR) with the uPD78112 and Table C-2 shows the same list with the uPD78122/78124.

The meanings of the symbols used in these tables are as follows:

o sfr symbol	Symbol indicating the address of the built-in special function register. This SFR symbol can be described in the Operand field of an instruction.
o R/W	Indicates whether the special function register is readable or writable. R/W: Read/Write R: Read only W: Write only
o Manipulatable.. bit units	Indicates the unit of bits that can be handled when manipulating the contents of each SFR register. If an SFR is capable of manipulation in units of 16 bits, the register can be described as operand "sfrp". If an SFR is capable of 1-bit manipulation, the register can be described in the operand field of a bit manipulation instruction.
o At RESET	Indicates the status of each register when $\overline{\text{RESET}}$ input is applied.

NOTE: In Tables C-1 and C-2, an address to which no special function register is assigned cannot be accessed.

Table C-1. List of Special Function Registers (SFR) with uPD78112

Address	Special function register (SFR) name	sfr symbol	R/W	Manipulatable bit units			At RESET
				1 bit	8 bits	16 bits	
FF01H	Port 1	P1	R/W	o	o	-	Un-known
FF02H	Port 2	P2	R	o	o	-	
FF03H	Port 3	P3	R/W	o	o		
FF04H	Port 4	P4		o	o	-	
FF05H	Port 5	P5		o	o		
FF06H	Port 6	P6		o	o	-	
FF08H	16-bit timer 0/ compare register 0	CR00		-	-	o	
FF09H				-	-		
FF0AH	16-bit timer 0/ compare register 1	CR01		-	-	o	
FF0BH				-	-		
FF0CH	16-bit timer 0/ compare register 2	CR02		-	-	o	
FF0DH				-	-		
FF0EH	16-bit timer 1/ compare register 0	CR10		-	-	o	
FF0FH				-	-		
FF10H	16-bit timer 1/ compare register 1	CR11		-	-	o	
FF11H				-	-		
FF12H	16-bit timer 1/ capture register 2	CR12	R	-	-	o	
FF13H				-	-		
FF14H	16-bit FRC capture register 0	CPT0		-	-	o	
FF15H				-	-		
FF16H	16-bit FRC capture register 1	CPT1		-	-	o	
FF17H				-	-		
FF18H	16-bit FRC capture register 2	CPT2		-	-	o	
FF19H				-	-		
FF1AH	16-bit FRC capture register 3	CPT3		-	-	o	
FF1BH				-	-		
FF1EH	16-bit timer 2/ compare register 0	CR20	R/W	-	-	o	
FF1FH				-	-		
FF21H	Port 1 mode register	PM1	W	-	o	-	3FH
FF23H	Port 3 mode register	PM3		-	o	-	FFH
FF25H	Port 5 mode register	PM5		-	o	-	
FF26H	Port 6 mode register	PM6		-	o	-	
FF30H	16-bit timer/ register 0	TM0	R	-	-	o	Un-known
FF31H				-	-		
FF32H	16-bit timer/ register 1	TM1		-	-	o	
FF33H				-	-		
FF34H	16-bit free-running counter	FRC		-	-	o	
FF35H				-	-		
FF36H	16-bit timer/ register 2	TM2		-	-	o	
FF37H				-	-		
FF38H	Timer control register 0	TMC0	W	-	o	-	
FF39H	Timer control register 1	TMC1	R/W	-	o		00H

Table C-1. List of Special Function Registers (SFR) with uPD78112 (contd)

Address	Special function register (SFR) name	sfr symbol		R/W	Manipulatable bit units			At RESET
					1 bit	8 bits	16 bits	
FF3AH	Capture mode register	CPTM			-	o	-	
FF43H	Port 3 mode control register	PMC3		W	-	o	-	30H
FF50H	Input control register	ICR			-	o	-	
FF53H	Event divider control register	EDVC			-	o	-	Un-known
FF68H	A/D conversion mode register	ADM		R/W	-	o	-	01H
FF6AH	A/D conversion sequential compare register	SA		R	-	o	-	Un-known
FF70H	PWM control register	PWMC			-	o	-	05H
FF72H	PWM modulo register 0	PWM0			-	-	o	
FF73H				W	-	-		Un-known
FF74H	PWM modulo register 1	PWM1			-	-	o	
FF75H					-	-		
FF80H	Serial interface mode register	CSIM			-	o	-	10H
FF86H	Serial shift register	SIO		R/W	-	o	-	Un-known
FFC0H	Standby control register	STBC			-	o	-	00H
FFC4H	Memory mapping register	MM		W	-	o	-	00H
FFE0H	Interrupt request flag register	IF0L	IF0		o	o	o	00H
FFE1H		IF0H			o	o		00H
FFE4H	Interrupt mask register	MK0L	MK0	R/W	o	o	o	FFH
FFE5H		MK0H			o	o		FFH
FFECH	Interrupt service mode register	ISM0L	ISM0		o	o	o	00H
FFEDH		ISM0H			o	o		00H
FFF4H	External interrupt mode register	INTM0		W	-	o	-	50H
FFF5H	External capture input mode register	INTM1			-	o		

Table C-2. List of Special Function Registers (SFR)
with uPD78122/uPD78124

Address	Special function register (SFR) name	sfr symbol	R/W	Manipulatable bit units			At RESET
				1 bit	8 bits	16 bits	
FF00H	Port 0	P0	R/W	o	o	-	Un-known
FF01H	Port 1	P1		o	o		
FF02H	Port 2	P2	R	o	o	-	
FF03H	Port 3	P3		o	o		
FF04H	Port 4	P4		o	o	-	
FF05H	Port 5	P5		o	o		
FF06H	Port 6	P6		o	o		
FF0AH	Port 0 buffer register	P0L		o	o	-	
FF0BH	Port 0 buffer register	P0H		o	o		
FF0CH	Real-time output port control register	RTPC		o	o	-	00H
FF10H	16-bit compare register 0	CR00	R/W	-	-	o	Un-known
FF11H				-	-		
FF12H	16-bit compare register 1	CR01		-	-	o	
FF13H				-	-		
FF14H	8-bit compare register (CH1-0)	CR10		-	o	-	
FF15H	8-bit compare register (CH2-0)	CR20		-	o		
FF16H	8-bit compare register (CH2-1)	CR21		-	o	-	
FF17H	8-bit compare register	CR30		-	o		
FF18H	16-bit capture register	CR02	R	-	-	o	
FF19H				-	-		
FF1AH	8-bit capture register	CR22		-	o	-	
FF1CH	8-bit capture/compare register (CH1-1)	CR11	R/W	-	o	-	
FF20H	Port 0 mode register	PM0	W	-	o	-	
FF21H	Port 1 mode register	PM1		-	o	-	FFH
FF23H	Port 3 mode register	PM3		-	o	-	
FF25H	Port 5 mode register	PM5		-	o	-	FFH
FF26H	Port 6 mode register	PM6		-	o	-	FFH
FF30H	Capture/compare control register 0	CRC0		-	o	-	10H
FF31H	Timer output control register	TOC		-	o		00H
FF32H	Capture/compare control register 1	CRC1		-	o	-	
FF34H	Capture/compare control register 2	CRC2		-	o	-	
FF40H	Pull-up resistor option register	PUO	R/W	o	o	-	
FF43H	Port 3 mode control register	PMC3	R/W	o	o	-	

Phase-out/Discontinued

Table C-2. List of Special Function Registers (SFR) with uPD78122/uPD78124 (contd)

Address	Special function register (SFR) name	sfr symbol	R/W	Manipulatable bit units			At RESET
				1 bit	8 bits	16 bits	
FF50H	16-bit timer/register 0	TM0	R	-	-	o	Un-known
FF51H				-	-		
FF52H	8-bit timer/register: CH-1	TM1		-	o	-	
FF54H	8-bit timer/register: CH-2	TM2		-	o	-	
FF56H	8-bit timer/register for BRG	TM3		-	o	-	
FF5CH	Prescaler mode register 0	PRM0	W	-	o	-	00H
FF5DH	Timer control register 0	TMC0	R/W	-	o		
FF5EH	Prescaler mode register 1	PRM1	W	-	o	-	
FF5FH	Timer control register 1	TMC1	R/W	-	o		
FF7FH	Clock output mode register	CLOM		o	o	-	
FF80H	Clock-synchronized serial interface mode register	CSIM	R/W	o	o	-	00H
FF82H	Serial bus interface control register	SBIC		o	o	-	
FF86H	Serial shift register	SIO		-	o	-	Un-known
FFC0H	Standby control register	STBC		-	o	-	
FFC4H	Memory expansion mode register	MM		o	o		00H
FFE0H	Interrupt request flag register L	IF0L		o	o	o	00H
FFE1H	Interrupt request flag register H	IF0H		o	o		
FFE4H	Interrupt mask flag register L	MK0L		o	o	o	FFH
FFE5H	Interrupt mask flag register H	MK0H		o	o		
FFE8H	Priority specification flag register L	PR0L		o	o	o	
FFE9H	Priority specification flag register H	PR0H		o	o		

Table C-2. List of Special Function Registers (SFR)
with uPD78122/uPD78124 (contd)

Address	Special function register (SFR) name	sfr symbol	R/W	Manipulatable bit units			At RESET
				1 bit	8 bits	16 bits	
FFECH	Interrupt handling mode specification flag register L	ISM0L	R/W	o	o	o	00H
FFEDH	Interrupt handling mode specification flag register H	ISM0H	R/W	o	o	o	00H
FFF4H	External interrupt mode register 0	INTM0	R/W	o	o	-	00H
FFF5H	External interrupt mode register 1	INTM1	R/W	o	o	-	00H
FFF8H	Interrupt status register	IST	R/W	o	o	-	00H

Table C-3. Operation of Each uCOM-78K/I Instruction (1/8)

In the table, description "addr13" in the Operand column applies to the uPD78112. For the uPD78122 or uPD78124, change this to read as "addr16".

Group	Mnemonic	Operand	Bytes	Clocks	Operation	Flag		
						Z	AC	CY
8-bit Data Transfer	MOV	r,#byte	2	2	r ← byte			
		saddr,#byte	3	3	(saddr) ← byte			
		sfr,#byte (Note 1)	3	5	sfr ← byte			
		r,r	2	2	r ← r			
		A,r	1	2	A ← r			
		A,saddr	2	2	A ← (saddr)			
		saddr,A	2	3	(saddr) ← A			
		A,sfr	2	4	A ← sfr			
		sfr,A	2	5	sfr ← A			
		A,[r3] (Note2)	1	5/6	A ← (FE00H+r3) r3=40H-FFH			
		[r3],A (Note 2)	1	5/6	(FE00H+r3) ← A r3=40H-FFH			
		A,[HL]	1	5/7	A ← (HL)			
		[HL],A	1	5/7	(HL) ← A			
		A,word[r1]	4	7(8)/9(10)	A ← (word+r1)			
		word[r1],A	4	7(8)/9(10)	(word+r1) ← A			
		PSW,#byte	3	5	PSW ← byte	×	×	×
		PSW,A	2	5	PSW ← A	×	×	×
		A,PSW	2	4	A ← PSW			
	XCH	A,r	1	4	A ↔ r			
		A,saddr	2	4	A ↔ (saddr)			
		A,sfr	3	10	A ↔ sfr			
		A,[r4]	1	8	A ↔ (FE00H+r4) r4=40H-FFH			
16-bit Data Transfer	MOVW	rp,#word	3	3	rp ← word			
		saddrp,#word	4	4	(saddrp+1)(saddrp) ← word			
		sfrp,#word	4	8	sfrp ← word			
		rp,rp	2	4	rp ← rp			
		AX,saddrp	2	6	AX ← (saddrp+1)(saddrp)			
		saddrp,AX	2	5	(saddrp+1)(saddrp) ← AX			
		AX,sfrp	2	10	AX ← sfrp			
		sfrp,AX	2	9	sfrp ← AX			

NOTE: 1. If the STBC register is described as the operand "sfr" of a MOV instruction, this instruction becomes a dedicated instruction which differs from that shown in this table in the number of bytes and the number of clocks. (See CPU control instructions.)

2. If "E+" is described in the operand "r3" of a MOV instruction, the contents of the E register are incremented by 1 after the execution of the MOV instruction and the number of clocks becomes 6.

Table C-3. Operation of Each uCOM-78K/I Instruction (2/8)

Group	Mnemonic	Operand	Bytes	Clocks	Operation	Flag Z	Flag AC	Flag CY
8-bit Arithmetic Operation	ADD	A,#byte	2	2	A,CY ← A+byte	×	×	×
		saddr,#byte	3	3	(saddr),CY ← (saddr)+byte	×	×	×
		sfr,#byte	4	9	sfr,CY ← sfr+byte	×	×	×
		r,r	2	3	r,CY ← r + r	×	×	×
		A,saddr	2	3	A,CY ← A+(saddr)	×	×	×
		A,sfr	3	7	A,CY ← A+sfr	×	×	×
		A,[r4]	2	7	A,CY ← A+(FE00H+r4) r4=40H-FFH	×	×	×
		A,[HL]	2	8/10	A,CY ← A+(HL)	×	×	×
	ADDC	A,#byte	2	2	A,CY ← A+byte+CY	×	×	×
		saddr,#byte	3	3	(saddr),CY ← (saddr)+byte+CY	×	×	×
		sfr,#byte	4	9	sfr,CY ← sfr+byte+CY	×	×	×
		r,r	2	3	r,CY ← r+r+CY	×	×	×
		A,saddr	2	3	A,CY ← A+(saddr)+CY	×	×	×
		A,sfr	3	7	A,CY ← A+sfr+CY	×	×	×
		A,[r4]	2	7	A,CY←A+(FE00H+r4)+CY r4=40H-FFH	×	×	×
		A,[HL]	2	8/10	A,CY ← A+(HL)+CY	×	×	×
	SUB	A,#byte	2	2	A,CY ← A-byte	×	×	×
		saddr,#byte	3	3	(saddr),CY ← (saddr)-byte	×	×	×
		sfr,#byte	4	9	sfr,CY ← sfr-byte	×	×	×
		r,r	2	3	r,CY ← r-r	×	×	×
		A,saddr	2	3	A,CY ← A-(saddr)	×	×	×
		A,sfr	3	7	A,CY ← A-sfr	×	×	×
		A,[r4]	2	7	A,CY ← A-(FE00H+r4) r4=40H-FFH	×	×	×
		A,[HL]	2	8/10	A,CY ← A-(HL)	×	×	×
	SUBC	A,#byte	2	2	A,CY ← A-byte-CY	×	×	×
		saddr,#byte	3	3	(saddr),CY ← (saddr)-byte-CY	×	×	×
		sfr,#byte	4	9	sfr,CY ← sfr-byte-CY	×	×	×
		r,r	2	3	r,CY ← r-r-CY	×	×	×
		A,saddr	2	3	A,CY ← A-(saddr)-CY	×	×	×
		A,sfr	3	7	A,CY ← A-sfr-CY	×	×	×
		A,[r4]	2	7	A,CY←A-(FE00H+r4)-CY r4=40H-FFH	×	×	×
		A,[HL]	2	8/10	A,CY ← A-(HL)-CY	×	×	×

Table C-3. Operation of Each uCOM-78K/I Instruction (3/8)

Group	Mnemonic	Operand	Bytes	Clocks	Operation	Flag		
						Z	AC	CY
8-bit Arithmetic Operation	AND	A,#byte	2	2	A ← A∧byte	×		
		saddr,#byte	3	3	(saddr) ← (saddr)∧byte	×		
		sfr,#byte	4	9	sfr ← sfr∧byte	×		
		r,r	2	3	r ← r∧r	×		
		A,saddr	2	3	A ← A∧(saddr)	×		
		A,sfr	3	7	A ← A∧sfr	×		
		A,[r4]	2	7	A ← A∧(FE00H+r4) r4=40H-FFH	×		
		A,[HL]	2	8/10	A ← A∧(HL)	×		
	OR	A,#byte	2	2	A ← A∨byte	×		
		saddr,#byte	3	3	(saddr) ← (saddr)∨byte	×		
		sfr,#byte	4	9	sfr ← sfr∨byte	×		
		r,r	2	3	r ← r∨r	×		
		A,saddr	2	3	A ← A∨(saddr)	×		
		A,sfr	3	7	A ← A∨sfr	×		
		A,[r4]	2	7	A ← A∨(FE00H+r4) r4=40H-FFH	×		
		A,[HL]	2	8/10	A ← A∨(HL)	×		
	XOR	A,#byte	2	2	A ← A∀byte	×		
		saddr,#byte	3	3	(saddr) ← (saddr)∀byte	×		
		sfr,#byte	4	9	sfr ← sfr∀byte	×		
		r,r	2	3	r ← r∀r	×		
		A,saddr	2	3	A ← A∀(saddr)	×		
		A,sfr	3	7	A ← A∀sfr	×		
		A,[r4]	2	7	A ← A∀(FE00H+r4) r4=40H-FFH	×		
		A,[HL]	2	8/10	A ← A∀(HL)	×		
	CMP	A,#byte	2	2	A − byte	×	×	×
		saddr,#byte	3	3	(saddr) − byte	×	×	×
		sfr,#byte	4	7	sfr − byte	×	×	×
		r,r	2	3	r − r	×	×	×
		A,saddr	2	3	A − (saddr)	×	×	×
		A,sfr	3	7	A − sfr	×	×	×
		A,[r4]	2	7	A − (FE00H+r4) r4=40H-FFH	×	×	×
		A,[HL]	2	8/10	A − (HL)	×	×	×

Table C-3. Operation of Each uCOM-78K/I Instruction (4/8)

Group	Mnemonic	Operand	Bytes	Clocks	Operation	フ	ラ	グ
						Z	AC	CY
16-bit Arithmetic Operation	ADDW	AX,#word	3	4	AX,CY ← AX+word	×	×	×
		AX,rp	2	6	AX,CY ← AX+rp	×	×	×
		AX,saddrp	2	7	AX,CY ← AX+(saddrp+1)(saddrp)	×	×	×
		AX,sfr	3	13	AX,CY ← AX+sfrp	×	×	×
	SUBW	AX,#word	3	4	AX,CY ← AX−word	×	×	×
		AX,rp	2	6	AX,CY ← AX−rp	×	×	×
		AX,saddrp	2	7	AX,CY ← AX−(saddrp+1)(saddrp)	×	×	×
		AX,sfrp	3	13	AX,CY ← AX−sfrp	×	×	×
	CMPW	AX,#word	3	3	AX − word	×	×	×
		AX,rp	2	5	AX − rp	×	×	×
		AX,saddrp	2	6	AX − (saddrp+1)(saddrp)	×	×	×
		AX,sfrp	3	12	AX − sfrp	×	×	×
Multiply/Divide	MULUW	r	2	43	AX(low 16 bits), r(high 16bits)←AX×r			
	DIVUW	r	2	71	AX(quo), r(rem)←AX÷r			
Increment/Decrement	INC	r	1	2	r ← r+1	×	×	
		saddr	2	2	(saddr) ← (saddr)+1	×	×	
	DEC	r	1	2	r ← r−1	×	×	
		saddr	2	2	(saddr) ← (saddr)−1	×	×	
	INCW	rp	1	3	rp ← rp+1			
	DECW	rp	1	3	rp ← rp−1			
Shift/Rotate	ROR	r,n	2	3+2n	$(CY, r_7 \leftarrow r_0, r_{m-1} \leftarrow r_m) \times nT$ n=0-7			×
	ROL	r,n	2	3+2n	$(CY, r_0 \leftarrow r_7, r_{m+1} \leftarrow r_m) \times nT$ n=0-7			×
	RORC	r,n	2	3+2n	$(CY \leftarrow r_0, r_7 \leftarrow CY, r_{m-1} \leftarrow r_m) \times nT$ n=0-7			×
	ROLC	r,n	2	3+2n	$(CY \leftarrow r_7, r_0 \leftarrow CY, r_{m+1} \leftarrow r_m) \times nT$ n=0-7			×
	SHR	r,n	2	3+2n	$(CY \leftarrow r_0, r_7 \leftarrow 0, r_{m-1} \leftarrow r_m) \times nT$ n=0-7	×	0	×
	SHL	r,n	2	3+2n	$(CY \leftarrow r_7, r_0 \leftarrow 0, r_{m+1} \leftarrow r_m) \times nT$ n=0-7	×	0	×
	SHRW	rp,n	2	3+3n	$(CY \leftarrow rp_0, rp_{15} \leftarrow 0, rp_{m-1} \leftarrow rp_m) \times nT$ n=0-7	×	0	×
	SHLW	rp,n	2	3+3n	$(CY \leftarrow rp_{15}, rp_0 \leftarrow 0, rp_{m+1} \leftarrow rp_m) \times nT$ n=0-7	×	0	×
	ROR4	[r4]	2	22	$A_{3-0} \leftarrow (FE00+r4)_{3-0}$, $(FE00+r4)_{7-4} \leftarrow A_{3-0}$, $(FE00+r4)_{3-0} \leftarrow (FE00+r4)_{7-4}$			
	ROL4	[r4]	2	23	$A_{3-0} \leftarrow (FE00+r4)_{7-4}$, $(FE00+r4)_{3-0} \leftarrow A_{3-0}$, $(FE00+r4)_{7-4} \leftarrow (FE00+r4)_{3-0}$			

* quo: quotient
rem: remainder
nT : n times

Phase-out/Discontinued

Table C-3. Operation of Each uCOM-78K/I Instruction (5/8)

Group	Mnemonic	Operand	Bytes	Clocks	Operation	Flag		
						Z	AC	CY
BCD Adjust	ADJBA		1	3	Decimal Adjust Accumulator after Addition	×	×	×
	ADJBS		1	3	Decimal Adjust Accumulator after Subtract	×	×	×
Bit Manipulation	MOV1	CY,saddr.bit	3	5	CY ← (saddr.bit)			×
		CY,sfr.bit	3	7	CY ← sfr.bit			×
		CY,A.bit	2	5	CY ← A.bit			×
		CY,X.bit	2	5	CY ← X.bit			×
		CY,PSW.bit	2	5	CY ← PSW.bit			×
		saddr.bit,CY	3	8	(saddr.bit) ← CY			
		sfr.bit,CY	3	12	sfr.bit ← CY			
		A.bit,CY	2	8	A.bit ← CY			
		X.bit,CY	2	8	X.bit ← CY			
		PSW.bit,CY	2	7	PSW.bit ← CY	×	×	
	AND1	CY,saddr.bit	3	5	CY ← CY $\wedge$ (saddr.bit)			×
		CY,/saddr.bit	3	5	CY ← CY $\wedge$ $(\overline{\text{saddr.bit}})$			×
		CY,sfr.bit	3	7	CY ← CY $\wedge$ sfr.bit			×
		CY,/sfr.bit	3	7	CY ← CY $\wedge$ $\overline{\text{sfr.bit}}$			×
		CY,A.bit	2	5	CY ← CY $\wedge$ A.bit			×
		CY,/A.bit	2	5	CY ← CY $\wedge$ $\overline{\text{A.bit}}$			×
		CY,X.bit	2	5	CY ← CY $\wedge$ X.bit			×
		CY,/X.bit	2	5	CY ← CY $\wedge$ $\overline{\text{X.bit}}$			×
		CY,PSW.bit	2	5	CY ← CY $\wedge$ PSW.bit			×
		CY,/PSW.bit	2	5	CY ← CY $\wedge$ $\overline{\text{PSW.bit}}$			×
	OR1	CY,saddr.bit	3	5	CY ← CY $\vee$ (saddr.bit)			×
		CY,/saddr.bit	3	5	CY ← CY $\vee$ $(\overline{\text{saddr.bit}})$			×
		CY,sfr.bit	3	7	CY ← CY $\vee$ sfr.bit			×
		CY,/sfr.bit	3	7	CY ← CY $\vee$ $\overline{\text{sfr.bit}}$			×
		CY,A.bit	2	5	CY ← CY $\vee$ A.bit			×
		CY,/A.bit	2	5	CY ← CY $\vee$ $\overline{\text{A.bit}}$			×
		CY,X.bit	2	5	CY ← CY $\vee$ X.bit			×
		CY,/X.bit	2	5	CY ← CY $\vee$ $\overline{\text{X.bit}}$			×
		CY,PSW.bit	2	5	CY ← CY $\vee$ PSW.bit			×
		CY,/PSW.bit	2	5	CY ← CY $\vee$ $\overline{\text{PSW.bit}}$			×

Table C-3. Operation of Each uCOM-78K/I Instruction (6/8)

Group	Mnemonic	Operand	Bytes	Clocks	Operation	Flag		
						Z	AC	CY
Bit Manipulation	XOR1	CY,saddr.bit	3	5	CY ← CY ∀ (saddr.bit)			×
		CY,sfr.bit	3	7	CY ← CY ∀ sfr.bit			×
		CY,A.bit	2	5	CY ← CY ∀ A.bit			×
		CY,X.bit	2	5	CY ← CY ∀ X.bit			×
		CY,PSW.bit	2	5	CY ← CY ∀ PSW.bit			×
	SET1	saddr.bit	2	3	(saddr.bit) ← 1			
		sfr.bit	3	10	sfr.bit ← 1			
		A.bit	2	6	A.bit ← 1			
		X.bit	2	6	X.bit ← 1			
		PSW.bit	2	5	PSW.bit ← 1	×	×	×
	CLR1	saddr.bit	2	3	(saddr.bit) ← 0			
		sfr.bit	3	10	sfr.bit ← 0			
		A.bit	2	6	A.bit ← 0			
		X.bit	2	6	X.bit ← 0			
		PSW.bit	2	5	PSW.bit ← 0	×	×	×
	NOT1	saddr.bit	3	6	(saddr.bit) ← ($\overline{\text{saddr.bit}}$)			
		sfr.bit	3	10	sfr.bit ← $\overline{\text{sfr.bit}}$			
		A.bit	2	6	A.bit ← $\overline{\text{A.bit}}$			
		X.bit	2	6	X.bit ← $\overline{\text{X.bit}}$			
		PSW.bit	2	5	PSW.bit ← $\overline{\text{PSW.bit}}$	×	×	×
	SET1	CY	1	2	CY ← 1			1
	CLR1	CY	1	2	CY ← 0			0
	NOT1	CY	1	2	CY ← $\overline{\text{CY}}$			×
Call/Return	CALL	!addr13	3	9	(SP−1)(SP−2) ← PC+3,PC ← !addr13, SP ← SP−2			
	CALLF	!addr11	2	9	(SP−1)(SP−2) ← PC+2,PC_{12-11}←01, PC_{10-0} ← !addr11,SP ← SP−2			
	CALLT	[addr5]	1	12	(SP−1)(SP−2) ← PC+1,PC_H ← (addr5+1),PC_L ← (addr5),SP ← SP−2			
	RET		1	8	PC_L ← (SP),PC_H ← (SP+1),SP ← SP+2			
	RETI		1	10	PC_L ← (SP),PC_H ← (SP+1),PSW← (SP+2),SP ← SP+3	R	R	R

Table C-3. Operation of Each uCOM-78K/I Instruction (7/8)

Group	Mnemonic	Operand	Bytes	Clocks	Operation	Flag Z	Flag AC	Flag CY
Stack Manipulation	PUSH	rp	1	7	$(SP-1) \leftarrow rp_H, (SP-2) \leftarrow rp_L, SP \leftarrow SP-2$			
		PSW	1	3	$(SP-1) \leftarrow PSW, SP \leftarrow SP-1$			
	POP	rp	1	8	$rp_L \leftarrow (SP), rp_H \leftarrow (SP+1), SP \leftarrow SP+2$			
		PSW	1	4	$PSW \leftarrow (SP), SP \leftarrow SP+1$	R	R	R
	MOV	SP,#byte	3	5	SP ← byte			
		SP,A	2	5	SP ← A			
		A,SP	2	4	A ← SP			
Uncondi-tional Branch	BR	!addr13	3	5	PC ← !addr13			
		rp	2	5	$PC_H \leftarrow rp_H, PC_L \leftarrow rp_L$			
		$addr13	2	4	PC ← $addr13			
Conditional Branch	BC / BL	$addr13	2	4(2)	PC ← $addr13 if CY=1			
	BNC / BNL	$addr13	2	4(2)	PC ← $addr13 if CY=0			
	BZ / BE	$addr13	2	4(2)	PC ← $addr13 if Z=1			
	BNZ / BNE	$addr13	2	4(2)	PC ← $addr13 if Z=0			
	BT	saddr.bit,$addr13	3	6(4)	PC ← $addr13 if (saddr.bit)=1			
		sfr.bit,$addr13	4	9(7)	PC ← $addr13 if sfr.bit=1			
		A.bit,$addr13	3	7(5)	PC ← $addr13 if A.bit=1			
		X.bit,$addr13	3	7(5)	PC ← $addr13 if X.bit=1			
		PSW.bit,$addr13	3	7(5)	PC ← $addr13 if PSW.bit=1			
	BF	saddr.bit,$addr13	4	7(5)	PC ← $addr13 if (saddr.bit)=0			
		sfr.bit,$addr13	4	9(7)	PC ← $addr13 if sfr.bit=0			
		A.bit,$addr13	3	7(5)	PC ← $addr13 if A.bit=0			
		X.bit,$addr13	3	7(5)	PC ← $addr13 if X.bit=0			
		PSW.bit,$addr13	3	7(5)	PC ← $addr13 if PSW.bit=0			

Table C-3. Operation of Each uCOM-78K/I Instruction (8/8)

Group	Mnemonic	Operand	Bytes	Clocks	Operation	Flag
						Z AC CY
Conditional Branch	BTCLR	saddr.bit,$addr13	4	9(5)	PC ← $addr13 if (saddr.bit)=1 then reset (saddr.bit)	
		sfr.bit,$addr13	4	13(7)	PC ← $addr13 if sfr.bit=1 then reset sfr.bit	
		A.bit,$addr13	3	9(5)	PC ← $addr13 if A.bit=1 then reset A.bit	
		X.bit,$addr13	3	9(5)	PC ← $addr13 if X.bit=1 then reset X.bit	
		PSW.bit,$addr13	3	8(5)	PC ← $addr13 if PSW.bit=1 then reset PSW.bit	× × ×
	DBNZ	r2,$addr13	2	5(3)	r2 ← r2−1, then PC ← $addr13 if r2≠ 0	
		saddr,$addr13	3	6(4)	saddr ← saddr−1, then PC ← $addr13 if saddr≠0	
CPU Control	MOV	STBC,#byte	4	12	STBC ← byte	
	SEL	RBn	2	2	RBS1-0 ← n n=0-3	
	NOP		1	2	No Operation	
	EI		1	2	IE ←1 (Enable Interrupt)	
	DI		1	2	IE ←0 (Disable Interrupt)	

APP C-2. Instruction Codes

(1) Legend of symbols in "Instruction Code" column

r

R_2	R_1	R_0	reg	
R_6	R_5	R_4		
0	0	0	R0	X
0	0	1	R1	A
0	1	0	R2	C
0	1	1	R3	B
1	0	0	R4	E
1	0	1	R5	D
1	1	0	R6	L
1	1	1	R7	H

r1

R_3	reg
0	A
1	B

r2

R_0	reg
0	C
1	B

r3

R_1	R_0	reg
0	0	E
0	1	E+
1	0	D

r4

R_1	reg
R_2	
R_4	
0	E
1	D

rp

P_1	P_0	reg-pair	
P_2	P_1		
P_6	P_5		
0	0	RP0	AX
0	1	RP1	BC
1	0	RP2	DE
1	1	RP3	HL

Bn	:	Immediate data corresponding to "bit"
Nn	:	Immediate data corresponding to "n"
Data	:	8-bit immediate data corresponding to "byte"
Low/High Byte	:	16-bit immediate data corresponding to "word"
Saddr-offset	:	Low-order 8-bit offset data of the 16-bit address corresponding to "saddr"
Sfr-offset	:	Low-order 8-bit offset data of the 16-bit address corresponding to "sfr" (special function register)
Low/High offset	:	16-bit offset data corresponding to "word" in indexed addressing
Low/High Addr.	:	16-bit immediate data corresponding to "addr13"

jdisp	: Signed twos complement data (8 bits) of relative address distance between the first address of the next instruction and the branch destination address
fa	: Low-order 11 bits of the immediate data corresponding to "addr11"
ta	: Low-order 5 bits of the immediate data corresponding to "addr5 x 1/2"

Note: If the 1st and 2nd operands in the Operand field of an instruction are both registers or register pairs, the instruction code becomes as follows:
Of the byte specifying registers, the high-order 4 bits of the byte become a code specifying the 2nd operand and the low-order 4 bits becomes a code specifying the 1st operand.

Example: MOV r,r

Instruction code

$\boxed{0\ 0\ 1\ 0\ \ 0\ 1\ 0\ 0}\ \boxed{0\ R_6 R_5 R_4\ 0\ R_2 R_1 R_0}$

To specify A register as the 1st operand and L register as the 2nd operand, describe the MOV instruction as follows:

MOV A, L

The instruction code for this instruction becomes as follows:

Instruction code

$\boxed{0\ 0\ 1\ 0\ \ 0\ 1\ 0\ 0}\ \boxed{0\ \underbrace{1\ 1\ 0}\ \ 0\ \underbrace{0\ 0\ 1}}$

Code specifying A register

Code specifying L register

Table C-4. uCOM-78K/I Instruction Codes (1/7)

Group	Mnemonic	Operand	Instruction Code			
			B 1	B 2	B 3	B 4
8-bit Data Transfer	MOV	r,#byte	1 0 1 1 1 $R_2R_1R_0$	← Data →		
		saddr,#byte	0 0 1 1 1 0 1 0	← Saddr-offset →	Data	
		sfr,#byte	0 0 1 0 1 0 1 1	← Sfr-offset →	Data	
		r,r	0 0 1 0 0 1 0 0	0 $R_6R_5R_4$ 0 $R_2R_1R_0$		
		A,r	1 1 0 1 0 $R_2R_1R_0$			
		A,saddr	0 0 1 0 0 0 0 0	← Saddr-offset →		
		saddr,A	0 0 1 0 0 0 1 0	← Saddr-offset →		
		A,sfr	0 0 0 1 0 0 0 0	← Sfr-offset →		
		sfr,A	0 0 0 1 0 0 1 0	← Sfr-offset →		
		A,[r3]	0 1 1 1 1 1 R_1R_0			
		[r3],A	0 1 1 1 1 0 R_1R_0			
		A,[HL]	0 1 0 1 1 1 0 1			
		[HL],A	0 1 0 1 0 1 0 1			
		A,word[r1]	0 0 0 0 1 0 1 0	0 0 $R_5$1 0 0 0 0	Low offset	High offset
		word[r1],A	0 0 0 0 1 0 1 0	1 0 $R_5$1 0 0 0 0	Low offset	High offset
		PSW,#byte	0 0 1 0 1 0 1 1	1 1 1 1 1 1 1 0	Data	
		PSW,A	0 0 0 1 0 0 1 0	1 1 1 1 1 1 1 0		
		A,PSW	0 0 0 1 0 0 0 0	1 1 1 1 1 1 1 0		
	XCH	A,r	1 1 0 1 1 $R_2R_1R_0$			
		A,saddr	0 0 1 0 0 0 0 1	← Saddr-offset →		
		A,sfr	0 0 0 0 0 0 0 1	0 0 1 0 0 0 0 1	Sfr-offset	
		A,[r4]	0 1 1 1 1 $R_2$1 1			
16-bit Data Transfer	MOVW	rp,#word	0 1 1 0 0 $P_2P_1$0	← Low Byte →	High Byte	
		saddrp,#word	0 0 0 0 1 1 0 0	← Saddr-offset →	Low Byte	High Byte
		sfrp,#word	0 0 0 0 1 0 1 1	← Sfr-offset →	Low Byte	High Byte
		rp,rp	0 0 1 0 0 1 0 0	0 $P_6P_5$0 1 $P_2P_1$0		
		AX,saddrp	0 0 0 1 1 1 0 0	← Saddr-offset →		
		saddrp,AX	0 0 0 1 1 0 1 0	← Saddr-offset →		
		AX,sfrp	0 0 0 1 0 0 0 1	← Sfr-offset →		
		sfrp,AX	0 0 0 1 0 0 1 1	← Sfr-offset →		

Table C-4. uCOM-78K/I Instruction Codes (2/7)

Group	Mnemonic	Operand	Instruction Code			
			B 1	B 2	B 3	B 4
8-bit Arithmetic Operation	ADD	A,#byte	1 0 1 0 1 0 0 0	← Data →		
		saddr,#byte	0 1 1 0 1 0 0 0	← Saddr-offset →	Data	
		sfr,#byte	0 0 0 0 0 0 0 1	0 1 1 0 1 0 0 0	Sfr-offset	Data
		r,r	1 0 0 0 1 0 0 0	0 $R_6R_5R_4$ 0 $R_2R_1R_0$		
		A,saddr	1 0 0 1 1 0 0 0	← Saddr-offset →		
		A,sfr	0 0 0 0 0 0 0 1	1 0 0 1 1 0 0 0	Sfr-offset	
		A,[r4]	0 0 0 1 0 1 1 0	0 1 1 R_4 1 0 0 0		
		A,[HL]	0 0 0 1 0 1 1 0	0 1 0 1 1 0 0 0		
	ADDC	A,#byte	1 0 1 0 1 0 0 1	← Data →		
		saddr,#byte	0 1 1 0 1 0 0 1	← Saddr-offset →	Data	
		sfr,#byte	0 0 0 0 0 0 0 1	0 1 1 0 1 0 0 1	Sfr-offset	Data
		r,r	1 0 0 0 1 0 0 1	0 $R_6R_5R_4$ 0 $R_2R_1R_0$		
		A,saddr	1 0 0 1 1 0 0 1	← Saddr-offset →		
		A,sfr	0 0 0 0 0 0 0 1	1 0 0 1 1 0 0 1	Sfr-offset	
		A,[r4]	0 0 0 1 0 1 1 0	0 1 1 R_4 1 0 0 1		
		A,[HL]	0 0 0 1 0 1 1 0	0 1 0 1 1 0 0 1		
	SUB	A,#byte	1 0 1 0 1 0 1 0	← Data →		
		saddr,#byte	0 1 1 0 1 0 1 0	← Saddr-offset →	Data	
		sfr,#byte	0 0 0 0 0 0 0 1	0 1 1 0 1 0 1 0	Sfr-offset	Data
		r,r	1 0 0 0 1 0 1 0	0 $R_6R_5R_4$ 0 $R_2R_1R_0$		
		A,saddr	1 0 0 1 1 0 1 0	← saddr-offset →		
		A,sfr	0 0 0 0 0 0 0 1	1 0 0 1 1 0 1 0	Sfr-offset	
		A,[r4]	0 0 0 1 0 1 1 0	0 1 1 R_4 1 0 1 0		
		A,[HL]	0 0 0 1 0 1 1 0	0 1 0 1 1 0 1 0		
	SUBC	A,#byte	1 0 1 0 1 0 1 1	← Data →		
		saddr,#byte	0 1 1 0 1 0 1 1	← saddr-offset →	Data	
		sfr,#byte	0 0 0 0 0 0 0 1	0 1 1 0 1 0 1 1	Sfr-offset	Data
		r,r	1 0 0 0 1 0 1 1	0 $R_6R_5R_4$ 0 $R_2R_1R_0$		
		A,saddr	1 0 0 1 1 0 1 1	← saddr-offset →		
		A,sfr	0 0 0 0 0 0 0 1	1 0 0 1 1 0 1 1	Sfr-offset	
		A,[r4]	0 0 0 1 0 1 1 0	0 1 1 R_4 1 0 1 1		
		A,[HL]	0 0 0 1 0 1 1 0	0 1 0 1 1 0 1 1		

Table C-4. uCOM-78K/I Instruction Codes (3/7)

Group	Mnemonic	Operand	Instruction Code			
			B 1	B 2	B 3	B 4
8-bit Arithmetic Operation	AND	A,#byte	1 0 1 0 1 1 0 0	← Data →		
		saddr,#byte	0 1 1 0 1 1 0 0	← Saddr-offset →	Data	
		sfr,#byte	0 0 0 0 0 0 0 1	0 1 1 0 1 1 0 0	Sfr-offset	Data
		r,r	1 0 0 0 1 1 0 0	0 $R_6R_5R_4$ 0 $R_2R_1R_0$		
		A,saddr	1 0 0 1 1 1 0 0	← Saddr-offset →		
		A,sfr	0 0 0 0 0 0 0 1	1 0 0 1 1 1 0 0	Sfr-offset	
		A,[r4]	0 0 0 1 0 1 1 0	0 1 1 R_4 1 1 0 0		
		A,[HL]	0 0 0 1 0 1 1 0	0 1 0 1 1 1 0 0		
	OR	A,#byte	1 0 1 0 1 1 1 0	← Data →		
		saddr,#byte	0 1 1 0 1 1 1 0	← Saddr-offset →	Data	
		sfr,#byte	0 0 0 0 0 0 0 1	0 1 1 0 1 1 1 0	Sfr-offset	Data
		r,r	1 0 0 0 1 1 1 0	0 $R_6R_5R_4$ 0 $R_2R_1R_0$		
		A,saddr	1 0 0 1 1 1 1 0	← Saddr-offset →		
		A,sfr	0 0 0 0 0 0 0 1	1 0 0 1 1 1 1 0	Sfr-offset	
		A,[r4]	0 0 0 1 0 1 1 0	0 1 1 R_4 1 1 1 0		
		A,[HL]	0 0 0 1 0 1 1 0	0 1 0 1 1 1 1 0		
	XOR	A,#byte	1 0 1 0 1 1 0 1	← Data →		
		saddr,#byte	0 1 1 0 1 1 0 1	← Saddr-offset →	Data	
		sfr,#byte	0 0 0 0 0 0 0 1	0 1 1 0 1 1 0 1	Sfr-offset	Data
		r,r	1 0 0 0 1 1 0 1	0 $R_6R_5R_4$ 0 $R_2R_1R_0$		
		A,saddr	1 0 0 1 1 1 0 1	← Saddr-offset →		
		A,sfr	0 0 0 0 0 0 0 1	1 0 0 1 1 1 0 1	Sfr-offset	
		A,[r4]	0 0 0 1 0 1 1 0	0 1 1 R_4 1 1 0 1		
		A,[HL]	0 0 0 1 0 1 1 0	0 1 0 1 1 1 0 1		
	CMP	A,#byte	1 0 1 0 1 1 1 1	← Data →		
		saddr,#byte	0 1 1 0 1 1 1 1	← Saddr-offset →	Data	
		sfr,#byte	0 0 0 0 0 0 0 1	0 1 1 0 1 1 1 1	Sfr-offset	Data
		r,r	1 0 0 0 1 1 1 1	0 $R_6R_5R_4$ 0 $R_2R_1R_0$		
		A,saddr	1 0 0 1 1 1 1 1	← Saddr-offset →		
		A,sfr	0 0 0 0 0 0 0 1	1 0 0 1 1 1 1 1	Sfr-offset	
		A,[r4]	0 0 0 1 0 1 1 0	0 1 1 R_4 1 1 1 1		
		A,[HL]	0 0 0 1 0 1 1 0	0 1 0 1 1 1 1 1		

Table C-4. uCOM-78K/I Instruction Codes (4/7)

Group	Mnemonic	Operand	Instruction Code			
			B 1	B 2	B 3	B 4
16-bit Arithmetic Operation	ADDW	AX,#word	0 0 1 0 1 1 0 1	← Low Byte →	High Byte	
		AX,rp	1 0 0 0 1 0 0 0	0 0 0 0 1 $P_2P_1$0		
		AX,saddrp	0 0 0 1 1 1 0 1	← Saddr-offset →		
		AX,sfrp	0 0 0 0 0 0 0 1	0 0 0 1 1 1 0 1	Sfr-offset	
	SUBW	AX,#word	0 0 1 0 1 1 1 0	← Low Byte →	High Byte	
		AX,rp	1 0 0 0 1 0 1 0	0 0 0 0 1 $P_2P_1$0		
		AX,saddrp	0 0 0 1 1 1 1 0	← Saddr-offset →		
		AX,sfrp	0 0 0 0 0 0 0 1	0 0 0 1 1 1 1 0	Sfr-offset	
	CMPW	AX,#word	0 0 1 0 1 1 1 1	← Low Byte →	High Byte	
		AX,rp	1 0 0 0 1 1 1 1	0 0 0 0 1 $P_2P_1$0		
		AX,saddrp	0 0 0 1 1 1 1 1	← Saddr-offset →		
		AX,sfrp	0 0 0 0 0 0 0 1	0 0 0 1 1 1 1 1	Sfr-offset	
Multiply/ Divide	MULUW	r	0 0 0 0 0 1 0 1	0 0 0 0 0 $R_2R_1R_0$		
	DIVUW	r	0 0 0 0 0 1 0 1	0 0 0 1 1 $R_2R_1R_0$		
Increment/Decrement	INC	r	1 1 0 0 0 $R_2R_1R_0$			
		saddr	0 0 1 0 0 1 1 0	← Saddr-offset →		
	DEC	r	1 1 0 0 1 $R_2R_1R_0$			
		saddr	0 0 1 0 0 1 1 1	← Saddr-offset →		
	INCW	rp	0 1 0 0 0 1 P_1P_0			
	DECW	rp	0 1 0 0 1 1 P_1P_0			
Shift/Rotate	ROR	r,n	0 0 1 1 0 0 0 0	0 1 N_2N_1 $N_0R_2R_1R_0$		
	ROL	r,n	0 0 0 1	0 1 N_2N_1 $N_0R_2R_1R_0$		
	RORC	r,n	0 0 0 0	0 0 N_2N_1 $N_0R_2R_1R_0$		
	ROLC	r,n	0 0 0 1	0 0 N_2N_1 $N_0R_2R_1R_0$		
	SHR	r,n	0 0 0 0	1 0 N_2N_1 $N_0R_2R_1R_0$		
	SHL	r,n	0 0 0 1	1 0 N_2N_1 $N_0R_2R_1R_0$		
	SHRW	rp,n	0 0 0 0	1 1 N_2N_1 $N_0P_2P_1$0		
	SHLW	rp,n	0 0 0 1	1 1 N_2N_1 $N_0P_2P_1$0		
	ROR4	[r4]	0 0 0 0 0 1 0 1	1 0 0 0 1 0 $R_1$1		
	ROL4	[r4]	0 0 0 0 0 1 0 1	1 0 0 1 1 0 $R_1$1		
BCD Adj	ADJBA		0 0 0 0 1 1 1 0			
	ADJBS		0 0 0 0 1 1 1 1			

Table C-4. uCOM-78K/I Instruction Codes (5/7)

Group	Mnemonic	Operand	Instruction Code			
			B 1	B 2	B 3	B 4
Bit Manipulation	MOV1	CY,saddr.bit	0 0 0 0 1 0 0 0	0 0 0 0 0 $B_2B_1B_0$	Saddr-offset	
		CY,sfr.bit	1 0 0 0	1 $B_2B_1B_0$	Sfr-offset	
		CY,A.bit	0 0 1 1	1 $B_2B_1B_0$		
		CY,X.bit	0 0 1 1	0 $B_2B_1B_0$		
		CY,PSW.bit	0 0 1 0	0 $B_2B_1B_0$		
		saddr.bit,CY	1 0 0 0	0 0 0 1 0 $B_2B_1B_0$	Saddr-offset	
		sfr.bit,CY	1 0 0 0	1 $B_2B_1B_0$	Sfr-offset	
		A.bit,CY	0 0 1 1	1 $B_2B_1B_0$		
		X.bit,CY	0 0 1 1	0 $B_2B_1B_0$		
		PSW.bit,CY	0 0 1 0	0 $B_2B_1B_0$		
	AND1	CY,saddr.bit	0 0 0 0 1 0 0 0	0 0 1 0 0 $B_2B_1B_0$	Saddr-offset	
		CY,/saddr.bit		0 0 1 1 0 $B_2B_1B_0$	Saddr-offset	
		CY,sfr.bit		0 0 1 0 1 $B_2B_1B_0$	Sfr-offset	
		CY,/sfr.bit		0 0 1 1 1 $B_2B_1B_0$	Sfr-offset	
		CY,A.bit	0 0 1 1	0 0 1 0 1 $B_2B_1B_0$		
		CY,/A.bit		0 0 1 1 1 $B_2B_1B_0$		
		CY,X.bit		0 0 1 0 0 $B_2B_1B_0$		
		CY,/X.bit		0 0 1 1 0 $B_2B_1B_0$		
		CY,PSW.bit	0 0 1 0	0 0 1 0 0 $B_2B_1B_0$		
		CY,/PSW.bit	0 0 1 0	0 0 1 1 0 $B_2B_1B_0$		
	OR1	CY,saddr.bit	0 0 0 0 1 0 0 0	0 1 0 0 0 $B_2B_1B_0$	Saddr-offset	
		CY,/saddr.bit		0 1 0 1 0 $B_2B_1B_0$	Saddr-offset	
		CY,sfr.bit		0 1 0 0 1 $B_2B_1B_0$	Sfr-offset	
		CY,/sfr.bit		0 1 0 1 1 $B_2B_1B_0$	Sfr-offset	
		CY,A.bit	0 0 1 1	0 1 0 0 1 $B_2B_1B_0$		
		CY,/A.bit		0 1 0 1 1 $B_2B_1B_0$		
		CY,X.bit		0 1 0 0 0 $B_2B_1B_0$		
		CY,/X.bit		0 1 0 1 0 $B_2B_1B_0$		
		CY,PSW.bit	0 0 1 0	0 1 0 0 0 $B_2B_1B_0$		
		CY,/PSW.bit	0 0 1 0	0 1 0 1 0 $B_2B_1B_0$		
	XOR1	CY,saddr.bit	0 0 0 0 1 0 0 0	0 1 1 0 0 $B_2B_1B_0$	Saddr-offset	
		CY,sfr.bit	1 0 0 0	1 $B_2B_1B_0$	Sfr-offset	
		CY,A.bit	0 0 1 1	1 $B_2B_1B_0$		
		CY,X.bit	0 0 1 1	0 $B_2B_1B_0$		
		CY,PSW.bit	0 0 1 0	0 $B_2B_1B_0$		

Table C-4. uCOM-78K/I Instruction Codes (6/7)

Group	Mnemonic	Operand	Instruction Code			
			B 1	B 2	B 3	B 4
Bit Manipulation	SET1	saddr.bit	1 0 1 1 0 $B_2B_1B_0$	← Saddr-offset →		
		sfr.bit	0 0 0 0 1 0 0 0	1 0 0 0 1 $B_2B_1B_0$	Sfr-offset	
		A.bit	0 0 1 1	1 $B_2B_1B_0$		
		X.bit	0 0 1 1	0 $B_2B_1B_0$		
		PSW.bit	0 0 1 0	0 $B_2B_1B_0$		
	CLR1	saddr.bit	1 0 1 0 0 $B_2B_1B_0$	← Saddr-offset →		
		sfr.bit	0 0 0 0 1 0 0 0	1 0 0 1 1 $B_2B_1B_0$	Sfr-offset	
		A.bit	0 0 1 1	1 $B_2B_1B_0$		
		X.bit	0 0 1 1	0 $B_2B_1B_0$		
		PSW.bit	0 0 1 0	0 $B_2B_1B_0$		
	NOT1	saddr.bit	0 0 0 0 1 0 0 0	0 1 1 1 0 $B_2B_1B_0$	Saddr-offset	
		sfr.bit	1 0 0 0	1 $B_2B_1B_0$	Sfr-offset	
		A.bit	0 0 1 1	1 $B_2B_1B_0$		
		X.bit	0 0 1 1	0 $B_2B_1B_0$		
		PSW.bit	0 0 1 0	0 $B_2B_1B_0$		
	SET1	CY	0 1 0 0 0 0 0 1			
	CLR1	CY	0 1 0 0 0 0 0 0			
	NOT1	CY	0 1 0 0 0 0 1 0			
Call/Return	CALL	!addr13	0 0 1 0 1 0 0 0	← Low Addr. →	High Addr.	
	CALLF	!addr11	1 0 0 1 0 ←	fa →		
	CALLT	[addr5]	1 1 1 ← ta →			
	RET		0 1 0 1 0 1 1 0			
	RETI		0 1 0 1 0 1 1 1			
Stack Manipulation	PUSH	rp	0 0 1 1 1 1 P_1P_0			
		PSW	0 1 0 0 1 0 0 1			
	POP	rp	0 0 1 1 0 1 P_1P_0			
		PSW	0 1 0 0 1 0 0 0			
	MOV	SP,#byte	0 0 1 0 1 0 1 1	1 1 1 1 1 1 0 0	Data	
		SP,A	0 0 0 1 0 0 1 0	1 1 1 1 1 1 0 0		
		A,SP	0 0 0 1 0 0 0 0	1 1 1 1 1 1 0 0		

Table C-4. uCOM-78K/I Instruction Codes (7/7)

Group	Mnemonic	Operand	Instruction Code B 1	B 2	B 3	B 4
Uncondi-tional BR	BR	!addr13	0 0 1 0 1 1 0 0	← Low Addr. →	High Addr.	
		rp	0 0 0 0 0 1 0 1	0 1 0 0 1 $P_2P_1$0		
		$addr13	0 0 0 1 0 1 0 0	← jdisp →		
Conditional Branch	BC BL	$addr13	1 0 0 0 0 0 1 1	← jdisp →		
	BNC BNL	$addr13	0 0 1 0	← jdisp →		
	BZ BE	$addr13	0 0 0 1	← jdisp →		
	BNZ BNE	$addr13	0 0 0 0	← jdisp →		
	BT	saddr.bit,$addr13	0 1 1 1 0 $B_2B_1B_0$	← Saddr-offset →	jdisp	
		sfr.bit,$addr13	0 0 0 0 1 0 0 0	1 0 1 1 1 $B_2B_1B_0$	Sfr-offset	jdisp
		A.bit,$addr13	0 0 1 1	1 $B_2B_1B_0$	jdisp	
		X.bit,$addr13	0 0 1 1	0 $B_2B_1B_0$	jdisp	
		PSW.bit,$addr13	0 0 1 0	0 $B_2B_1B_0$	jdisp	
	BF	saddr.bit,$addr13	0 0 0 0 1 0 0 0	1 0 1 0 0 $B_2B_1B_0$	Saddr-offset	jdisp
		sfr.bit,$addr13	1 0 0 0	1 $B_2B_1B_0$	Sfr-offset	jdisp
		A.bit,$addr13	0 0 1 1	1 $B_2B_1B_0$	jdisp	
		X.bit,$addr13	0 0 1 1	0 $B_2B_1B_0$	jdisp	
		PSW.bit,$addr13	0 0 1 0	0 $B_2B_1B_0$	jdisp	
	BTCLR	saddr.bit,$addr13	0 0 0 0 1 0 0 0	1 1 0 1 0 $B_2B_1B_0$	Saddr-offset	jdisp
		sfr.bit,$addr13	1 0 0 0	1 $B_2B_1B_0$	Sfr-offset	jdisp
		A.bit,$addr13	0 0 1 1	1 $B_2B_1B_0$	jdisp	
		X.bit,$addr13	0 0 1 1	0 $B_2B_1B_0$	jdisp	
		PSW.bit,$addr13	0 0 1 0	0 $B_2B_1B_0$	jdisp	
	DBNZ	r2,$addr13	0 0 1 1 0 0 1 R_0	← jdisp →		
		saddr,$addr13	0 0 1 1 1 0 1 1	← Saddr-offset →	jdisp	
CPU Control	MOV	STBC,#byte	0 0 0 0 1 0 0 1	1 1 0 0 0 0 0 0	$\overline{\text{Data}}$	Data
	SEL	RBn	0 0 0 0 0 1 0 1	1 0 1 0 1 0 N_1N_0		
	NOP		0 0 0 0 0 0 0 0			
	EI		0 1 0 0 1 0 1 1			
	DI		0 1 0 0 1 0 1 0			

APPENDIX D. MAXIMUM PERFORMANCE CHARACTERISTICS

(1) Maximum performance characteristics of Assembler

Item	Limit
Effective symbol length	6 characters
Number of characters per line	99 characters
Number of code segments per type	1 segment
Number of absolute segments	10 segments
Number of macrodefinitions	10 definitions

(2) Maximum performance characteristics of Linker

Item	Limit
Number of input modules files	100 files
Number of different segment names	255 segments
Number of absolute segments	100 segments

(3) Maximum performance characteristics of Locater

Item	Limit
Number of relocatable segments per input module	256 segments

(4) Restrictions on number of symbols

	No. of local symbols	No. of PUBLIC symbols
Assembler	Approx. 1,800 symbols	256 symbols
Linker	1,800 symbols x No. of modules	Approx. 2,000 symbols
Locater	1,800 symbols x No. of modules	Approx. 2,000 symbols

INDEX

Phase-out/Discontinued

EEU−1204
August 1988 P
Printed in Japan