OKI

SASM64K

User's Manual

Program Development Support Software

First Edition

February 1994

NOTICE

1. The information contained herein can change without notice owing to product and/or technical improvements. Before using the product, please make sure that the information being referred to is up-to-date.
2. The outline of action and examples for application circuits described herein have been chosen as an explanation for the standard action and performance of the product. When planning to use the product, please ensure that the external conditions are reflected in the actual circuit and assembly designs.
3. When developing and evaluating your product, please use our product below the specified maximum ratings and within the specified operating ranges including, but not limited to, operating voltage, power dissipation, and operating temperature.
4. **OKI assumes no responsibility or liability whatsoever for any failure or unusual or unexpected operation resulting from misuse, neglect, improper installation, repair, alteration or accident, improper handling, or unusual physical or electrical stress including, but not limited to, exposure to parameters beyond the specified maximum ratings or operation outside the specified operating range.**
5. Neither indemnity against nor license of a third party's industrial and intellectual property right, etc. is granted by us in connection with the use of product and/or the information and drawings contained herein. No responsibility is assumed by us for any infringement of a third party's right which may result from the use thereof.
6. The products listed in this document are intended only for use in development and evaluation of control programs for equipment and systems. These products are not authorized for other use (as an embedded device and a peripheral device).
7. Certain products in this document may need government approval before they can be exported to particular countries. The purchaser assumes the responsibility of determining the legality of export of these products and will take appropriate and necessary steps at their own expense for these.
8. No part of the contents contained herein may be reprinted or reproduced without our prior permission.
9. MS-DOS is a registered trademark of Microsoft Corporation.

PREFACE

This manual describes SASM64K, a structured assembler for the OLMS-64K series of 4-bit single-chip microcontrollers. The contents of this manual are written for developers with some experience using assembly language.

SASM64K operates on MS-DOS. It is supplied on floppy disk.

Typographic conventions

Symbol	Meaning
[ ]	Indicates that items enclosed in the brackets can be omitted.
n1	Indicates an constant expression that has a value of 0 or 1.
n2	Indicates an constant expression that has a value from 0 to 3.
n4	Indicates an constant expression that has a value from 0 to 15.
n8	Indicates an constant expression that has a value from 0 to 255.

TABLE OF CONTENTS

Chapter 1. Introduction

Chapter 2. Starting SASM64K

Chapter 3. Assembly Language Syntax

Chapter 4. Directives

Chapter 5. SASM Instructions

Chapter 6. SASM Instruction Details

Chapter 7. Control Statements

Chapter 8. Error Messages

Chapter 9. Output Files

Chapter 10. Sample Program

Appendices

Chapter 1

Introduction

This chapter describes the assembler’s functions.

1.1 Functional Overview

SASM64K is a structured assembler for the OLMS-64K series of 4-bit single-chip microcontrollers. A structured assembler provides a new language that combines the ease of coding of high-level languages with the high coding efficiency of assemblers. It raises the overall level of application program development efficiency.

SASM64K has the following features.

- **Upward compatibility with the ASM64K assembler**

  SASM64K is upward-compatible with the ASM64K assembler. Source programs previously created for ASM64K can also be assembled by SASM64K. In addition, programmers can introduce the new features of SASM64K a few at a time into their previous programming style, enabling an easy transition to SASM64K programming.

- **Extended instructions (SASM instructions) added**

  Extended instructions are macro instructions that further enhance object orientation of data. Each extended instruction is a combination of multiple CPU instructions (hereafter called basic instructions). For example, an extended instruction can express a memory-to-memory transfer with a single statement. Coding is also similar to high-level languages, making programs easier to read and understand.

- **Flow control blocks implemented**

  Programs can use control blocks for selection and iteration, such as the IF and WHILE statement blocks used in high-level languages. This allows development of structured programs, which are easy to maintain and update.

- **Preprocessor directives added**

  SASM64K adds preprocessor directives for macros and include files. Programmers can define proprietary macro instructions, and can include the files that define them.

SASM64K references the contents defined in a DCL file to perform assembler processing for each particular device.

A DCL file contains device-specific information. One is provided for each target device. the SASM64K assembler can be used for all devices in the OLMS-64K series by changing the DCL file.

Please refer to the following file provided with the assembler for the OLMS-64K series devices supported by SASM64K.

- README.JPN (Japanese version)
- README.ENG (English version)

The descriptions in this manual assume the following characteristics of the target device.

Program ROM size	0H to 7FFFH
Data RAM size	0H to 7FFH

The actual values will differ depending on the particular OLMS-64K series device. (This information is given in the DCL file provided with each device.)

SASM64K outputs four files: an object file, a print file, an error file, and an assembly source file.

The object file contains the machine language in Intel HEX format.

The print file lists the mnemonics alongside the machine language that they generate.

The error file consists of error messages and the source file statements that generated the errors. It will be output to the standard console if not specified otherwise.

The assembly source file contains the source code with preprocessor directives and extended instructions expanded. It can be assembled with ASM64K Ver. 1.01 or higher, as well as SASM64K itself.

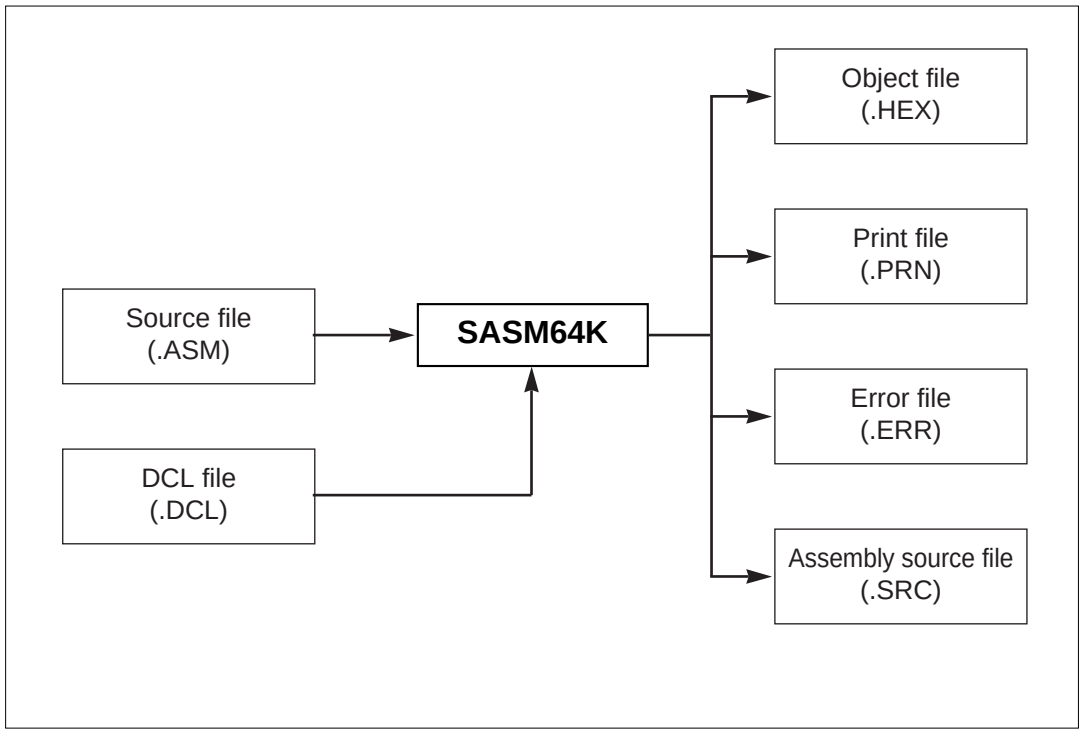


Figure 1-1. Input/Output Flow

1.2 Sample Program

Here is a small sample SASM64K program to start with.

```
 1:   TYPE(M64152)
 2:   TITLE "Sample program for sasm64k"
 3:
 4:   INCLUDE(SYMBOL.DEF)                ; File include
 5:
 6:   DEFINE RESET_DATA 0H               ; Macro definition by DEFINE
 7:
 8:   MACRO INC_BCD_2N()                 ; Macro definition
 9:     BA = 7
10:     [] == 9H
11:     A=1
12:     BA += BYTE[]
13:     BYTE[] = BA
14:   ENDM
15:
16:     ORG   100H
17:   START:
18:     HL = RESET_DATA
19:     WHILE(NOSKIP X == 0)            ; WHILE statement
20:       INC_BCD_2N()                  ; Macro call
21:       IF(SKIP BYTE[] == 19H)        ; IF statement
22:         [OPMODE_DATA]=4
23:         BREAK                       ; BREAK statement
24:       ENDI
25:     ENDW
26:     BCAL PLAY_SOUND_5
27:
28:   END
```

This program uses many SASM instructions instead of the CPU's basic instructions. The "HL=RESET_DATA" on line 18 is one of these. This instruction assigns the value RESET_DATA to the register pair HL. It corresponds to the basic instruction LHLI RESET_DATA. The SASM instructions are close to high-level language code, so programs can be more easily written and more easily read. Some of the SASM instructions expand into multiple basic instructions.

The definition of the symbol RESET_DATA with the DEFINE directive on line 6 will cause RESET_DATA to be replaced with the string "0H" each place it occurs in the source program. Another preprocessor directive is the INCLUDE directive on line 4. In this example, line 4 will expand to the contents of the file SYMBOL.DEF.

The macro definition on lines 8 to 14 and the macro call on line 20 are also preprocessor directives. In this example, the symbol INC_BCD_2N is defined as a macro. At the point it is called in line 20, it will be expanded to the contents of lines 9 to 13. Local labels can be declared that generate new label names each time macro arguments are expanded.

The WHILE statement block on lines 19 to 25 will cause the inner statements to be repeatedly executed until some condition is fulfilled. In this example, the condition is that register X not be equal to 0. Statements that control program flow like the WHILE statement are called control statements. An additional control statement is shown in the IF statement block on lines 21 to 24 in the sample program.

1.3 DCL Files

DCL files contain information in text format. SASM64K performs device-specific processing based on a DCL file. In other words SASM64K becomes an OLMS-64K device-specific assembler by replacing the DCL file for each particular device. The DCL file must be specified with a TYPE directive.

1.3.1 File Name

The assembler determine the DCL file name based on the device name specified in the TYPE directive.

DCL file name = *devicename*.DCL

You may change the base name of a DCL file to another name, but the ".DCL" extension cannot be changed.

1.3.2 DCL File Search

The assembler searches for the DCL file in the following sequence. Therefore the DCL file must be placed somewhere specified by one of these paths.

1. Search the path of the DCL file specification of the TYPE directive (normally the current directory).
2. Search the paths defined by the PATH environment variable. However, if the DCL file specification includes a path specification, then the search from the PATH environment variable will not be performed.

Examples are shown below. The target device for these examples is the MSM64152. Assume that the PATH environment variable has been defined as follows.

PATH = A:\BIN; A:\; DCL;

■ Example 1 ■

TYPE(M64152) is specified.

1. The assembler searches for M64152.DCL in the current directory.
2. If not found in (1), then the assembler will search for the following files in sequence, in accordance with the PATH environment variable.

A:\BIN\M64152.DCL
A:\M64152.DCL
DCL\M64152.DCL

■ Example 2 ■

TYPE(DCL\M64152) is specified.

1. The assembler searches for M64152.DCL in the DCL subdirectory of the current directory.
2. If not found in (1), then the search will terminate because the DCL file specification includes a path specification (DCL\).

1.3.3 DCL Instructions And File Declaration Contents

The DCL file declares memory address ranges, address symbols, and permitted instructions by using DCL instructions. The various DCL instructions are shown in Table 1-1.

Table 1-1. DCL Instructions

DCL Instruction	Declaration
#ROM	Declares program ROM area of target device.
#RAM	Declares data RAM area of target device.
#INSTRUCTION	Declares permitted instruction mnemonics of target device.
#SFR	Declares SFR area of target device.
#DEFSFR	Declares data address symbols of target device.
#ACCESS	Declares SFR area access attributes of target device.

Data address symbols are assigned to registers in the SFR area to indicate the addresses of the corresponding registers. Refer to the following file for data address symbol information for each target device.

- README.JPN (Japanese version)
- README.ENG (English version)

■ Note ■

Never attempt to rewrite the contents of a DCL file. Assembler results are not guaranteed if assembly is performed using a DCL file whose contents have been rewritten.

You should also use the most recent DCL file. When using SASM64K, always use the DCL files included in the SASM64K package.

1.3.4 Error Processing

The DCL file is processed in the assembler's first phase. If errors occur during this processing, then error messages will be displayed on the screen. However, if just one error occurs, then the assembler will abort without performing further processing.

If no errors occur, then assembly will continue.

Chapter 2

Starting SASM64K

This chapter explains how to start SASM64K.

2.1 Starting Methods

SASM64K can be started with either of two methods.

Method 1: SASM64K *file_name* [*options*]
Method 2: SASM64K

The *file_name* is the name of the file containing the source program to be assembled. The *options* specify listing control of the output file, etc. The start command, file name, and options are delimited by one or more spaces or tabs.

2.1.1 Starting Method 1

Type the following at the operating system prompt.

SASM64K *file_name* [*options*]

The assembler will load and then immediately start assembly.

2.1.2 Starting Method 2

Type the following at the operating system prompt.

SASM64K

The assembler will load and display a command prompt (*) on the console. Type the file name and options at the prompt as below.

file_name [*options*]

After this input the assembler will start assembly.

If nothing is input, then the assembler's usage will be output on the screen, and assembly will not be performed.

2.2 File Specifications

The file specification method described in this section applies to file specifications on the starting command line, in include directives, and in listing directives.

SASM64K allows files to be specified with the hierarchical directory structure that the operating system supports. In general the format of file specifications is as follows.

[*d*:] [\] [*directory_name*\] ... [*directory_name*\] *file_name* [.*extension*]

The length of a single file specification can be up to 50 characters. All characters that exceed this limit will be ignored.

The entire file name specification does not need to be specified; parts can be omitted. SASM64K determines defaults for drive names, directories, file names, and extensions for the different file types as shown in Table 2-1. When an item is omitted, the corresponding default shown in the table will be used.

Table 2-1. File Types And Defaults

File Type	Item			
	Drive	**Directory**	**File Name**	**Extension**
Source file	Current drive	Current directory	None	.ASM
Object file	Current drive	Current directory	Same as source file	.HEX
Print file	Current drive	Current directory	Same as source file	.PRN
Error file	Current drive	Current directory	Same as source file	.ERR
Assembly source file	Current drive	Current directory	Same as source file	.SRC

For example, if a file name with extension is specified for the source file when the assembler is started (Figure 2-1(1)), then SASM64K will recognize that as the file name. If no extension is added (Figure 2-1(2)), then SASM64K will assume an extension of ".ASM." If a period (.) is the last character of the file name (Figure 2-1(3)), then SASM64K will assume a file name with no extension.

The file specification for the listing directive is permitted up to the path name. In order to distinguish this from a file name, append a backslash (\) to the end of the specification string.

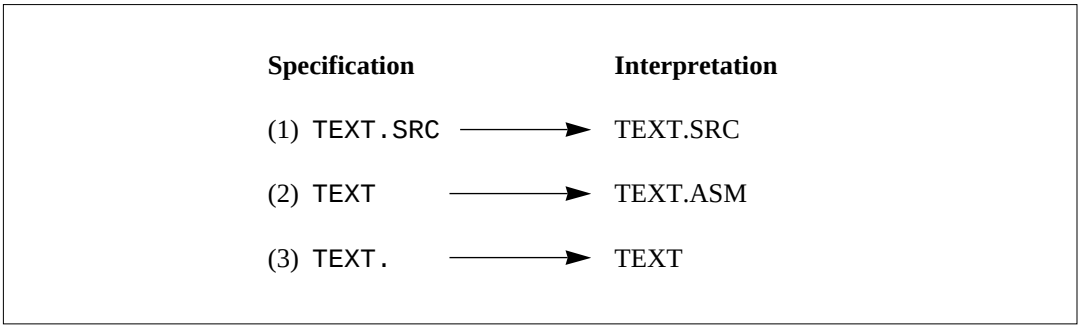


Figure 2-1. File Specifications And Interpretations

2.3 Options

Various specifications for controlling assembly functions can be specified when SASM64K is started. The options that can be specified are shown in Table 2-2. These options allow some of the listing directives described in section 4.5, “Listing Control,” to be input from the starting command line. The format differs, but the functions are identical to all the directives.

Options take precedence over directives. If SASM64K encounters a directive in a source program that tries to invalidate an option, then it will not output an error message for that directive.

Always specify options after the source file name. Some of the options specify an output file name. For the method of specifying a file name, refer to section 2.2, “File Specifications,” as well as the explanation for the corresponding directive.

As many options as needed can be specified in any order. When two or more options are specified, they must be delimited by one or more spaces or tabs.

Table 2-2. Options

Options	Function or corresponding directive
/O[(file)]	OBJ
/NO	NOOBJ
/PR[(file)]	PRN
/NPR	NOPRN
/S	SYM
/NS	NOSYM
/R	REF
/NR	NOREF
/E[(file)]	ERR
/NE	NOERR
/D	Output symbol debug information to the HEX file.
/ND	Do not output symbol debug information to the HEX file.
/A[(file)]	Output an assembly source file.
/NA	Do not output an assembly source file.

2.4 Termination Codes

When SASM64K terminates operation, it returns a value corresponding with the termination state (termination code) to the operating system.

Table 2-3. Termination Codes

Termination code	Termination state
0	No error.
1	An assembly error occurred.

2.5 Example Of Starting SASM64K

This section takes the assembly of a source file TEXT.ASM as an example. It shows the two methods previously described for starting the assembler.

This example will assemble the source file TEXT.ASM in the \USR\MYDIR directory, and will generate a print file and an error file. It will not generate a cross-reference list.

■ **Example 1 (Starting Method 1)**

```
A>SASM64K \USR\MYDIR\TEXT /PR /E /NR
SASM64K Structured Macro Assembler, Ver.1.00 Feb 1994
Copyright (C) 1994. Oki Electric Ind. Co.,Ltd.

pass1...
pass2...

Errors      :15
Warnings    :2
...Assembly End
```

■ **Example 2 (Starting Method 2)**

```
A>SASM64K
SASM64K Structured Macro Assembler, Ver.1.00 Feb 1994
Copyright (C) 1994. Oki Electric Ind. Co.,Ltd.
*\USR\MYDIR\TEXT /PR /E /NR

pass1...
pass2...

Errors      :15
Warnings    :2
...Assembly End
```

Chapter 3

Assembly Language Syntax

This chapter describes the rules of assembly language and the format of source programs.

3.1 Characters Allowed In Programs

The following characters can be used when coding a source program for SASM64K.

Letters:	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
Digits:	0 1 2 3 4 5 6 7 8 9
Symbols:	! ″ # $ % & Ô ( ) * + , -
	. / : ; < = > ? @ [ ] ^ \
	tab space _ (underscore) ˜

SASM64K allows lower case letters to be coded, but internal it converts them all to upper case before processing. Therefore, “TELEX” and “telex” will be handled as the same symbol.

3.2 Structual Elements Of Source Programs

An SASM64K source program is a collection of statements. There are several types of statements, but they all end with a carriage return. Please note that an end of file (EOF) cannot replace the carriage return. There are also statements that consist only of spaces, tabs, or carriage returns.

The maximum number of characters in a statement is 255, including the carriage return. Be sure not to exceed this limit.

Within a program, spaces and tabs are used to delimit between symbols, operators, etc. There are no special cautions here.

The statement types are described next.

3.2.1 Instruction Statements

SASM64K has two types of instructions. Instructions found in the CPU itself are called basic instructions. Instructions that SASM64K expands to one or more instructions when interpreting them during assembly are called SASM instructions. The beginning of all instructions can include a label definition. Instructions can also be converted to bit expressions by added a SKIP operator at the beginning. For details refer to section 3.3, “Statement Format Of Basic Instructions,” and chapter 5, “SASM Instructions.”

3.2.2 Directive Statements

Directive statements give the assembler itself various specifications. They include both preprocessor directives and assembler directives. For details refer to chapter 4, “Directives.”

3.2.3 Control Statements

Control statements (control blocks) control program flow. A control block is configured from two or more control statements. For details refer to chapter 7, “Control Statements.”

3.3 Statement Format Of Basic Instructions

Basic instruction statements are configured from four fields (label definition, instruction, operand, and comment). They are generally coded as follows.

LABEL:	LMA	@XY	;comment
label definition	instruction	operand	comment

Actual statements do not necessarily need to fill all four fields. Only required fields will be filled. Each field can start from any column, but the fields cannot be entered out of order.

Each field is defined below.

3.3.1 Label Field

The label field codes a symbol that will be defined as the current address. A colon (:) must always be placed after the symbol.

This symbol can be referenced anywhere else in the program.

3.3.2 Instruction Field And Operand Field

The instruction field and operand field code the basic instruction. When an operand follows the instruction, they are delimited by spaces or tabs. Two more instructions cannot be coded in one statement.

3.3.3 Comment Field

The comment field starts with a semicolon (;) and ends with the carriage return. The contents of a comment field have no effect on the assembly results.

3.4 Symbols

Symbols represent numbers, addresses, and registers. They are broadly divided into reserved word symbols and user defined symbols. Reserved word symbols are defined by the assembler, and user defined symbols are defined within the program by the user. Programs can be developed more effectively by using symbols properly.

3.4.1 Reserved Word Symbols

SASM64K has pre-defined the basic instructions, special assembler symbols, and data address symbols given in the appendix as reserved words. The user cannot define these reserved words within a program.

Reserved words can be used only as specifically designated. In other words, the assembler does not permit reserved words to be labels in programs or to be redefined with symbol directives.

The types of reserved words are described below.

(1) Basic instructions

Mnemonics that represent basic instructions are all reserved words.

(2) Directives

Directives for controlling the assembler are all reserved words.

(3) Special assembler symbols

Special assembler symbols are reserved words provided for OLMS-64K series internal registers, and are special reserved words used with certain instruction operands and SASM instructions.

(4) Data address symbols

Data address symbols are assigned to registers in the SFR area. Each represents the address of its corresponding register. For details refer to the following file.

- README.JPN (Japanese version)
- README.ENG (English version)

(5) Symbols that begin with a question mark (?)

SASM64K uses symbols that begin with a question mark internally, so they are reserved words.

3.4.2 User Defined Symbols

New symbols can be defined in a program as labels or with symbol definition directives. Symbols defined in a program by the user in this manner are called user defined symbols.

User defined symbols are given their values when defined.

The following characters can be used in user defined symbols.

A to Z, a to z, 0 to 9, $, ?, _

However, in order to distinguish user defined symbols from integer constants, the first character must not be a numeric digit.

There is no restriction on the number of characters coded for a symbol, but only the first 31 characters are valid. All characters past this will be ignored.

Correct examples	**Incorrect examples**	
_LOOP:	DATA:	...same as reserved word
LOOP_1:	1ABC:	...starts with a numeric digit
$XYZ:		

When a user defined symbol is defined in a program (except when defined with a preprocessor directive), it will be given a value and segment type.

Segments type will be either CSEG or DSEG, depending on which address space the label or name was defined in. However, names defined with the EQU and SET directives will not be given a segment type.

Labels are defined as addresses. They must be followed by a semicolon (:) when defined. A label will be assigned the segment type and the segment address value that defines the label.

Names are defined with symbol definition directives, to be described later.

The basic instructions, directives, special assembler symbols, and data address symbols listed in the appendix cannot be used as user defined symbols.

3.4.3 Location Symbol

The dollar sign ($) can be used as a symbol representing the value of the location counter. When coded in a normal operand, it represents the address at which the instruction is located.

■ **Example** ■

Location	Code	Line No.	Instruction	Operand	Operand Value
		1	TYPE	(M64162)	
0000	40	2	LAL		;
0001	BF 70	3	LAM	@XY	;
0003	52	4	XAB		;
0004	A8 06	5	JP	$+2	;06
0006	06	6	DB	$	;06
		7			

3.4.4 Symbol Scope And Overlapping Definitions

In general a symbol must be defined before it is referenced. However, labels can be defined after being referenced (forward references are possible).

A symbol can be defined only once within a single file, but it can be redefined with the SET directive. A symbol defined with the DEFINE directive can be redefined as long as it completely matches the string to expand.

Parameters and local symbols defined within a macro definition are valid only with that macro. Even if an identical name is defined within another macro, it will be handled as a different symbol.

3.5 Constants

3.5.1 Integer Constants

SASM64K handles terms that start with a digit 0 to 9 as integer constants.

The allowed range of values is 0 to 65535 (0FFFFH). Values above that limit will cause an error.

Binary, octal, decimal, and hexadecimal numeric expressions are permitted as integer constants. In order to distinguish between these expression types, a type identifier is appended to the value as shown in Table 3-1. Also, when a hexadecimal expression would start with an alphabetic character (A to F), it is prefixed with the digit 0 in order to distinguish it from a symbol.

Also, to enhance program readability, underscores (_) may be coded anyplace within numeric expression strings. However, if added at the start of the string, then it will be handled as a symbol.

Table 3-1. Integer Constant Expressions

Number Type	Permitted Characters	Type Identifier	Examples
Binary	0, 1, _	B	1010B, 0100_1101B
Octal	0 to 7, _	O, Q	271O, 514Q
Decimal	0 to 9, _	D	1263, 401D
Hexadecimal	0 to 9, A to F, _	H	753H, 0C67EH

3.5.2 Character Constants

A character constant is one ASCII character equivalent enclosed in single quotation marks (‘). The reason we say “one ASCII character equivalent” is that the codes below are also permitted.

(1) An octal number with up to 3 digits (up to 377Q=0FFH) can be coded after a backslash.

‘\0’ Evaluates to the number 0.
‘\123’ Evaluates to the number 123Q.

(2) Any character other than ‘0’ to ’7’ after a backslash represents the character itself.

‘\A’ Evaluates to the character A.
‘\#’ Evaluates to the character #.

(3) A single quotation mark itself is coded as ‘\’’ using method (2) above.

3.5.3 String Constants

A string constant is a string of displayable characters enclosed in double quotation marks (“). To represent non-displayable characters, use an escape sequence.

String constants are only used in DB directives. For details refer to section 4.4, “Data Definition.”

3.6 Operators

Arithmetic expressions (constant expressions) that use the previously described integer constants, character constants, user defined symbols, and location symbol can be coded as instruction operands. SASM64K expresses numbers internally as 32-bit unsigned integers, and all operations are performed as 32-bit unsigned operations. Overflows that result from operations will be ignored, so pay special attention when using relational operators.

3.6.1 Arithmetic Operators

Operator	Function
+	Addition and unary operator.
-	Subtraction and unary operator.
*	Multiplication
/	Division
%	Modulo calculation. Returns the remainder dividing the left term by the right term.

■ Example ■

Arithmetic expressions and their values are shown below.

Arithmetic Expression	Value
1234H+80H	12B4H
1234H–80H	11B4H
1234H*80H	91A00H
1234H/80H	24H
1234H%80H	34H

3.6.2 Logical Operators

The result of a logical operation is always either 0 (false) or 1 (true).

Operator	Function
&&	Returns 1 if both the left term and right term are not 0; otherwise returns 0.
||	Returns 0 if both the left term and right term are 0; otherwise returns 1.
!	Returns 1 if the right term is 0; otherwise returns 0.

■ Example ■

Logical expressions and their values are shown below.

Logical Expression	Value
5588H&&0H	0
5588H || 0H	1
!5588H	0

3.6.3 Bitwise Logical Operators

Bitwise logical operators operate on each bit of their operand terms.

Operator	Function
&	Logical AND of left term and right term.
|	Logical OR of left term and right term.
^	Exclusive OR of left term and right term.
>>	Shift left term to the right by the number of bits specified by the right term.
<<	Shift left term to the left by the number of bits specified by the right term.
~	Bit complement of right term.

■ Example ■

Bitwise logical expressions and their values are shown below.

Bitwise Logical Expression	Value
1234H&4321H	0220H
1234H | 4321H	5335H
1234H^4321H	5115H
1234H<<1	2468H
1234H>>1	091AH
~1234H	0EDCBH

3.6.4 Relational Operators

The result of a relational operation is always either 0 (false) or 1 (true).

Operator	Function
>	Returns 1 if left term is greater than right term; otherwise returns 0.
>=	Returns 1 if left term is greater than or equal to the right term; otherwise returns 0.
<	Returns 1 if left term is less than right term; otherwise returns 0.
<=	Returns 1 if left term is less than or equal to right term; otherwise returns 0.
==	Returns 1 if left term is equal to right term; otherwise returns 0.
!=	Returns 1 if left term is not equal to right term; otherwise returns 0.

■ Example ■

Relational expressions and their values are shown below.

Relational Expression	Value
1234H>1234H	0
1234H>=1234H	1
1234H<1234H	0
1234H<=1234H	1
1234H==1234H	1
1234H!=1234H	0

3.6.5 Other Operators

Operator	Function
LOW	Returns value of low 4 bits of an 8-bit constant.
HIGH	Returns value of high 4 bits of an 8-bit constant.
OFFSET	Returns value of low 8 bits of a 16-bit constant.
BNK	Returns value of high 8 bits of a 16-bit constant.

■ Example ■

Expressions and their values are shown below.

Expression	Value
LOW 12H	2H
HIGH 12H	1H
OFFSET 1234H	34H
BNK 1234H	12H

3.6.6 Precedence

Table 3-2 shows operator precedence. The highest precedence is 1, with progressively lower precedences following in order. Operators on the same line have the same precedence. Operators are evaluated from highest to lowest precedence. Operators with the same precedence are evaluated in their order of appearance from the left of the expression.

Table 3-2. Operator Precedence

Precedence	Operator
1	( )
2	.
3	! ~ – (unary) LOW HIGH OFFSET BNK
4	* / %
5	+ –
6	<< >>
7	< <= > >=
8	== !=
9	&
10	^
11	|
12	&&
13	||

3.7 Comments

Comments have no effect on programs, so the programmer can freely code them throughout. The format is to start with a semicolon (;), and follow with the comment itself.

■ **Example** ■

```
        LAB                 ;ACC<-B
        JP    LOOP          ;GOTO LOOP
;--------------------------------
;   SUB-PROGRAM
;--------------------------------
        ORG   100H
```

As shown above, statements can be coded with comments after instructions and operands, and they can be coded to consist only of comments. Any displayable character can be coded in a comment, not just the letters, digits, and symbols given in section 3.1, “Characters Allowed In Programs.”

3.8 Address Space And Segment Types

The OLMS-64K series has two address spaces (code address space and internal data address space) that it can process independently.

SASM64K gives each symbol defined as an address in these address spaces a segment type as an attribute. SASM64K does not perform segment type checks (checking whether or not operands are of the expected type). However, by clearly declaring which address space each symbol resides in, you can make programs easier to read and can avoid mixing symbols between different segments.

Table 3-3 shows the relationship of address spaces and segment types.

Table 3-3

Address Space	Contents	Segment Type
Code space	0 to 7FFFH x 8 bits	CSEG
Internal data space	0 to 7FFH x 4 bits	DSEG

The actual size of each address space differs for the various devices in the OLMS-64K series. This information is given in the DCL file provided for each device.

3.9 Data RAM Address Specifications

There are four modes to specify in operands how data is to be accessed (addressing).

(1) HL indirect addressing mode
(2) XY indirect addressing mode
(3) Direct addressing mode
(4) Stack pointer indirect addressing mode

Each addressing mode is valid with a subset of the instructions.

The bank is specified with the low 3 bits of either Bank Select Register 0 (BSR0) or Bank Select Register 1 (BSR1).

3.9.1 HL Indirect Addressing Mode

If nothing is coded in the operand, the entire area of RAM can be addressed by the combination of the HL register and the bank specified by the low 3 bits of BSR0.

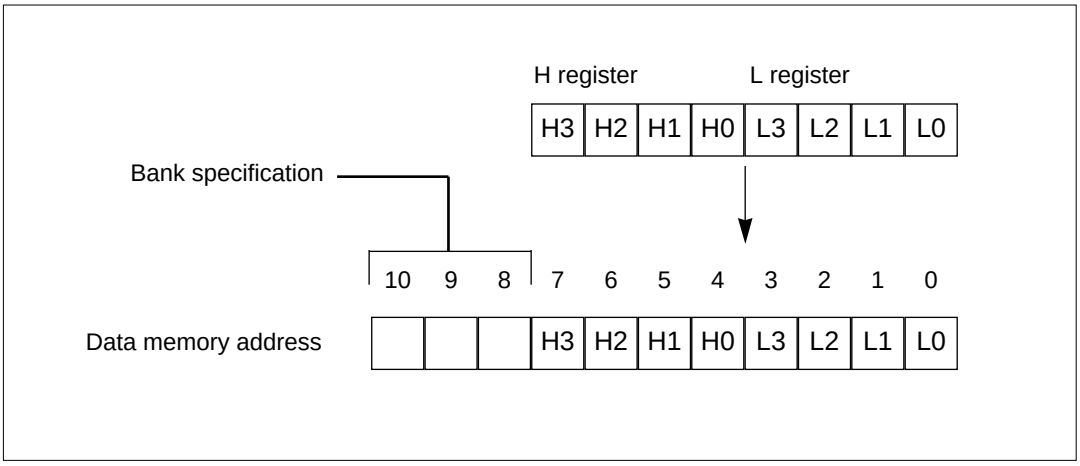


Figure 3-1. Data Memory Address For HL Indirect Addressing

■ Code Example ■

```
LBS0I     7H          ; Sets BSR0 to 7.
LHLI      3FH         ; Loads 3FH in the HL register.
INM                   ; Increments the data at RAM address 73FH.
```

3.9.2 XY Indirect Addressing Mode

If "@XY" is coded in the operand, the entire area of RAM can be addressed by the combination of the XY register and a bank select register. When BEF (bit 3 of BSR1) is 0, the bank will be specified by the low 3 bits of BSR0. When BEF is 1, the bank will be specified by the low 3 bits of BSR1.

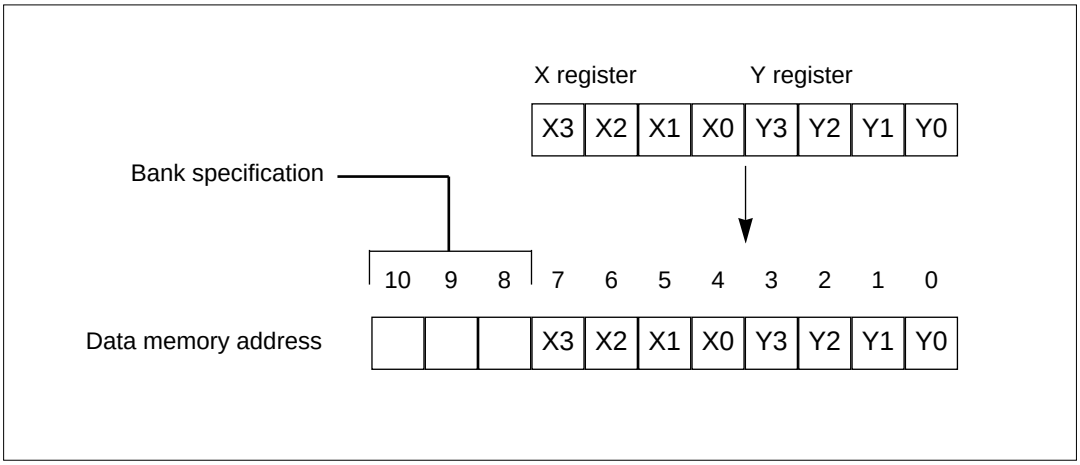


Figure 3-2. Data Memory Address For XY Indirect Addressing

■ Code Example ■

```
LBS1I    7H      ; Sets BSR1 to 7.
SBE              ; Sets the bank enable flag (BEF).
RBC              ; Resets the bank common flag (BCF).
LXYI     3FH     ; Loads 3FH in the XY register.
INM      @XY     ; Increments the data at RAM address 73FH.
```

3.9.3 Direct Addressing Mode

If an 8-bit (1-byte) address value is coded in the operand, the entire area of RAM can be addressed by the combination of the address value and a bank select register. When BEF (bit 3 of BSR1) is 0, the bank will be specified by the low 3 bits of BSR0. When BEF is 1, the bank will be specified by the low 3 bits of BSR1. However, when BCF is 1 and the address within the bank is 0~7FH, addressing will be in bank 0.

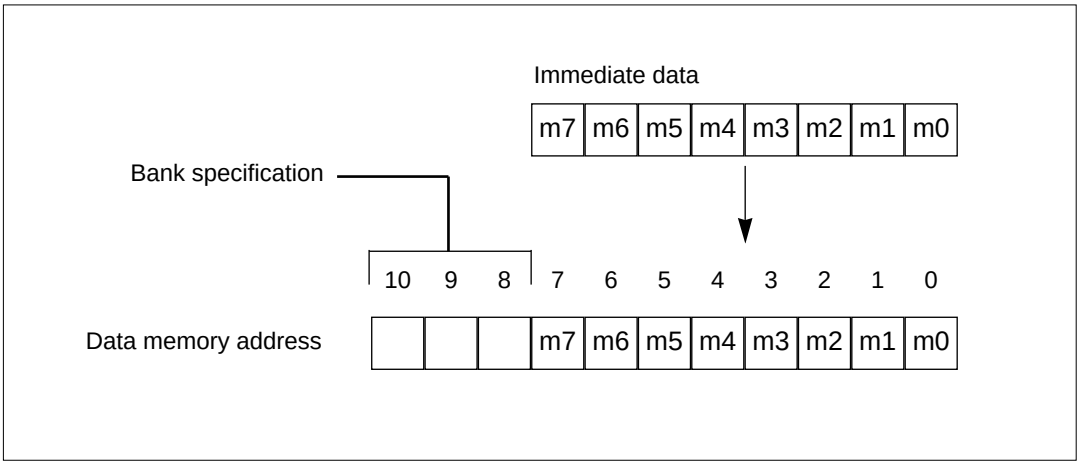


Figure 3-3. Data Memory Address For Direct Addressing

■ Code Example ■

```
LBS0I     7H            ; Sets BSR0 to 7.
RBE                     ; Resets the bank enable flag (BEF).
RBC                     ; Resets the bank common flag (BCF).
INMD3FH           ; Increments the data at RAM address 3FH.
```

3.9.4 Stack Pointer Indirect Addressing Mode

In this mode (execution of PUSH and POP instructions) the address will be that in bank 7 given by the stack pointer, regardless of the bank select registers (BSR0, BSR1).

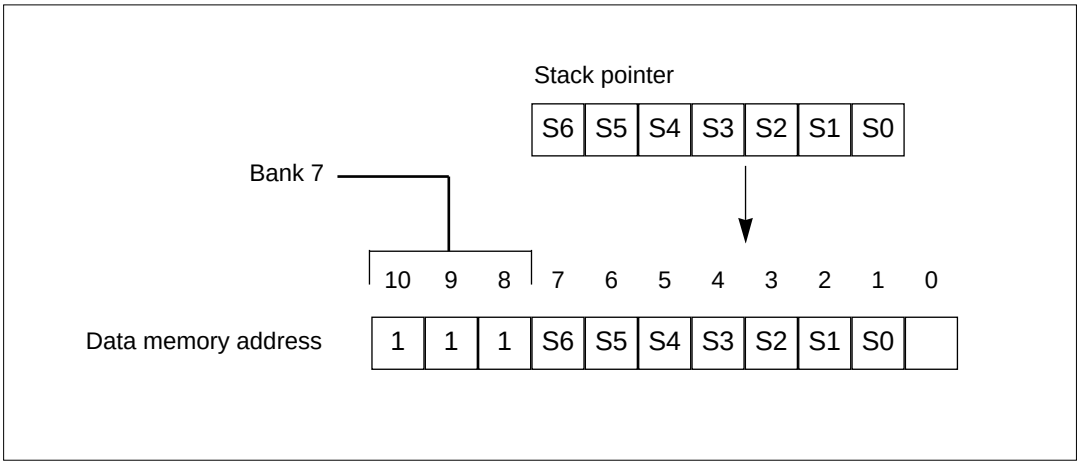


Figure 3-4. Data Memory Address For Stack Pointer Indirect Addressing

■ Code Example ■

```
PUSH    HL          ; Pushes the contents of the HL register into the RAM address
                    ; in bank 7 specified by the stack pointer.
```

Chapter 4

Directives

This chapter explains directives. Directives are provides for controlling assembler states, so except for the DB, DW, B, and BCAL directives, they do not generate code.

4.1 Symbol Definitions

Symbol definition directives enable the user to define symbols that represent numeric or address values. Defined symbols can be referenced anywhere in a program.

4.1.1 EQU

■ **Syntax** ■

```
symbol  EQU  constant_expression
symbol  =  constant_expression
```

■ **Function** ■

The EQU directive assigns the value of a constant expression to a symbol.

A symbol defined with EQU is not permitted to be redefined within the same program as a label or new symbol. It will also not be given a segment type.

The constant expression cannot include forward references. Its result must evaluate within the range 0 to 0FFFFH.

4.1.2 SET

■ **Syntax** ■

```
symbol  SET  constant_expression
```

■ **Function** ■

The SET directive has the same function as the EQU directive. However, when the SET directive is used, the same symbol can be redefined any number of times with additional SET directives. The symbol will be assigned the value of the expression of the current SET directive. Accordingly, an EQU directive assigns a single value to a symbol, but the value assigned by a SET directive will be valid until changed by the next SET directive.

4.1.3 DATA

■ **Syntax** ■

```
symbol  DATA  constant_expression
```

■ **Function** ■

The DATA directive assigns a RAM data address to a symbol.

The expression cannot include forward references. Its result must evaluate within the range 0 to 7FFH (the 7FFH maximum value will differ depending on the device). The symbol will be assigned the DSEG segment type.

Symbols defined with the DATA directive cannot be redefined.

4.1.4 CODE

■ **Syntax** ■

```
symbol  CODE  constant_expression
```

■ **Function** ■

The CODE directive assigns a code address to a symbol.

The expression cannot include forward references. Its result must evaluate within the range 0 to 7FFFH (the 7FFFH maximum value will differ depending on the device). The symbol will be assigned the CSEG segment type.

Symbols defined with the CODE directive cannot be redefined.

4.1.5 Code Example

```
ABC     EQU     8H
VAL1    EQU     ABC+1
VAL2    EQU     ABC+2
FLAG    SET     1

DADR    DATA    30H             ; Assigns data address 30H to DADR.
CADR    CODE    250H            ; Assigns code address 250H to CADR.

        LAI     ABC
        LAI     FLAG            ; Value of FLAG is 1.
FLAG    SET     2
        LAI     FLAG            ; Value of FLAG is 2.
```

4.2 Memory Segment Control

Memory segment control directives define the start of segments (address spaces). There are two segments: the code segment and data segment. Each segment has its own location counter. The location counter values correspond one-for-one with addresses in their respective segments.

The default segment when the assembler starts is CSEG.

4.2.1 CSEG

■ **Syntax** ■

```
CSEG
```

■ **Function** ■

The CSEG directive defines the start of the code address segment. The assembler starts out in the code segment. When CSEG is first defined, the location counter value will become 0. The range of values of the code segment's location counter is 0 to 7FFFH (the 7FFFH value will differ depending on the device). The location counter will be modified by the ORG, DS, DB, DW, and NSE directives, as well as instructions in machine language.

If CSEG is defined in a second or later place, then its location counter value will begin from the last location counter value in the previous CSEG.

4.2.2 DSEG

■ **Syntax** ■

```
DSEG
```

■ **Function** ■

The DSEG directive defines the start of the data address segment. In this segment, symbols for RAM addresses are defined and areas are reserved. The DSEG directive can be used any number of times in a program. The range of values of the data segment's location counter is 0 to 7FFH (the 7FFH value will differ depending on the device) corresponding to the RAM address. The location counter will be modified by the ORG and DS directives. When DSEG is first defined, the location counter value will become 0.

If DSEG is defined in a second or later place, then its location counter value will begin from the last location counter value in the previous DSEG.

4.2.3 Code Example

```
        DSEG                    ; Start data segment.
        ORG     10H
DT1:    DS      10
DT2:    DS      5

        CSEG                    ; Start code segment.
        ORG     100H
        DCM
        DW      123
```

4.3 Location Counter Control

The location counter control directives modify the values of the location counters of each address space. The location counter control directives are ORG, DS, and NSE.

4.3.1 ORG

■ **Syntax** ■

```
ORG  constant_expression
```

■ **Function** ■

The ORG directive sets the location to begin storing the program or data. It can be used in either segment. When the assembler encounters an ORG directive, it evaluates the constant expression and stores the result in the location counter.

The location counters are managed by the assembler itself. They correspond to addresses in the different address segments. The ORG directive may be used any number of times in one program.

The constant expression cannot include forward references or labels. Its result must evaluate within the permitted range for the appropriate address segment.

4.3.2 DS

■ **Syntax** ■

```
[label:]  DS  constant_expression
```

■ **Function** ■

The DS directive reserves a memory area with undefined contents. Its size will be the number of bytes (for CSEG) or nibbles (for DSEG) indicated by the constant expression.

The DS directive simply adds the value of the constant expression to the location counter value of the current segment type. However, the addresses restrictions for the segment cannot be exceeded.

The constant expression cannot include forward references or labels.

4.3.3 NSE

■ Syntax ■

```
NSE
```

■ Function ■

The NSE directive modifies the location to the next 16-byte boundary.

4.3.4 Code Example

The following example shows examples of statements and the location counter value (in hexadecimal).

Location	
	DSEG
0000	DD1: DS 10H
0010	DD2: DS 10H
0020	ORG 100H
0100	DD3: DS 50H
0150	
	CSEG
0000	ORG 130H
0130	JP 140H
0132	NSE
0140	NOP

4.4 Data Definitions

Data definition directives initialize code memory in one or two-byte units.

4.4.1 DB

■ **Syntax** ■

```
[label:]  DB  constant_expression_list
```

■ **Function** ■

The DB directive defines the contents of code memory in 1-byte units. Therefore it can only be used in the code segment. A string expression can be used as the constant expression list. Data is allocated to memory in the order it listed starting from the current program address.

If the location symbol ($) is specified, then the assembler will recognize it as the code address value of the memory to be defined.

4.4.2 DW

■ **Syntax** ■

```
[label:]  DW  constant_expression_list
```

■ **Function** ■

The DW directive defines the contents of code memory in 2-byte units. Therefore it can only be used in the code segment. A string expression cannot be used as the constant expression list. Data is allocated to memory in the order it is listed starting from the current program address, low byte followed by high byte.

If the location symbol ($) is specified, then the assembler will recognize it as the code address value of the memory to be defined.

4.4.3 Code Example

```
        DB      12H,34H         ; Allocates in order 12, 34.
TBL:    DB      "AB","CD"       ; Allocates in order 41, 42, 43, 44.

        DW      1234H           ; Allocates in order 34, 12.
        DW      12H,34H         ; Allocates in order 12, 00, 34, 00.
```

4.5 Listing Control

Listing control directives affect the output format of the assembly listing (print file). They have absolutely no effect on the code generated as the result of assembly.

4.5.1 DATE

■ **Syntax** ■

```
DATE  "string of up to 255 characters"
```

■ **Function** ■

The DATE directive assigns the date to be inserted in the print file header.

If no date is specified with a DATE directive, then the date of assembly obtained from the operating system will be assigned. The string is enclosed in double quotation marks (“). Up to 255 characters are valid in the specified string. If the string exceeds 255 characters, then the excess will be ignored.

If the DATE directive is specified two or more times, then the assembler will recognize the last specified one as effective.

4.5.2 TITLE

■ **Syntax** ■

```
TITLE  "string of up to 255 characters"
```

■ **Function** ■

The TITLE directive assigns a title to be inserted in the print file header.

If no title is specified with a TITLE directive, then the title will be blank. The string is enclosed in double quotation marks (“). Up to 255 characters are valid in the specified string. If the string exceeds 255 characters, then the excess will be ignored.

If the TITLE directive is specified two or more times, then the assembler will recognize the last specified one as effective.

4.5.3 PAGE

■ **Syntax** ■

```
PAGE  [rows] [,columns]
```

■ **Function** ■

The PAGE directive specifies a new page in the print file. The row and column numbers are provided only for compatibility with ASM64K, and are ignored by SASM64K.

4.5.4 OBJ/NOOBJ

■ **Syntax** ■

```
OBJ [(file_name)]
NOOBJ
```

■ **Function** ■

These directives specify the generation and output of the object file.

The NOOBJ directive specifies that no object file is to be generated.

The OBJ directive specifies that an object file is to be generated, and that it is to be output to the file specified by *file_name*. If *file_name* is omitted, then the assembler will use the default output specification. Refer to section 2.2, “File Specifications,” for default file names.

The default directive is OBJ.

4.5.5 PRN/NOPRN

■ **Syntax** ■

```
PRN [(file_name)]
NOPRN
```

■ **Function** ■

These directives specify the generation and output of the print file.

The NOPRN directive specifies that no print file is to be generated.

The PRN directive specifies that a print file is to be generated, and that it is to be output to the file specified by *file_name*. If *file_name* is omitted, then the assembler will use the default output specification. Refer to section 2.2, “File Specifications,” for default file names.

The default directive is NOPRN.

4.5.6 ERR/NOERR

■ **Syntax** ■

```
ERR [(file_name)]
NOERR
```

■ **Function** ■

These directives would specify the generation and output of the error file. However, they are provided only for compatibility with ASM64K, and are ignored by SASM64K. To output an error file, specify the /E option when you start the assembler.

4.5.7 SYM/NOSYM

■ **Syntax** ■

```
SYM
NOSYM
```

■ **Function** ■

These directives specify whether or not a symbol list is to be output to the print file.

The NOSYM directive specifies to not output a symbol list to the print file.

The SYM directive specifies to output a symbol list to the print file. However, if NOPRN has been specified, then SYM will be ineffective.

The default directive is NOSYM.

4.5.8 REF/NOREF

■ **Syntax** ■

```
REF
NOREF
```

■ **Function** ■

These directives specify whether or not a cross-reference list is to be output to the print file.

The NOREF directive specifies to not output a cross-reference list to the print file.

The REF directive specifies to output a cross-reference list to the print file. However, if NOPRN has been specified, then REF will be ineffective.

The default directive is NOREF.

4.5.9 LIST/NOLIST

■ **Syntax** ■

```
LIST
NOLIST
```

■ **Function** ■

The LIST directive specifies that following statements are to be listed. The NOLIST directive specifies that the listing of following statements is to be suppressed. Each of these is effective until its counterpart is next specified. However, if NOPRN has been specified, then LIST will be ineffective.

The default when the assembler starts is LIST.

4.5.10 Code Example

```
TYPE     (M64162)

TITLE    "Sample Program"
DATE     "Sep. 10th 92"

PAGE     60,80

OBJ
PRN(SAMPL3.LST)
SYM

         :
NOLIST
LAI5             ; These two lines are not output
LMA              ; to the assembly list.
LIST
         :
```

4.6 Optimization

4.6.1 B

■ **Syntax** ■

```
B  symbol (label)
```

■ **Function** ■

The B (branch) directive is provided to make more efficient use of jump instructions. Depending on the symbol (label) indicated as a branch destination, the B directive will convert between JCP (1-byte instruction), JP (2-byte instruction), and LJP (3-byte instruction).

When the branch destination is within a 64-byte boundary, the B directive will convert to a JCP instruction. When the branch destination is within a 2K-byte boundary, it will convert to a JP instruction. When the branch destination is in the entire ROM area, it will convert to a LJP instruction.

Only labels are permitted as operands. Calculation expressions that use labels are not permitted.

4.6.2 GOTO

■ **Syntax** ■

```
GOTO  symbol (label)
```

■ **Function** ■

The GOTO directive is the same as the B directive. It can be used instead of the B directive.

4.6.3 BCAL

■ **Syntax** ■

```
BCAL  symbol (label)
```

■ **Function** ■

The BCAL (branch call) directive is provided to make more efficient use of call instructions. Depending on the symbol (label) indicated as a call destination, the BCAL directive will convert between CAL (2-byte instruction) and LCAL (3-byte instruction).

When the branch destination is within a 2K-byte boundary, the BCAL directive will convert to a CAL instruction. When the branch destination is in the entire ROM area, it will convert to a LCAL instruction.

Only labels are permitted as operands. Calculation expressions that use labels are not permitted.

4.6.4 Code Example

The following example shows examples of statements and the location counter value (in hexadecimal).

Location				
0000	LOOP0:			
:			:	
:			:	
0700	LOOP1:			
:			:	
:			:	
073F		B	LOOP1	; Converts to JCP instruction.
:			:	
:			:	
07FF	SUB0:	B	LOOP1	; Converts to LJP instruction.
0801		B	LOOP0	; Converts to LJP instruction.
0804				
:			:	
:			:	
0900		BCAL	SUB1	; Converts to CAL instruction.
0902		BCAL	SUB0	; Converts to LCAL instruction.
:			:	
:			:	
0910	SUB1:			

4.7 Assembler Control

4.7.1 TYPE

■ **Syntax** ■

```
TYPE (DCL_file_name)
```

■ **Function** ■

The TYPE directive specifies the DCL file name for the target device of assembly.

The assembler reads the DCL file specified by the TYPE directive and performs initial assembly settings based on the device features. Therefore, a TYPE directive must be coded at the start of a program.

A path name can be specified for the file specification. In such cases the assembler will not refer to the PATH environment variable.

An extension can be specified for the file specification, but DCL file extensions are restricted to ".DCL."

A TYPE directive must be coded before all instructions and before the following directives.

EQU SET CODE DATA DB DW DS ORG NSE B BCAL DEFINE MACRO

If the TYPE directive violates any of the above rules, then assembly will not be performed.

Refer to section 1.3, "DCL Files," for details on DCL files.

4.7.2 END

■ **Syntax** ■

```
END
```

■ **Function** ■

The END directive is the statement that indicates the end of the program. The assembler will assemble until the END directive.

Labels and operands cannot be coded with the END directive.

4.8 Preprocessor Directives

4.8.1 INCLUDE

■ **Syntax** ■

```
INCLUDE (file_name)
```

■ **Function** ■

The INCLUDE directive expands to the file specified by *file_name* at the position where the directive is coded. No extension is assumed for *file_name*, so the extension must be coded if one exists. If not path is specified for *file_name*, then the assembler will search the current directory.

Files expanded with the INCLUDE directive may themselves contain INCLUDE directives.

■ **Example** ■

```
INCLUDE(macro.lib)
```

4.8.2 DEFINE

■ **Syntax** ■

```
DEFINE  symbol  text_string
```

■ **Function** ■

The DEFINE directive assigns a text string to a symbol. Whenever the assembler encounters the symbol in the source program, it will replace the symbol with the text string.

There must be at least one space or tab between the symbol and the text string. This space or tab will not be included in the text string. The text string is a string of any characters. The assembler recognizes the end of the text string by the carriage return or semicolon (;).

A symbol with the same name cannot be doubly defined with the DEFINE directive, but if the text strings are identical then a double definition is possible.

■ **Example** ■

```
DEFINE  FLAG  [OFFSET WORK][0]
```

4.8.3 SUB

■ **Syntax** ■

```
SUB symbol  [LOCAL(symbol_list)]
    .
    .
    .
ENDSUB
```

■ **Function** ■

The SUB directive causes the statements between SUB and ENDSUB to be assembled if the *symbol* has previously been referenced. If the *symbol* has not been referenced, then the statements to the ENDSUB directive will be ignored.

This directive also automatically adds a label definition for the symbol at the start of the statement block.

The statement block between SUB and ENDSUB can contain local labels. When declaring local labels, then those label names are enclosed in parentheses after LOCAL. If there are multiple local lables, then they are delimited by commas. The local label name is replaced with a different name for each expansion. So, same name can be used in multiple statement blocks.

The SUB directive cannot be coded between another SUB and ENDSUB directive.

Refer to Chapter 10 “Sample Programs” regarding actual usage of the SUB directive.

■ **Example** ■

```
    BCAL    SUB1

SUB SUB1 LOCAL(LAB1,LAB2)
LAB1:
    [HL]=A
LAB2:
    .
    .
ENDSUB

SUB SUB2 LOCAL(LAB1)
    [HL]=A
LAB1:
    .
    .
ENDSUB

    BCAL SUB2       ; Error—label SUB2 is not defined.
```

In the above example SUB1 is expanded because it was previously referenced, but SUB2 is a forward reference and is therefore ignored. Double-defined LAB1 will not cause an error because LAB1 is declared as a local label.

4.8.4 REFER

■ **Syntax** ■

```
REFER  symbol
```

■ **Function** ■

The REFER directive is used to expand a block selected by a SUB directive regardless of whether or not the symbol has been referenced. In other words, by declaring a symbol with the REFER directive, it will be handled as a referenced symbol.

■ **Example** ■

```
REFER SUB2
SUB SUB2
    [HL]=A
    .
    .
    .
ENDSUB
```

In this example, the symbol SUB2 is declared with a REFER directive, so the statement block between SUB and ENDSUB will be assembled unconditionally.

4.8.5 Macro Definitions

■ **Syntax** ■

```
MACRO symbol ([parameter_list])[LOCAL(symbol_list)]
    .
    .
    .
ENDM
```

■ **Function** ■

Macros assign a series of statements to a single symbol. The programmer can then express the series of statements using that symbol. Macro expansion is performed during assembly each time the assembler encounters a symbol defined as a macro.

In addition, parameters can be defined for assigning different strings each time the macro is expanded. When label definitions are needed for statements within a macro, the LOCAL label definition is necessary to generate a different name for each expansion.

The macro definition is coded for the symbol after MACRO. The parameter list is enclosed in parentheses after the symbol. The parentheses are needed even if there are no parameters. The parameter list is a list of symbols delimited by commas (,). If there are label definitions within the

macro, then those label names are enclosed in parentheses after LOCAL. If there are multiple local labels, then they are delimited by commas.

Parameter names and local label names are valid only within their macro definition. They cannot be referenced elsewhere. This means that parameters and local labels with identical names can be used within other macro definitions.

■ Example ■

(1) Example of simplest macro definition

```
MACRO SHIFT_L()
  RC
  RAL
ENDM
```

(2) Example of macro with parameter

```
MACRO LBI (cnt)
  LAI cnt
  LBA
ENDM
```

(3) Example of macro with local label

```
MACRO INHL() LOCAL (label1)
  INL
  B  label1
  INH
label1:      ; This is replaced with a different name each time the macro is expanded.
ENDM
```

4.8.6 Macro Calls

■ Syntax ■

```
symbol([argument_list])
```

■ Function ■

For a macro call, the *symbol* must be the symbol of a previous macro definition. When arguments are needed, the argument list is enclosed in parentheses following the symbol. Multiple arguments are delimited by commas (,). The parentheses are required even when there are no arguments.

An argument is a string of any characters that ends just before a comma or right parenthesis. Arguments will be substituted for their corresponding parameters in the macro definition.

■ **Example** ■

These examples call the macros defined on the previous page.

(1) `SHIFT_L()`

This call will expand as follows.

```
RC
RAL
```

(2) `LBI(5 + 1)`

The macro defined a parameter “cnt,” so this call will replace “cnt” in the macro with “5 + 1.” Thus the call will expand as follows.

```
LAI 5 + 1
LBA
```

When a right parenthesis or comma is needed within an argument, add a backslash (\) before it, as shown in the example below.

```
LBI(2*(1+2\))
```

(3) `INHL()`

The assembler expands this call by giving a new name to local label “label1.”

```
    INL
    B  ?00001
    INH
?00001:
```

Chapter 5

SASM Instructions

This chapter explains SASM instructions. SASM instructions are extended instructions that are more object oriented, are easier to read, and easier to code than device instructions. Refer to chapter 6 for details of each instruction.

5.1 SASM Instruction Syntax

SASM instructions are instructions that combine multiple native CPU instructions (hereafter called basic instructions) to further enhance object orientation of data. Coding is similar to high-level languages, making programs easier to read and understand. SASM instructions are coded as follows:

LABEL:	C,	[HL]	+=	3
label definition	option	data object	operator	data object

The statement ends with a carriage return, just like other statements. Options and label definitions are specified as necessary.

5.1.1 Skips

Some SASM instructions follow “skips” the same as basic instructions. If another SASM instruction is placed immediately after a “skipping” SASM instruction, then the skipped instruction will be one basic instruction, not the SASM instruction.

■ **Example** ■

```
B == L;LAL          SASM instruction to skip
                ;CAB
B = 4           ;LAI 4
                ;LBA
```

In the above example, B==L is the skipping instruction. The B=4 instruction expands to the two basic instructions LAI 4 and LBA, so when the skip condition is met, the only instruction skipped will be LAI 4. LBA will be executed.

5.1.2 Data Objects

Data objects are operands that are the objects of calculations. They are either nibbles, bytes, or bits. The data objects of each type are listed below.

Nibble data objects

Data Object	Meaning
A	Register A
B	Register B
L	Register L
H	Register H
Y	Register Y
X	Register X
[HL]	Data nibble in memory indicated by register pair HL.
[XY]	Data nibble in memory indicated by register pair XY.
[n8]	Data nibble in memory at address n8.
[HL+]	Data nibble in memory indicated by register pair HL, which is incremented after it is ref erenced.
[HL-]	Data nibble in memory indicated by register pair HL, which is decremented after it is ref erenced.
n4	Integer constant with value 0 to 15.

Byte data objects

Data Object	Meaning
BA	Register pair BA
HL	Register pair HL
XY	Register pair XY
BYTE[HL]	Data byte in memory indicated by register pair HL.
BYTE[XY]	Data byte in memory indicated by register pair XY.
BYTE[n8]	Data byte in memory at address n8
n8	Integer constant with value 0 to 255.

Bit data objects

Data Object	Meaning
C	Carry flag
A[n2]	Bit number n2+1 from the least significant bit of register A.
[HL][n2]	Bit number n2+1 (from the least significant bit) in memory indicated by register pair HL.
[XY][n2]	Bit number n2+1 (from the least significant bit) in memory indicated by register pair XY.
[n8][n2]	Bit number n2+1 (from the least significant bit) in memory at address n8.
[HL+][n2]	Bit number n2+1 (from the least significant bit) in memory indicated by register pair HL, which is incremented after it is referenced.
[HL-][n2]	Bit number n2+1 (from the least significant bit) in memory indicated by register pair HL, which is decremented after it is referenced.
n1	Integer constant with value 0 or 1.

HL can be omitted, as shown in the examples below

■ Example ■

```
[]          ; Same as [HL]
[+]         ; Same as [HL+]
[][2]       ; Same as [HL][2]
```

[HL+] and [HL-] cannot be used in skipping instructions. They also cannot be used as both the left and right data objects in a statement. When an instruction uses one of these data objects, if the result of incrementing or decrementing HL generates an overflow or borrow, then the next basic instruction will be skipped.

■ Example ■

```
[HL+] = A           ;correct
[HL+] += 3          ;error—cannot be used in skipping instruction
[HL+] = [HL-]       ;error—cannot be used on both sides
```

5.1.3 Operators

This section shows which operators are appropriate for which calculations with data objects. Some operators work on two data objects, and some on one. When they work on two data objects, the size of both must match.

There are restrictions on the data objects that can be used with operators. For details refer to chapter 6, “SASM Instruction Details.”

Transfer operators

Operator	Meaning	Type
obj1 = obj2	Assigns contents of obj2 to obj1.	bit, nibble, byte
obj1 <> obj2	Exchanges contents of obj1 and obj2.	bit, nibble, byte

Comparison operators

Operator	Meaning	Type
obj1 == obj2	If obj1 is equal to obj2, then do not execute the next basic instruction.	bit, nibble, byte
obj1 != obj2	If obj1 is not equal to obj2, then do not execute the next basic instruction.	bit, nibble, byte
obj1 < obj2	If obj1 is less than obj2, then do not execute the next basic instruction.	nibble, byte
obj1 <= obj2	If obj1 is less than or equal to obj2, then do not execute the next basic instruction.	nibble, byte
obj1 > obj2	If obj1 is greater than obj2, then do not execute the next basic instruction.	nibble, byte
obj1 >= obj2	If obj1 is greater than or equal to obj2, then do not execute the next basic instruction.	nibble, byte

Arithmetic operators

Operator	Meaning	Type
obj1 += obj2	Assign the addition of obj1 and obj2 to obj1.	nibble, byte
obj1 –= obj2	Assign the subtraction of obj2 from obj1 to obj1.	nibble, byte
obj1 &= obj2	Assign the bitwise logical AND of obj1 and obj2 to obj1.	nibble, byte
obj1 |= obj2	Assign the bitwise logical OR of obj1 and obj2 to obj1.	nibble, byte
obj1 ^= obj2	Assign the bitwise exclusive OR of obj1 and obj2 to obj1.	nibble, byte
obj1 >> obj2	Right shift obj1 by the value of obj2.	nibble, byte
obj1 << obj2	Left shift obj1 by the value of obj2.	nibble, byte
obj1++	Increment obj1. If the increment generates an overflow, then skip the next instruction.	nibble, byte
obj1—	Decrement obj1. If the decrement generates a borrow, then skip the next instruction.	nibble, byte

5.1.4 Options

Options allow finer control of instruction operation. An options is specified before the SASM instruction and delimited from it by a comma (,).

Multiple differing options can be coded. In such cases the options should be delimited by commas. Their order is irrelevant. The same option cannot be coded multiple times.

Option types and functions are shown below.

C (carry) option

Instruction Used In	Function
Used in addition (+=)	Perform addition with carry. If the addition generates an overflow, then the carry will be set.
Used in subtraction (–=)	Perform subtraction with carry. If the subtraction generates a borrow, then the carry will be set.
Used in right shift (>>)	Perform right rotate including carry.
Used in left shift (<<)	Perform left rotate including carry.
Used in other instructions	No effect.

NS (no skip) option

Instruction Used In	Function
Used in increment (++)	Do not skip the next instruction even if the increment generates an overflow.
Used in decrement (—)	Do not skip the next instruction even if the decrement generates a borrow.
Used in other instructions	No effect.

■ **Example** ■

```
C, [80H] += []          ;Add with carry.
NS, HL++                ;Do not skip on overflow.
```

5.1.5 Data Object Restrictions

There are restrictions on the data objects that can be used with the operators. The table below lists the data option restrictions. Depending on the instruction, there may be various other restrictions, so refer to chapter 6, “SASM Instruction Details.”

● **Nibble**

Data objects: A, B, H, L, X, Y, [HL], [XY], [n8], n4

obj1, operation, obj2:

Type	Operation	obj1	obj2	[HL+],[HL-]
Transfer	=	All except n4	All	Allowed
	<>	All except n4	All except n4	Allowed
Comparison	==	All except n4	All	Not allowed
	!=	All except n4	All	Not allowed
	>	All except n4	[HL], [XY], n4	Not allowed
	>=	All except n4	[HL], [XY], n4	Not allowed
	<	All except n4	[HL], [XY], n4	Not allowed
	<=	All except n4	[HL], [XY], n4	Not allowed
Arithmetic	+=	All except n4	[HL], [XY], n4	Allowed
	–=	All except n4	[HL], [XY], n4	Allowed
	&=	All except n4	[HL], [XY], n4	Allowed
	|=	All except n4	[HL], [XY], n4	Allowed
	^=	All except n4	[HL], [XY], n4	Allowed
	>>	All except n4	n4	Allowed
	<<	All except n4	n4	Allowed
	++	All except n4	None	Not allowed
	—	All except n4	None	Not allowed

● **Byte**

Data objects: BA, HL, XY, BYTE[HL], BYTE[XY], n8

obj1, operation, obj2:

Type	Operation	obj1	obj2	Notes
Transfer	=	All except n8	All	–
	<>	All except n8	All except n8	See **Note 1**
Comparison	==	All except n8	All	See **Note 1**
	!=	All except n8	All	See **Note 1**
	>	All except n8	BYTE[HL], BYTE[XY], n8	–
	>=	All except n8	BYTE[HL], BYTE[XY], n8	–
	<	All except n8	BYTE[HL], BYTE[XY], n8	–
	<=	All except n8	BYTE[HL], BYTE[XY], n8	–
Arithmetic	+=	All except n8	BYTE[HL], BYTE[XY], n8	–
	–=	All except n8	BYTE[HL], BYTE[XY], n8	–
	&=	All except n8	BYTE[HL], BYTE[XY], n8	See **Note 2**
	|=	All except n8	BYTE[HL], BYTE[XY], n8	See **Note 2**
	^=	All except n8	BYTE[HL], BYTE[XY], n8	See **Note 2**
	>>	All except n8	n8	–
	<<	All except n8	n8	–
	++	All except n8	None	–
	—	All except n8	None	–

[Note 1] Combinations of BA, HL, XY not allowed.

[Note 2] Combinations of BA, BYTE[HL], BYTE[XY] not allowed.

● **Bit**

Data objects: A[n2], [HL][n2], [XY][n2], [n8][n2], n1

obj1, operation, obj2:

Type	Operation	obj1	obj2	[HL+],[HL-]
Transfer	=	All except n1	All	Allowed
	<>	All except n1	All except n1	Allowed
Comparison	==	All except n1	All	Not allowed
	!=	All except n1	All	Not allowed

5.1.6 SASM Instruction Expansion

If the operation of a SASM instruction is that of a basic instruction, then it will always expand into that basic instruction. For example, A=1 will always expand to LAI 1. When a SASM instruction expands to multiple basic instructions which need a register temporarily, it will use the A register, B register, or carry. Therefore you should be aware when using SASM instructions that there is a possibility that the contents of those registers or carry may change.

You can see how SASM instructions are expanded by specifying the /A option when you start the assembler and then looking at the assembly source file output.

Chapter 6

SASM Instruction Details

This chapter explains the details of SASM instructions. Be sure to read this chapter if you will use SASM instructions.

6.1 Nibble Assignments

■ **Syntax** ■

```
obj1 = obj2
```

■ **Function** ■

Assigns obj2 to obj1.

■ **Data Objects Allowed** ■

obj1: A, B, L, H, Y, X, [HL], [XY], [n8], [HL+], [HL-]
obj2: A, B, L, H, Y, X, [HL], [XY], [n8], [HL+], [HL-], n4

■ **Options** ■

None

■ **Examples** ■

Source	Expansion
A = [HL]	LAM
[70H] = [XY]	LAM @XY LMAD 070H
[HL+] = 4	LAI 04H LMA+

■ Byte Number ■

obj1	obj2											
	A	B	L	H	Y	X	[HL]	[XY]	[m8]	[HL+]	[HL–]	n4
A	0	2	1	1	1	1	1	2	2	1	1	1
B	2	0	2	2	2	2	2	3	3	3	3	2
L	1	3	0	2	2	2	2	3	3	3	3	1
H	1	3	2	0	2	2	2	3	3	3	3	2
Y	1	3	2	2	0	2	2	3	3	3	3	2
X	1	3	2	2	2	0	2	3	3	3	3	2
[HL]	1	3	2	2	2	2	0	3	3	1	1	2
[XY]	2	4	3	3	3	3	3	0	4	4	4	3
[m8]	2	4	3	3	3	3	3	4	4	4	4	3
[HL+]	1	3	2	2	2	2	1	3	3	–	–	2
[HL–]	1	3	2	2	2	2	1	3	3	–	–	2

6.2 Nibble Exchanges

■ Syntax ■

```
obj1 <> obj2
```

■ Function ■

Exchanges contents of obj1 and obj2.

■ Data Objects Allowed ■

obj1: A, B, L, H, Y, X, [HL], [XY], [n8], [HL+], [HL-]
obj2: A, B, L, H, Y, X, [HL], [XY], [n8], [HL+], [HL-]

■ Options ■

None

■ Examples ■

Source	Expansion
[70H] <> A	XAMD 070H
[HL] <> [XY]	LAM
	XAM @XY
	LMA
B <> [HL+]	LAM
	XAB
	LMA+

■ Byte Number ■

obj1	obj2											
	A	B	L	H	Y	X	[HL]	[XY]	[m8]	[HL+]	[HL–]	n4
A	0	1	5	5	5	5	1	2	2	1	1	–
B	1	0	3	3	3	3	3	4	4	3	3	–
L	5	3	0	6	6	6	3	4	4	4	4	–
H	5	3	6	0	6	6	3	4	4	4	4	–
Y	5	3	6	6	0	6	3	4	4	4	4	–
X	5	3	6	6	6	0	3	4	4	4	4	–
[HL]	1	3	3	3	3	3	0	4	4	1	1	–
[XY]	2	4	4	4	4	4	4	0	6	4	4	–
[m8]	2	4	4	4	4	4	4	6	6	4	4	–
[HL+]	1	3	4	4	4	4	1	4	4	–	–	–
[HL–]	1	3	4	4	4	4	1	4	4	–	–	–

6.3 Nibble Equivalence Comparisons

■ Syntax ■

(1) `obj1 == obj2`
(2) `obj1 != obj2`

■ Function ■

(1) If obj1 is equal to obj2 ,then skip the next basic instruction.
(2) If obj1 is not equal to obj2, then skip the next basic instruction.

■ Data Objects Allowed ■

obj1: A, B, L, H, Y, X, [HL], [XY], [n8]
obj2: A, B, L, H, Y, X, [HL], [XY], [n8], n4

■ Options ■

None

■ Examples ■

Source	Expansion
Y == 3	LAY
	CAI 03H
[70H] == [72H]	LAMD 070H
	CAMD 072H
A != B	SC
	CAB
	TC
A != [HL]	EOR
	AIS 0FH

■ Byte Number ■

(1)

obj1	obj2											
	A	B	L	H	Y	X	[HL]	[XY]	[m8]	[HL+]	[HL–]	n4
A	0	1	3	3	3	3	1	2	2	–	–	2
B	1	0	2	2	2	2	2	3	3	–	–	2
L	3	2	0	3	4	4	2	3	3	–	–	2
H	3	2	3	0	4	4	2	3	3	–	–	3
Y	3	2	4	4	0	3	2	3	3	–	–	3
X	3	2	4	4	3	0	2	3	3	–	–	3
[HL]	1	2	2	2	2	2	0	3	3	–	–	2
[XY]	2	3	3	3	3	3	3	0	4	–	–	3
[m8]	2	3	3	3	3	3	3	4	4	–	–	3
[HL+]	–	–	–	–	–	–	–	–	–	–	–	–
[HL–]	–	–	–	–	–	–	–	–	–	–	–	–

(2)

obj1	obj2											
	A	B	L	H	Y	X	[HL]	[XY]	[m8]	[HL+]	[HL–]	n4
A	0	3	5	5	5	5	2	3	4	–	–	3
B	3	0	4	4	4	4	3	4	5	–	–	5
L	5	4	0	5	6	6	3	4	5	–	–	4
H	5	4	5	0	6	6	3	4	5	–	–	4
Y	5	4	6	6	0	5	3	4	5	–	–	4
X	5	4	6	6	5	0	3	4	5	–	–	4
[HL]	2	3	3	3	3	3	0	4	4	–	–	3
[XY]	3	4	4	4	4	4	4	0	5	–	–	4
[m8]	4	5	5	5	5	5	4	5	6	–	–	5
[HL+]	–	–	–	–	–	–	–	–	–	–	–	–
[HL–]	–	–	–	–	–	–	–	–	–	–	–	–

6.4 Nibble Greater/Less-Than Comparisons

■ Syntax ■

(1) `obj1 > obj2`
(2) `obj1 >= obj2`
(3) `obj1 < obj2`
(4) `obj1 <= obj2`

■ Function ■

(1) If obj1 is greater than obj2, then skip the next basic instruction.
(2) If obj1 is greater than or equal to obj2, then skip the next basic instruction.
(3) If obj1 is less than obj2, then skip the next basic instruction.
(4) If obj1 is less than or equal to obj2, then skip the next basic instruction.

■ Data Objects Allowed ■

obj1: A, B, L, H, Y, X, [HL], [XY], [n8]
obj2: [HL], [XY], n4

■ Options ■

None

■ Examples ■

Source	Expansion
A > [HL]	SC
	SUBCS
A >= [XY]	RC
	SUBCS @XY
L < [HL]	LAL
	SUBS
A <= 2	SC
	AIS ((~(02H))&0FH)
	TC

■ Byte Number ■

(1)

obj1	obj2		
	[HL]	[XY]	n4
A	2	3	1
B	4	5	3
L	3	4	2
H	3	4	2
Y	3	4	2
X	3	4	2
[HL]	0	4	2
[XY]	4	0	3
[m8]	4	5	3
[HL+]	–	–	–
[HL–]	–	–	–

(2)

obj1	obj2		
	[HL]	[XY]	n4
A	2	3	4
B	4	5	6
L	3	4	5
H	3	4	5
Y	3	4	5
X	3	4	5
[HL]	0	4	5
[XY]	4	0	6
[m8]	4	5	6
[HL+]	–	–	–
[HL–]	–	–	–

(3)

obj1	obj2		
	[HL]	[XY]	n4
A	1	2	3
B	3	4	5
L	2	3	4
H	2	3	4
Y	2	3	4
X	2	3	4
[HL]	0	3	4
[XY]	3	0	5
[m8]	3	4	5
[HL+]	–	–	–
[HL–]	–	–	–

(4)

obj1	obj2		
	[HL]	[XY]	n4
A	3	4	3
B	5	6	5
L	4	5	4
H	4	5	4
Y	4	5	4
X	4	5	4
[HL]	0	5	4
[XY]	5	0	5
[m8]	5	6	5
[HL+]	–	–	–
[HL–]	–	–	–

6.5 Nibble Additions & Subtractions

■ Syntax ■

(1) `obj1 += obj2`
(2) `obj1 Đ= obj2`

■ Function ■

(1) Assign the addition of obj1 and obj2 to obj1.
(2) Assign the subtraction of obj2 from obj1 to obj1.

■ Data Objects Allowed ■

obj1: A, B, L, H, Y, X, [HL], [XY], [n8], [HL+], [HL-]
obj2: [HL], [XY], n4

■ Options ■

C Perform both addition and subtraction with carry. If the result generates an overflow or borrow, then set the carry.

■ Examples ■

Source	Expansion
[HL] += 4	LAM
	AIS 04H
	NOP
	LMA
C, [HL+] += 4	LAI 04H
	ADC
	XAM+
B –= [XY]	XAB
	SUBS @XY
	NOP
	XAB

■ Byte Number ■

(1)

obj1	obj2		
	[HL]	[XY]	n4
A	2	3	2
B	4	5	4
L	4	5	4
H	4	5	4
Y	4	5	4
X	4	5	4
[HL]	4	5	4
[XY]	6	6	6
[m8]	6	7	6
[HL+]	4	5	4
[HL–]	4	5	4

(2)

obj1	obj2		
	[HL]	[XY]	n4
A	2	3	2
B	4	5	4
L	4	5	4
H	4	5	4
Y	4	5	4
X	4	5	4
[HL]	2	5	4
[XY]	6	3	6
[m8]	6	7	6
[HL+]	2	5	4
[HL–]	2	5	4

(1) with C option

obj1	obj2		
	[HL]	[XY]	n4
A	1	2	7
B	3	4	9
L	3	4	9
H	3	4	9
Y	3	4	9
X	3	4	9
[HL]	3	4	3
[XY]	5	6	5
[m8]	5	6	11
[HL+]	3	4	3
[HL–]	3	4	3

(2) with C option

obj1	obj2		
	[HL]	[XY]	n4
A	1	2	11
B	3	4	13
L	3	4	13
H	3	4	13
Y	3	4	13
X	3	4	13
[HL]	3	4	4
[XY]	5	6	7
[m8]	5	6	15
[HL+]	3	4	4
[HL–]	3	4	4

6.6 Nibble Logical Operations

■ Syntax ■

(1) `obj1 &= obj2`
(2) `obj1 |= obj2`
(3) `obj1 ^= obj2`

■ Function ■

(1) Assign logical AND of obj1 and obj2 to obj1.
(2) Assign logical OR of obj1 and obj2 to obj1.
(3) Assign exclusive OR of obj1 and obj2 to obj1.

■ Data Objects Allowed ■

obj1: A, B, L, H, Y, X, [HL], [XY], [n8], [HL+], [HL-]
obj2: [HL], [XY], n4

■ Options ■

None

■ Examples ■

Source	Expansion
H &= [XY]	LAH
	AND @XY
	LHA
B |= 0AH	XAB
	ORI 0AH
	XAB
[HL] ^= 5	LAI 05H
	EOR
	LMA

■ Byte Number ■

(1)

obj1	obj2		
	[HL]	[XY]	n4
A	1	2	2
B	3	4	4
L	3	4	4
H	3	4	4
Y	3	4	4
X	3	4	4
[HL]	0	4	3
[XY]	5	0	5
[m8]	5	6	6
[HL+]	1	4	3
[HL–]	1	4	3

(2)

obj1	obj2		
	[HL]	[XY]	n4
A	1	2	2
B	3	4	4
L	3	4	4
H	3	4	4
Y	3	4	4
X	3	4	4
[HL]	0	4	3
[XY]	5	0	5
[m8]	5	6	6
[HL+]	1	4	3
[HL–]	1	4	3

(3)

obj1	obj2		
	[HL]	[XY]	n4
A	1	2	2
B	3	4	4
L	3	4	4
H	3	4	4
Y	3	4	4
X	3	4	4
[HL]	2	4	3
[XY]	5	3	5
[m8]	5	6	6
[HL+]	3	4	3
[HL–]	3	4	3

6.7 Nibble Shifts

■ Syntax ■

(1) `obj1 >> obj2`
(2) `obj1 << obj2`

■ Function ■

(1) Right shift obj1 by value of obj2.
(2) Left shift obj1 by value of obj2.

■ Data Objects Allowed ■

obj1: A, B, L, H, Y, X, [HL], [XY], [n8], [HL+], [HL-]
obj2: n2

■ Options ■

C
(1) Rotate, with the carry value assigned to bit 4 of obj1 (from the least significant bit) and then bit 1 of obj1 assigned to the carry.
(2) Rotate, with the carry value assigned to bit 1 of obj1 (from the least significant bit) and then bit 4 of obj1 assigned to the carry.

■ Examples ■

Source	Expansion
L >> 1	LAL
	RC
	RAR
	LLA
C, L >> 2	LAL
	RAR
	RAR
	LAL

■ Byte Number ■

(1)

obj1	obj2
	n2=1
A	2
B	4
L	4
H	4
Y	4
X	4
[HL]	4
[XY]	6
[m8]	6
[HL+]	4
[HL–]	4

(2)

obj1	obj2
	n2=1
A	2
B	4
L	4
H	4
Y	4
X	4
[HL]	4
[XY]	6
[m8]	6
[HL+]	4
[HL–]	4

(1) with C option

obj1	obj2
	n2=1
A	1
B	3
L	3
H	3
Y	3
X	3
[HL]	3
[XY]	5
[m8]	5
[HL+]	3
[HL–]	3

(2) with C option

obj1	obj2
	n2=1
A	1
B	3
L	3
H	3
Y	3
X	3
[HL]	3
[XY]	5
[m8]	5
[HL+]	3
[HL–]	3

6.8 Nibble Increments & Decrements

■ Syntax ■

(1) `obj1 ++`
(2) obj 1 – –

■ Function ■

(1) Increment obj1. If the increment generates an overflow, then skip the next basic instruction.
(2) Decrement obj1. If the decrement generates a borrow, then skip the next basic instruction.

■ Data Objects Allowed ■

obj1: A, B, L, H, Y, X, [HL], [XY], [n8]

■ Options ■

NS Do not skip the next basic instruction even if the calculation result generates an overflow or borrow.

■ Examples ■

Source	Expansion
[XY]++	INM @XY
NS, [70H]++	INMD 070H
	NOP
L--	DCL
NS, B--	XAB
	DCA
	NOP
	XAB

■ Byte Number ■

(1)

obj1	obj2
	–
A	1
B	6
L	1
H	1
Y	1
X	1
[HL]	1
[XY]	2
[m8]	2
[HL+]	–
[HL–]	–

(2)

obj1	obj2
	–
A	1
B	6
L	1
H	1
Y	1
X	1
[HL]	1
[XY]	2
[m8]	2
[HL+]	–
[HL–]	–

(1) with NS option

obj1	obj2
	–
A	2
B	4
L	2
H	2
Y	2
X	2
[HL]	2
[XY]	3
[m8]	3
[HL+]	–
[HL–]	–

(2) with NS option

obj1	obj2
	–
A	2
B	4
L	2
H	2
Y	2
X	2
[HL]	2
[XY]	3
[m8]	3
[HL+]	–
[HL–]	–

6.9 Byte Assignments

■ **Syntax** ■

```
obj1 = obj2
```

■ **Function** ■

Assigns obj2 to obj1.

■ **Data Objects Allowed** ■

obj1: BA, HL, XY, BYTE[HL], BYTE[XY], BYTE[n8]
obj2: BA, HL, XY, BYTE[HL], BYTE[XY], BYTE[n8], n8

■ **Options** ■

None

■ **Examples** ■

Source	Expansion
BA = HL	LALB
BYTE[HL] = BYTE[XY]	LAMB @XY LMAB

■ Byte Number ■

obj1	obj2						
	BA	HL	XY	BYTE[HL]	BYTE[XY]	BYTE[m8]	n8
BA	0	2	2	1	2	2	2
HL	2	0	4	3	4	4	2
XY	2	4	0	3	4	4	2
BYTE[HL]	1	3	3	0	3	3	2
BYTE[XY]	2	4	4	3	0	4	3
BYTE[m8]	2	4	4	3	4	4	4

6.10 Byte Exchanges

■ **Syntax** ■

```
obj1 <> obj2
```

■ **Function** ■

Exchanges contents of obj 1 and obj2.

■ **Data Objects Allowed** ■

obj1: BA, HL, XY, BYTE[HL], BYTE[XY], BYTE[n8]
obj2: BA, HL, XY, BYTE[HL], BYTE[XY], BYTE[n8]

However, BA cannot be exchanged with HL or XY.

■ **Options** ■

None

■ **Examples** ■

Source	Expansion
BA <> BYTE[70H]	XAMDB 070H
BYTE[HL] <> BYTE[70H]	LAMB
	XAMDB 070H
	LMAB

■ Byte Number ■

obj1	obj2						
	BA	HL	XY	BYTE[HL]	BYTE[XY]	BYTE[m8]	n8
BA	0	–	–	1	2	2	–
HL	–	0	12	5	6	6	–
XY	–	12	0	5	6	6	–
BYTE[HL]	1	5	5	0	4	4	–
BYTE[XY]	2	6	6	4	0	6	–
BYTE[m8]	2	6	6	4	6	6	–

6.11 Byte Equivalence Comparisons

■ **Syntax** ■

(1) `obj1 == obj2`
(2) `obj1 != obj2`

■ **Function** ■

(1) If obj1 is equal to obj2 ,then skip the next basic instruction.
(2) If obj1 is not equal to obj2, then skip the next basic instruction.

■ **Data Objects Allowed** ■

obj1: BA, HL, XY, BYTE[HL], BYTE[XY], BYTE[n8]
obj2: BA, HL, XY, BYTE[HL], BYTE[XY], BYTE[n8], n8

However, BA cannot be compared with HL or XY.

■ **Options** ■

None

■ **Examples** ■

Source	Expansion
BYTE[HL] == BYTE[XY]	LAMB CAMB @XY
BA == BYTE[70H]	CAMD (070H)&0FEH B ?00000 XAB CAMD (070H)|01H ?00000:
BA != BYTE[HL]	SC CAMB TC

■ Byte Number ■

(1)

obj1	obj2						
	BA	HL	XY	BYTE[HL]	BYTE[XY]	BYTE[m8]	n8
BA	0	–	–	1	2	6	6
HL	–	0	9	3	4	8	6
XY	–	9	0	3	4	8	8
BYTE[HL]	1	3	3	0	3	3	3
BYTE[XY]	2	4	4	3	0	4	4
BYTE[m8]	6	8	8	3	4	8	8

(2)

obj1	obj2						
	BA	HL	XY	BYTE[HL]	BYTE[XY]	BYTE[m8]	n8
BA	0	–	–	3	4	8	8
HL	–	0	11	5	6	10	8
XY	–	11	0	5	6	10	10
BYTE[HL]	3	5	5	0	5	5	5
BYTE[XY]	4	6	6	5	0	6	6
BYTE[m8]	8	10	10	5	6	10	10

6.12 Byte Greater/Less-Than Comparisons

■ Syntax ■

(1) `obj1 > obj2`
(2) `obj1 >= obj2`
(3) `obj1 < obj2`
(4) `obj1 <= obj2`

■ Function ■

(1) If obj1 is greater than obj2, then skip the next basic instruction.
(2) If obj1 is greater than or equal to obj2, then skip the next basic instruction.
(3) If obj1 is less than obj2, then skip the next basic instruction.
(4) If obj1 is less than or equal to obj2, then skip the next basic instruction.

■ Data Objects Allowed ■

obj1: BA, HL, XY, BYTE[HL], BYTE[XY], BYTE[n8]
obj2: BYTE[HL], BYTE[XY], n8

■ Options ■

None

■ Examples ■

Source	Expansion
BYTE[HL] > 10H	LBAI 010H SUBSB
BYTE[XY] >= 10H	LBAI 010H SC SUBCB @XY TC
BA < BYTE[HL]	SUBSB
BA <= BYTE[XY]	SC SUBCB @XY TC

■ Byte Number ■

(1)

obj1	obj2		
	BYTE[HL]	BYTE[XY]	n8
BA	6	7	12
HL	8	9	14
XY	8	9	14
BYTE[HL]	0	3	3
BYTE[XY]	3	0	4
BYTE[m8]	8	9	14

(2)

obj1	obj2		
	BYTE[HL]	BYTE[XY]	n8
BA	3	4	16
HL	5	6	18
XY	5	6	18
BYTE[HL]	0	5	5
BYTE[XY]	5	0	6
BYTE[m8]	5	6	18

(3)

obj1	obj2		
	BYTE[HL]	BYTE[XY]	n8
BA	1	2	16
HL	3	4	18
XY	3	4	18
BYTE[HL]	0	3	8
BYTE[XY]	3	0	9
BYTE[m8]	3	4	18

(4)

obj1	obj2		
	BYTE[HL]	BYTE[XY]	n8
BA	3	4	12
HL	5	6	14
XY	5	6	14
BYTE[HL]	0	5	5
BYTE[XY]	5	0	6
BYTE[m8]	5	6	14

6.13 Byte Additions & Subtractions

■ **Syntax** ■

(1) `obj1 += obj2`
(2) `obj1 -= obj2`

■ **Function** ■

(1) Assign the addition of obj1 and obj2 to obj1.
(2) Assign the subtraction of obj2 from obj1 to obj1.

■ **Data Objects Allowed** ■

obj1: BA, HL, XY, BYTE[HL], BYTE[XY], BYTE[n8]
obj2: BYTE[HL], BYTE[XY], n8

■ **Options** ■

C Perform both addition and subtraction with carry. If the result generates an overflow or borrow, then set the carry.

■ **Examples** ■

Source	Expansion
BA += BYTE[HL]	ADSB
	NOP
C, BYTE[HL] += 55H	LBAI 055H
	ADCB
	LMAB
BYTE[70H] –= BYTE[HL]	LAMDB 070H
	SUBSB
	NOP
	LMADB 070H
C, BYTE[XY] –= 55H	LBAI 055H
	XAMB @XY
	SUBCB @XY
	XAMB @XY

■ Byte Number ■

(1)

obj1	obj2		
	BYTE[HL]	BYTE[XY]	n8
BA	2	3	10
HL	6	7	14
XY	6	7	14
BYTE[HL]	4	5	5
BYTE[XY]	6	7	7
BYTE[m8]	6	7	14

(2)

obj1	obj2		
	BYTE[HL]	BYTE[XY]	n8
BA	2	3	10
HL	6	7	14
XY	6	7	14
BYTE[HL]	2	5	6
BYTE[XY]	6	3	9
BYTE[m8]	6	7	14

(1) with C option

obj1	obj2		
	BYTE[HL]	BYTE[XY]	n8
BA	1	2	16
HL	5	6	20
XY	5	6	20
BYTE[HL]	3	4	4
BYTE[XY]	5	6	6
BYTE[m8]	5	6	20

(2) with C option

obj1	obj2		
	BYTE[HL]	BYTE[XY]	n8
BA	1	2	24
HL	5	6	28
XY	5	6	28
BYTE[HL]	3	4	5
BYTE[XY]	5	6	8
BYTE[m8]	5	6	28

6.14 Byte Logical Operations

■ **Syntax** ■

(1) `obj1 &= obj2`
(2) `obj1 |= obj2`
(3) `obj1 ^= obj2`

■ **Function** ■

(1) Assign logical AND of obj1 and obj2 to obj1.
(2) Assign logical OR of obj1 and obj2 to obj1.
(3) Assign exclusive OR of obj1 and obj2 to obj1.

■ **Data Objects Allowed** ■

obj1: BA, HL, XY, BYTE[HL], BYTE[XY], BYTE[n8]
obj2: BYTE[HL], BYTE[XY], n8

However, BA cannot be operationed with BYTE[HL] or BYTE[XY].

■ **Options** ■

None

■ **Examples** ■

Source	Expansion
BA &= 55H	ANDI ((055H)&0FH)
	XAB
	ANDI (((055H)>>04H)&0FH)
	XAB
HL |= 55H	LALB
	ORI ((055H)&0FH)
	XAB
	ORI (((055H)>>04H)&0FH)
	LLAB
BA ^= 55H	EORI ((055H)&0FH)
	XAB
	EORI (((055H)>>04H)&0FH)
	XAB

■ Byte Number ■

(1)

obj1	obj2		
	BYTE[HL]	BYTE[XY]	n8
BA	–	–	6
HL	18	20	10
XY	18	20	10
BYTE[HL]	0	18	8
BYTE[XY]	18	0	10
BYTE[m8]	18	20	10

(2)

obj1	obj2		
	BYTE[HL]	BYTE[XY]	n8
BA	–	–	6
HL	18	20	10
XY	18	20	10
BYTE[HL]	0	18	8
BYTE[XY]	18	0	10
BYTE[m8]	18	20	10

(3)

obj1	obj2		
	BYTE[HL]	BYTE[XY]	n8
BA	–	–	6
HL	18	20	10
XY	18	20	10
BYTE[HL]	2	18	8
BYTE[XY]	18	3	10
BYTE[m8]	18	20	10

6.15 Byte Shifts

■ **Syntax** ■

(1) `obj1 >> obj2`
(2) `obj1 << obj2`

■ **Function** ■

(1) Right shift obj1 by value of obj2.
(2) Left shift obj1 by value of obj2.

■ **Data Objects Allowed** ■

obj1: BA, HL, XY, BYTE[HL], BYTE[XY], BYTE[n8]
obj2: n3

■ **Options** ■

C (1) Rotate, with the carry value assigned to bit 8 of obj1 (from the least significant bit) and then bit 1 of obj1 assigned to the carry.
(2) Rotate, with the carry value assigned to bit 1 of obj1 (from the least significant bit) and then bit 8 of obj1 assigned to the carry.

■ **Examples** ■

Source	Expansion
BA >> 1	RC
	XAB
	RAR
	XAB
	RAR
C, HL << 1	LALB
	RAL
	XAB
	RAL
	XAB
	LLAB

■ Byte Number ■

(1)

obj1	obj2
	n3=1
BA	5
HL	9
XY	9
BYTE[HL]	7
BYTE[XY]	9
BYTE[m8]	9

(2)

obj1	obj2
	n3=1
BA	5
HL	9
XY	9
BYTE[HL]	7
BYTE[XY]	9
BYTE[m8]	9

(1) with C option

obj1	obj2
	n3=1
BA	4
HL	8
XY	8
BYTE[HL]	6
BYTE[XY]	8
BYTE[m8]	8

(2) with C option

obj1	obj2
	n3=1
BA	4
HL	8
XY	8
BYTE[HL]	6
BYTE[XY]	8
BYTE[m8]	8

6.16 Byte Increments & Decrements

■ Syntax ■

(1) `obj1 ++`
(2) obj1 ÐÐ

■ Function ■

(1) Increment obj1. If the increment generates an overflow, then skip the next basic instruction.
(2) Decrement obj1. If the decrement generates a borrow, then skip the next basic instruction.

■ Data Objects Allowed ■

obj1: BA, HL, XY, BYTE[HL], BYTE[XY], BYTE[n8]

■ Options ■

NS Do not skip the next basic instruction even if the calculation result generates an overflow or borrow.

■ Examples ■

Source	Expansion
HL++	INL B ?00000 INH ?00000:
NS, XÐÐ	DCY B ?00001 DCX NOP ?00001:

■ Byte Number ■

(1)

obj1	obj2
	–
BA	8
HL	3
XY	3
BYTE[HL]	6
BYTE[XY]	8
BYTE[m8]	5

(2)

obj1	obj2
	–
BA	8
HL	3
XY	3
BYTE[HL]	7
BYTE[XY]	9
BYTE[m8]	5

(1) with NS option

obj1	obj2
	–
BA	6
HL	4
XY	4
BYTE[HL]	5
BYTE[XY]	7
BYTE[m8]	6

(2) with NS option

obj1	obj2
	–
BA	6
HL	4
XY	4
BYTE[HL]	5
BYTE[XY]	7
BYTE[m8]	6

6.17 Bit Assignments

■ **Syntax** ■

```
obj1 = obj2
```

■ **Function** ■

Assigns obj2 to obj1.

■ **Data Objects Allowed** ■

obj1: A[n2], [HL][n2], [XY][n2], [n8][n2], [HL+][n2], [HL-][n2]
obj2: A[n2], [HL][n2], [XY][n2], [n8][n2], [HL+][n2], [HL-][n2], n1

■ **Options** ■

None

■ **Examples** ■

Source	Expansion
A[0] = 1	ORI 01H<<(00H)
[HL+][1] = 0	RMB 01H
	INL
C = [70H][0]	SC
	TMDB 070H, 00H
	RC

■ Byte Number ■

obj1	obj2								
	C	A[n2]	[HL][n2]	[XY][n2]	[m8][n2]	[HL+][n2]	[HL–][n2]	n1=0	n1=1
C	0	3	3	4	4	4	4	1	1
A[n2]	5	5	5	6	6	6	6	2	2
[HL][n2]	3	3	3	4	4	4	4	1	1
[XY][n2]	5	5	5	6	6	6	6	2	2
[m8][n2]	5	5	5	6	6	6	6	2	2
[HL+][n2]	4	4	4	5	5	–	–	2	2
[HL–][n2]	4	4	4	5	5	–	–	2	2

6.18 Bit Exchanges

■ **Syntax** ■

```
obj1 <> obj2
```

■ **Function** ■

Exchanges contents of obj1 and obj2.

■ **Data Objects Allowed** ■

obj1: A[n2], [HL][n2], [XY][n2], [n8][n2], [HL+][n2], [HL-][n2]
obj2: A[n2], [HL][n2], [XY][n2], [n8][n2], [HL+][n2], [HL-][n2]

■ **Options** ■

None

■ **Examples** ■

Source	Expansion
C <> [HL][2]	LAM
	SMB 02H
	TC
	RMB 02H
	SC
	TAB 02H
	RC

■ Byte Number ■

obj1	obj2								
	C	A[n2]	[HL][n2]	[XY][n2]	[m8][n2]	[HL+][n2]	[HL–][n2]	n1=0	n1=1
C	0	12	7	10	10	8	8	–	–
A[n2]	12	13	11	14	14	12	12	–	–
[HL][n2]	7	11	7	10	10	8	8	–	–
[XY][n2]	10	14	10	13	13	11	11	–	–
[m8][n2]	10	14	10	13	13	11	11	–	–
[HL+][n2]	8	12	8	11	11	–	–	–	–
[HL–][n2]	8	12	8	11	11	–	–	–	–

6.19 Bit Equivalence Comparisons

■ Syntax ■

(1) `obj1 == obj2`
(2) `obj1 != obj2`

■ Function ■

(1) If obj1 is equal to obj2 ,then skip the next basic instruction.
(2) If obj1 is not equal to obj2, then skip the next basic instruction.

■ Data Objects Allowed ■

obj1: A[n2], [HL][n2], [XY][n2], [n8][n2]
obj2: A[n2], [HL][n2], [XY][n2], [n8][n2], n1

■ Options ■

None

■ Examples ■

Source	Expansion
[XY][0] == A[1]	TMB @XY, 00H CMA TAB 01H
[70H][1] == 1	TMBD 070H, 01H
[HL][0] != A[2]	TMB 00H B ?00000 CMA ?00000: TAB 02H
C != 0	TC

■ Byte Number ■

(1)

obj1	obj2								
	C	A[n2]	[HL][n2]	[XY][n2]	[m8][n2]	[HL+][n2]	[HL–][n2]	n1=0	n1=1
C	0	4	5	6	6	–	–	4	1
A[n2]	4	4	4	5	5	–	–	3	1
[HL][n2]	5	4	5	6	6	–	–	3	1
[XY][n2]	6	5	6	7	7	–	–	4	2
[m8][n2]	6	5	6	7	7	–	–	4	2
[HL+][n2]	–	–	–	–	–	–	–	–	–
[HL–][n2]	–	–	–	–	–	–	–	–	–

(2)

obj1	obj2								
	C	A[n2]	[HL][n2]	[XY][n2]	[m8][n2]	[HL+][n2]	[HL–][n2]	n1=0	n1=1
C	0	5	6	7	7	–	–	1	4
A[n2]	5	5	5	6	6	–	–	1	3
[HL][n2]	6	5	6	7	7	–	–	1	3
[XY][n2]	7	6	7	8	8	–	–	2	4
[m8][n2]	7	6	7	8	8	–	–	2	4
[HL+][n2]	–	–	–	–	–	–	–	–	–
[HL–][n2]	–	–	–	–	–	–	–	–	–

Chapter 7

Control Statements

This chapter explains the control statements that control program flow.

7.1 Bit Expressions

Bit expressions are used for determining conditions of flow control statements. Bit expressions have values of 0 or 1. A flow control statement's condition is judged true when the bit expression value is 1. Conversely, it is judged false when the value is 0.

Bit expressions cannot be coded by themselves. They are always used as part of control statements.

7.1.1 Structural Elements Of Bit Expressions

The following bit data objects are bit expressions.

```
C
A[n2]
[HL][n2]
[XY][n2]
[n8][n2]
```

If the corresponding bit is set to 1, then the bit expression will have the value 1. If reset, then the bit expression will have the value 0.

7.1.2 SKIP And NOSKIP Operators

Use of a SKIP or NOSKIP operator on a native CPU instruction (basic instruction), SASM instruction, B directive, or BCAL directive will form a bit expression.

■ Syntax ■

```
SKIP instruction
NOSKIP instruction
```

■ Function ■

The SKIP operator returns the value 1 if execution of the instruction would cause the next instruction to be skipped. It returns the value 0 if the next instruction would not be skipped.

The NOSKIP operator, opposite to the SKIP operator, returns the value 0 if the next instruction would be skipped, and the value 1 if it would not.

The SKIP operator can be omitted. When omitted, *instruction* must be enclosed in parentheses as follows.

```
(instruction)
```

■ **Example** ■

`NOSKIP INL`	This returns 0 if execution of INL would generate an overflow. Otherwise this returns 1.
`SKIP HL == 3`	This returns 1 if HL equals 3. It returns 0 if they are not equal.
`([HL] > 0)`	This returns 1 if [HL] is greater than 0. Otherwise this returns 0.

7.1.3 Operators Permitted In Bit Expressions

The logical complement operator '!' can be used on bit expressions. This operator returns 0 if the bit expression is 1, and returns 1 if the bit expression is 0.

■ **Syntax** ■

```
!bit_expression
```

The bit expression can also be enclosed in parentheses as needed.

■ **Syntax** ■

```
(bit_expression)
```

Parentheses are mainly used to make the expressions easier to read.

■ **Example** ■

```
!(SKIP HL == 3)
```

7.2 IF-ELSE-ELSEIF Statements

■ **Syntax** ■

```
IF bit_expression
    .
    .
    .
[ELSEIF bit_expression]
    .
    .
    .
[ELSE]
    .
    .
    .
ENDI
```

■ **Function** ■

This series of statements controls which statement block will be executed by evaluating the bit expressions.

ELSEIF and ELSE can be omitted, but IF and ENDI are required. Multiple ELSEIFs can be coded.

If the bit expression after IF is true, then the statement block between IF and ELSEIF (or ELSE if there is no ELSEIF) will be executed. Execution then continues from the statement after ENDI.

If the bit expression after IF is false, then the bit expressions after each ELSEIF will be evaluated in order. If one is true, then its following statement block will be executed. If there is no bit expression that is true, then the statement block after ELSE will be executed.

■ **Example 1** ■

```
IF (SKIP A[0] == 1)
    [HALT][0] = 1
ENDI
```

If the least significant bit of register A is 1, then [HALT][0]=1 will be executed. If it is not 1, then no action will be taken.

■ Example 2 ■

```
IF (SKIP BYTE[] == BA)
    A = 1
    RTS
ELSE
    A = 2
    RTS
ENDI
```

If BYTE[HL] equal the register pair BA, then A=1 and RTS will be executed. If not equal, then A=2 and RTS will be executed.

■ Example 3 ■

```
IF (SKIP BYTE[] == 0)
    RT
ELSEIF (NOSKIP BCAL CHK_1HZ)
    A = 0
ELSE
    DEC_BCD_2N()
ENDI
```

If BYTE[HL] equals 0, then RT will be executed. If not equal, and if the execution of BCAL CHK_1HZ would not cause the next instruction to be skipped, then A=0 will be executed. If it would cause the next instruction to be skipped, then the macro DEC_BCD_2N() will be executed.

7.3 WHILE Statements

■ **Syntax** ■

```
WHILE bit_expression
    .
    .
    .
ENDW
```

■ **Function** ■

The WHILE statement causes the statement block between WHILE and ENDW to be repeatedly executed while the bit expression is true. The WHILE statement differs from the REPEAT-UNTIL statements (described later) in that the condition is checked before entering the first iteration. Therefore, if the bit expression is false at the start, then the control will exit the WHILE statement block without executing the statements even once.

■ **Example** ■

```
WHILE (NOSKIP HL--)
    [] = 0
ENDW
```

While HL is decremented, and until a borrow is generated, [HL]=0 will be executed.

7.4 REPEAT-UNTIL Statements

■ **Syntax** ■

```
REPEAT
    .
    .
    .
UNTIL bit_expression
```

■ **Function** ■

The REPEAT-UNTIL statements cause the statement block between REPEAT and UNTIL to be repeatedly executed while the bit expression is true. The REPEAT-UNTIL statements differ from the WHILE statement in that the statement block is executed once before the condition is checked. Therefore, even if the bit expression is false at the start, the REPEAT-UNTIL statement block will be executed once.

■ **Example** ■

```
REPEAT
    LMTB  bnk STAGE_TABLE
    XY++
    L += 2
UNTIL (SKIP L != low GATE_GOL_NO_DATA + 2)
```

First, the statement block between REPEAT and UNTIL is executed. Then if register L is not equal to constant expression “low GATE_GOL_GO_DATA + 2”, until it becomes true the same statement block will be repeatedly executed.

7.5 SWITCH-CASE Statements

■ **Syntax** ■

```
SWITCH obj
CASE constant_expression
    .
    .
    .
[CASE constant_expression]
    .
    .
    .
[DEFAULT]
    .
    .
    .
ENDS
```

The obj can be one of the following nibble data objects.

A B L H Y X [HL] [XY] [n8] *constant_expression*

■ **Function** ■

The SWITCH statement passes control to one of the statement blocks depending on the value of obj.

The value of obj is compared with the constant expression after each CASE. The statements after the matching CASE will be executed, and then control will exit the entire SWITCH block.

When no matching CASE, the statements after DEFAULT will be executed and exit the SWITCH block. If DEFAULT is not defined, then no statement is executed and exit the SWITCH block.

Multiple statements can be defined after each CASE, and also they can be omitted.

■ **Example** ■

```
SWITCH[SLOPE_DATA]
CASE1
    BYTE[WORK] = 70H
CASE2
    BYTE[WORK] = 0AAH
CASE3
    BYTE[WORK] = 077H
DEFAULT
    BYTE[WORK] = 0A0H
ENDS
```

The value assigned to BYTE[WORK] will be determined by the value of [SLOPE_DATA]. The value assigned to BYTE[WORK] will be 70H if [SLOPE_DATA] is 1, 0AAH if 2, 077H if 3, and 0A0H for all other values.

7.6 FOR Statements

■ **Syntax** ■

```
FOR obj = constant_expression_1, constant_expression_2
    .
    .
    .
ENDF
```

The obj can be one of the following nibble data objects.

A B L H Y X [HL] [XY] [n8]

■ **Function** ■

The FOR statement initializes obj to *constant_expression_1*, and then until obj becomes equal to *constant_expression_2* will repeatedly increment obj and execute the statement block between FOR and ENDF. If obj overflows before it becomes equal to *constant_expression_2*, then control will exit the FOR statement block at that point.

■ **Example** ■

```
FOR H = high BALL_1_DATA,(high BALL_4_DATA) + 1
    L = low BALL_DATA
    IF (SKIP[][1] == 1)
        A = 0
        [HL+] = A
        [HL+] = A
        [HL+] = A
        [] = A
    ENDI
ENDF
```

Starting from “high BALL_1_DATA,” the value of H is incremented, and until it equals “(high BALL_4_DATA)+1” the statement block between FOR and ENDF will be repeatedly executed.

7.7 BREAK Statements

■ **Syntax** ■

```
BREAK
```

■ **Function** ■

The BREAK statement is used in SWITCH statement blocks and iteration blocks of FOR, WHILE, and REPEAT statements. When a BREAK statement is executed, control will exit the innermost statement block. Control will pass to the next statement after the exited control block or SWITCH block.

■ **Example** ■

```
WHILE (NOSKIP X == 0)
    INC_BCD_2N()
    IF (SKIP BYTE[] == 19H)
        BREAK
    ENDI
ENDW
```

If BYTE[HL] is equal to 19H, then control will exit the WHILE statement's iteration block, regardless of WHILE statement's iteration condition.

7.8 CONTINUE Statements

■ **Syntax** ■

```
CONTINUE
```

■ **Function** ■

The CONTINUE statement is used in iteration blocks of FOR, WHILE, and REPEAT statements. When a CONTINUE statement is executed, control will pass to the end of the innermost iteration block without executing the statements after CONTINUE. Accordingly, immediately after a CONTINUE statement is executed, the iteration condition will be checked.

■ **Example** ■

```
WHILE (NOSKIP L--)
    IF (SKIP BYTE[] == 19H)
        CONTINUE
    ENDI
    INC_BCD_2N()
ENDW
```

If BYTE[HL] is equal to 19H, then the conditional check L— will be executed without executing the macro INC_BCD_2N().

7.9 Optimization Of Bit Expressions

Some bit expressions have the same meaning although they are different in syntax. An example is shown below.

(1) SKIP A == 1

(2) NOSKIP A != 1

When these bit expressions are used in IF statements, ELSEIF statements, WHILE statements, or UNTIL statements, the expansion of both will actually be the expansion of (1), which is more efficient. Thus, following two IF statements will generate identical code.

(1) IF SKIP A == 1

(2) IF NOSKIP A != 1

These kinds of bit expressions will be optimized in the following cases.

Optimization of Bit Expression

Type	Bit Expression Being Optimized	Optimized Bit Expression
Nibble/Byte Equivalence Comparisons	SKIP obj1 != obj2	NOSKIP obj1 == obj2
	NOSKIP obj1 != obj2	SKIP obj1 == obj2
Bit Equivalence Comparisons	SKIP obj1 == 0	NOSKIP obj1 != 0
	NOSKIP obj1 == 0	SKIP obj1 != 0
	SKIP obj1 != 1	NOSKIP obj1 == 1
	NOSKIP obj1 != 1	SKIP obj1 == 1

For details on obj1 and obj2, refer to Chapter 6, “SASM Instruction Details.”

Optimization will also be performed if the SKIP operator is omitted or if a logical negation operator (!) is used instead of the NOSKIP operator.

Chapter 8

Error Messages

This chapter describes the SASM64K error messages.

8.1 Syntax Errors

Syntax errors generated during analysis of the source file will cause error messages to be output to the console or error file.

The error codes, error messages, and their meanings are listed below.

Code	Error Message
E0	**bad character xx (hex)** There is an illegal character in the source file.
E2	**bad operand** Operand syntax is incorrect.
E3	**bad range** The range of values coded for an operand is incorrect.
E4	**bad syntax** This error occurs at the start of analysis. An instruction could not be recognized.
E5	**bad const** Coding of a constant is incorrect.
E6	**bad location** The location value is not in the permitted range.
E7	**code segment only** This instruction is permitted only in the code segment.
E8	**divide by zero** There is a division by zero within a constant expression.
E11	**mnemonic not allowed.** The mnemonic is not allowed.
E13	**name required** A name is required.
E16	**redefinition xxxxxx** The symbol xxxxxx has been redefined.
E19	**undefined xxxxxx** There is no definition for the symbol xxxxxx.

Code	Error Message
E26	**absolute expression required** The operand must be a constant expression (resolvable during assembly). This error is occurred when the operand is not a constant expression in many of the directives.
E27	**B/BCAL operand must be label** The operand of a B or BCAL directive must be a label in the code segment.
E28	**missing ‘** A single quotation mark (‘) is missing.
E29	**missing “** A double quotation mark (“) is missing.
E30	**missing ENDM** An ENDM is missing.
E32	**missing)** A right parenthesis is missing.
E33	**cannot use macro name** The macro name cannot be used.
E34	**cannot use bit expression** A bit expression cannot be coded here.
E36	**missing statement xxxxxx** The statement xxxxxx is missing.
E37	**macro definition not found** A macro call was made without a macro definition.
E38	**not macro name** This name is not a macro name.
E43	**cannot use this object** This instruction cannot use this data object.
E44	**parameters mismatch** Number of macro parameters is not correct.

8.2 Warnings

Warnings are generated for code that does not have illegal syntax, but for which the assembly results might not be trusted. The output format to the console (error file) is the same as for syntax errors.

The warning codes, warning messages, and their meanings are listed below.

Code	Warning Message
W0	**illegal access to SFR** This access to the SFR area is invalid. This warning occurs is such cases as when a write is made to a read-only SFR or when a byte access is made to an SFR that does not allow byte accesses.
W1	**cannot access to high nibble of SFR byte** This is a byte access to the high nibble of an SFR that allows byte accesses.
W2	**do not put out code** This SASM instruction will not output object code.
W3	**overlap segment** The code or data segment overlaps. This warning is output when the value specified with an ORG directive is smaller than the location counter.

8.3 Fatal Errors

Fatal errors occur during SASM64K execution. When a fatal error occurs, SASM64K suspends processing , outputs a message to the console, and forcibly terminates.

Code	Error Message
F7	**file not found** The source file was not found.
F12	**memory is not enough** There is not enough memory to continue processing. The cause of this error is probably that too many symbols have been defined in the source program. If you have any TSR programs resident, you should remove them. If you have been specifying the REF option or SYM option, then remove those specifications. If the error still occurs after these measures, then unfortunately SASM64K cannot process your program. Reduce the number of symbols to fix this error.
F13	**line overflow** The number of source statements exceeds 65,535.
F16	**error(s) found in DCL file** One or more syntax errors exists in the DCL file. This means that assembler results cannot be guaranteed, so the assembler will output this message and terminate processing. This error will not occur as long as you use original DCL files provided by Oki Electric.
F17	**TYPE directive required** A TYPE directive is required.
F18	**duplicate TYPE directive** Duplicate TYPE directives are coded.
F19	**!branch table overflow!** The branch table has overflowed. This error occurs if there are too many B, BCAL, ORG, and NSE directives coded in the source program.
F20	**too many errors** There are too many errors.
F21	**too many include or macro nesting levels** Include or macro nesting levels are too deep.
F22	**expression too long** A constant expression is too long.

Code	Error Message
F23	**exist undefined symbol** There are one or more undefined symbols in the source file.
F24	**disk is full ?** The disk is full.

Chapter 9

Output Files

SASM64K creates four files as a result of assembly (object file, print file, error file, and assembly source file). This chapter describes these files.

9.1 Object File

The object file's contents are object code information and debug information.

Object code information is represented in Intel HEX format. Debug information is in the format used by SID64X and SID64K.

The file to which object code information and debug information are output has the name of the source program file with the extension ".HEX." (For example, if SAMPLE.ASM is assembled, then an object file called SAMPLE.HEX will be created.)

If there is one or more syntax errors in the source program, then an object file will not be created.

■ Attention ■

Debug information, used by SID64X and SID64K, is output if the /D option is specified when the assembler is started.

9.1.1 Object Code Information

Object code information is represented in Intel HEX format.

As shown in Figure 9-1, Intel HEX format is configured from two types of records: data records and an end-of-file record.

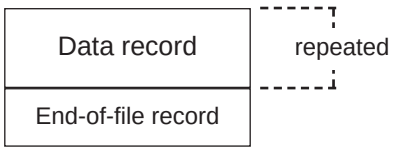
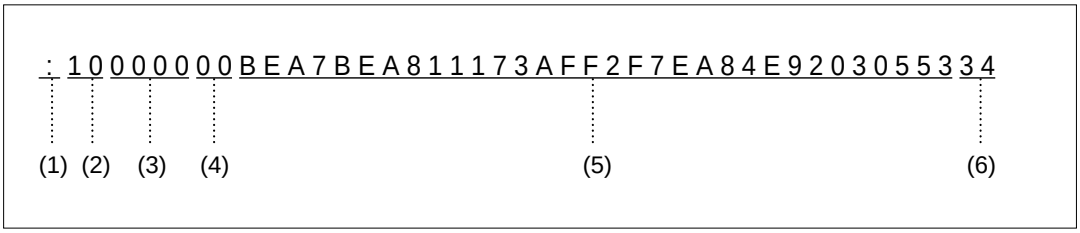


Figure 9-1. Intel HEX Format

Data records contain object code located in code space. The end-of-file record indicates the end of the file.

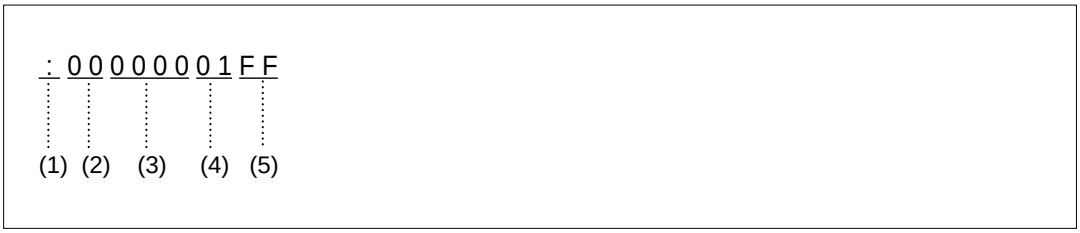
The format of each record is as follows.

■ Data Record



Field	Description
(1)	The character ":".
(2)	Number of bytes of object code stored in data field.
(3)	Address where object code stored at start of data field is to be loaded.
(4)	Fixed as "00". Indicates a data record.
(5)	Field containing object code.
(6)	Checksum.

■ End-of-File Record



Field	Description
(1)	The character ":".
(2)	Fixed as "00".
(3)	Fixed as "0000".
(4)	Fixed as "01". Indicates an end-of-file record.
(5)	Fixed as "FF".

9.1.2 Debug Information

Debug information, used by SID64X and SID64K, is output if the /D option is specified when the assembler is started.

As shown in Figure 9-2, debug information is configured from debug symbol records and end-of-debug information records.

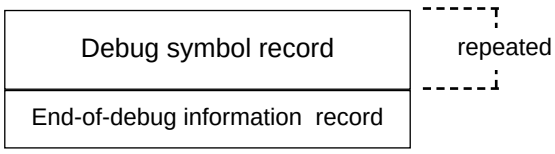
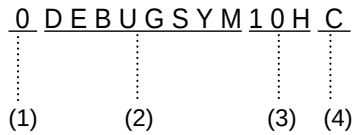


Figure 9-2. Debug Information Format

Debug symbol records contain information about user-defined symbols. The end-of-debug information record indicates the end of debug information.

The format of debug information independs on the format of object code information. The format of each record is as follows.

■ Debug Symbol Record



Field	Description
(1)	The character “0”. Indicates a debug symbol record.
(2)	User-defined symbol.
(3)	Value of the symbol.
(4)	Symbol type. Registered symbols are classified in the following types depending on how they were defined. C: Symbol is allocated in CODE space. D: Symbol is allocated in DATA space. N: Name was defined with EQU or SET directive.

■ End-of-Debug Information Record

```
$
```

This record indicates the end of debug information. It is the single character “$” (24H).

9.2 Print File

The print file is a variable length sequential file consisting of one record up to a carriage return. The file output to disk has the name of the source program file with the extension ".PRN."

```
<<SASM64K>> Structured-Macro-Assembler, Ver.1.00
 page :1 ..... (1)
 file :sample.ASM ..... (2)
 date :94 03/15 Tue.[18:19] ..... (3)
title :"Sample program for sasm64k" ..... (4)
Loc  Code  Statement        Line Source ..... (9)
 (5)   (6)      (7)            (8)

******************** sample.ASM ******************** ..... (10)
                                1:TYPE(M64152)
                                2:TITLE "Sample program for
                                  sasm64k"
                                3:
                                4:INCLUDE(SYMBOL.DEF)
******************** SYMBOL1.DEF ********************
                                1:NOLIST
******************** sample.asm ********************
                                5:
                                6:DEFINE RESET_DATA 0
                                7:
                                8:  ORG    100H
0100            ORG 0100H
                                9:START:
0100        START:
                               10:  HL = RESET_DATA
0100 50 00      LHLI  00H
                               11:  WHILE(NOSKIP X == 0)
0102        ?00000:
0102 43         LAX
0103 BE 40      CAI   00H
0105 C7         B     ?00002
0106 D0         B     ?00001
```

Field	Description
(1)	Page number.
(2)	Source file name.
(3)	Date given by DATE directive. If no DATE directive was used, then this field will display the current date from the operating system.
(4)	Title given by TITLE directive. If no TITLE directive was used, then this field will display nothing.
(5)	Location counter value (4-digit hexadecimal).
(6)	Object code.
(7)	Code segment statement. This field displays instructions, DB/DW/B/BCAL directives, label definitions in the code segment, and a converted form of NSE/ORG/DS directives. The calculated values of constant expressions are displayed.
(8)	Source file line number.
(9)	Source statement.
(10)	Source file name.

9.3 Cross-Reference List

The cross-reference list is a listing of both symbol information and a reference table. It shows where each symbol is defined and referenced.

Symbols are listed in alphabetic order. Symbols coded in the source program with lower-case letters will be output converted to all upper-case letters.

The cross-reference list is output following the assembly list.

```
---- cross reference list ----


BUFFER..................sample.ASM(10)
MAIN....................sample.ASM(17) ..... (2)
    ..... (1)      sample.ASM(20) ..... (3)
RAM0....................sample.ASM(6)
ROM0....................sample.ASM(7)
            sample.ASM(13)
VAL1....................sample.ASM(4)
VAL2....................sample.ASM(5)
```

Field	Description
(1)	Symbol.
(2)	File name and line number where symbol was defined (4-digit decimal).
(3)	File name and line number where symbol was referenced (decimal).

9.4 Symbol List

The symbol list is a listing of the symbol table contents. It provides information about the symbols that appear in the program.

The symbol list has the following format.

```
---- symbol information ----

name                          type  atr   value
   (1)                          (2)  (3)     (4)
BUFFER..................    LABEL DATA  0000H
MAIN....................    LABEL CODE  0100H
RAM0....................    DATA  DATA  0006H
ROM0....................    CODE  CODE  0004H
VAL1....................    EQU   NUM   0001H
VAL2....................    EQU   NUM   0000H
```

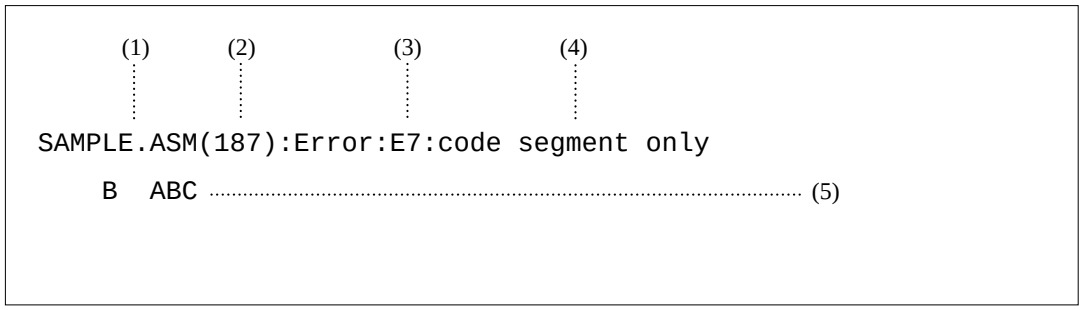
Field	Description
(1)	Symbol.
(2)	Directive that defined the symbol. However, LABEL will be displayed for labels.
(3)	Symbol type. Registered symbols are classified in the following types depending on how they were defined. CODE: Symbol is allocated in CODE space. DATA: Symbol is allocated in DATA space. NUM: Name was defined with EQU or SET directive.
(4)	Symbol value (4-digit hexadecimal).

9.5 Error File

The error file shows the statements that generated errors during assembly and the error messages.

First the error message is displayed. For pass 1, the source statement is then displayed. For pass 2 and part of pass 1, the source statement will not be displayed.

Refer to chapter 8, “Error Messages,” regarding the meanings of error messages. Error message output format is as follows.



Field	Description
(1)	Source file name.
(2)	Line number in source file.
(3)	Error number.
(4)	Error message.
(5)	Source statement.

9.6 Assembly Source File

The assembly source file is the SASM64K source file converted to a source file that can be assembled by ASM64K.

The assembly source file can be assembled by ASM64K Ver. 1.01 or higher, or by SASM64K itself. Be sure that the output file name is not a duplicate of the input file name when assembling with SASM64K.

An assembly source file output example is shown below.

```
;<<SASM64K>> Structured-Macro-Assembler, Ver.1.00
;file :sample.asm

TYPE(M64152)
TITLE "Sample program for sasm64k"

;sample.asm(4): INCLUDE(SYMBOL.DEF)
    DSEG
OPMODE_DATA    DATA   2
    CSEG
    ORG   80H
PLAY_SOUND_5:

;sample.asm(6): DEFINE RESET_DATA 0

;sample.asm(8): MACRO INC_BCD_2N()
;sample.asm(9):     BA = 7
;sample.asm(10):       [] == 9H
;sample.asm(11):       A = 1
;sample.asm(12):       BA += BYTE[]
;sample.asm(13):       BYTE[] = BA
;sample.asm(14):   ENDM

    ORG    100H
START:
;sample.asm(18):       HL = RESET_DATA
    LBAI   0
    LLAB

;sample.asm(19):       WHILE (NOSKIP X == 0)
?00000:
    LAH
    CAI    0
    B  ?00002
    B  ?00001
?00002:
```

Chapter 10

Sample Program

This chapter describes a sample application program using SASM64K.

10.1 Sample Program Specifications

This chapter introduces an application program for SASM64K. If you use all or part of this sample program, be sure to take into account other conditions and debug it properly.

10.1.1 Function of Sample Program

The sample program described in this chapter is a timer program that counts from 00:00 (minutes;seconds) to 99:59 in one-second increments, and displays the time on an LCD display.

10.1.2 Program Specifications

Minutes and seconds data are stored in two bytes in bank 7 of the data memory area (Figure 10-1). The COM1-4 and SEG0-7 LCD driver outputs are used for the LCD display (Figure 10-2).

To count time, the program checks the 1-Hz interrupt request flag in bank 0 of the data memory area (Figure 10-3). If the flag is 1, then the seconds data will be incremented in BCD (binary-coded decimal). If the seconds data has reached 60, then its carry will increment the minutes data in BCD.

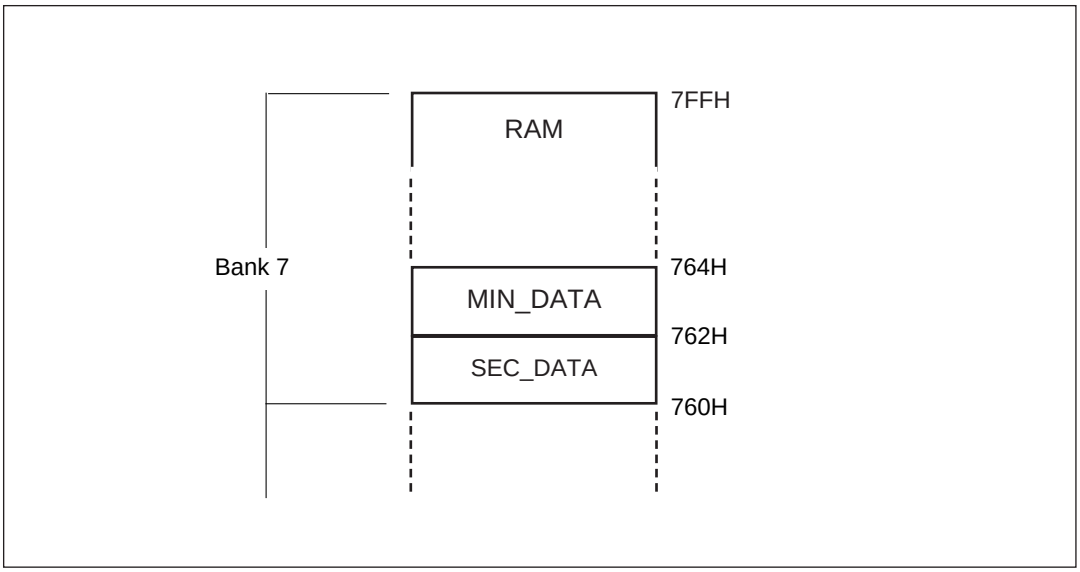


Figure 10-1. Seconds Data And Minutes Data

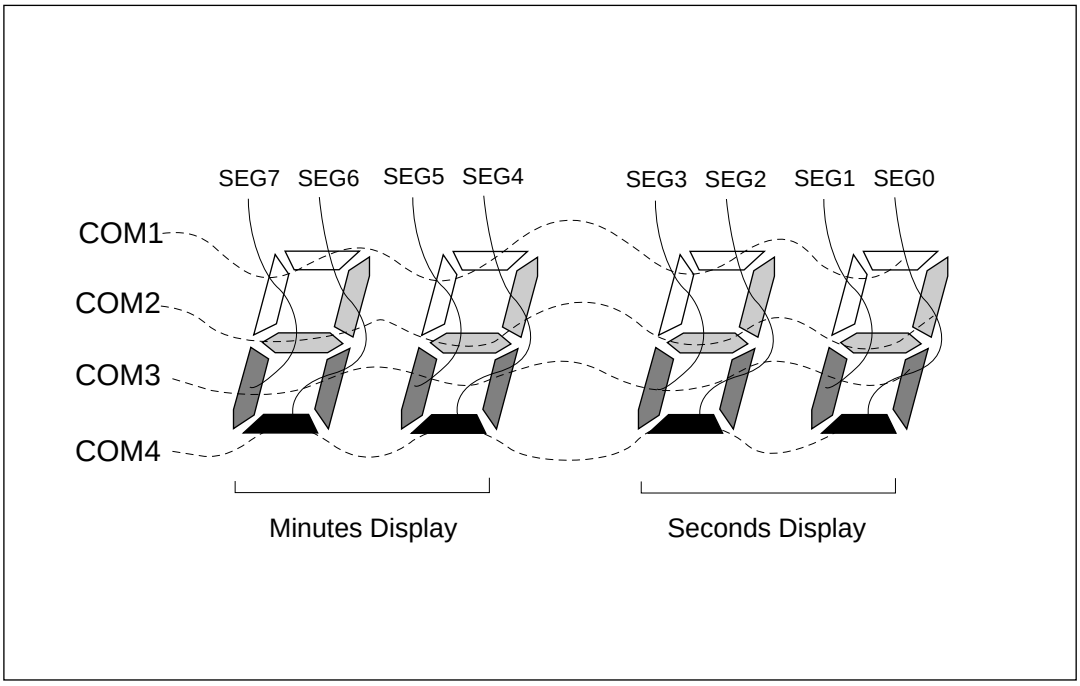


Figure 10-2 Correspondence Of LCD Driver Outputs And Seconds/Minutes Display

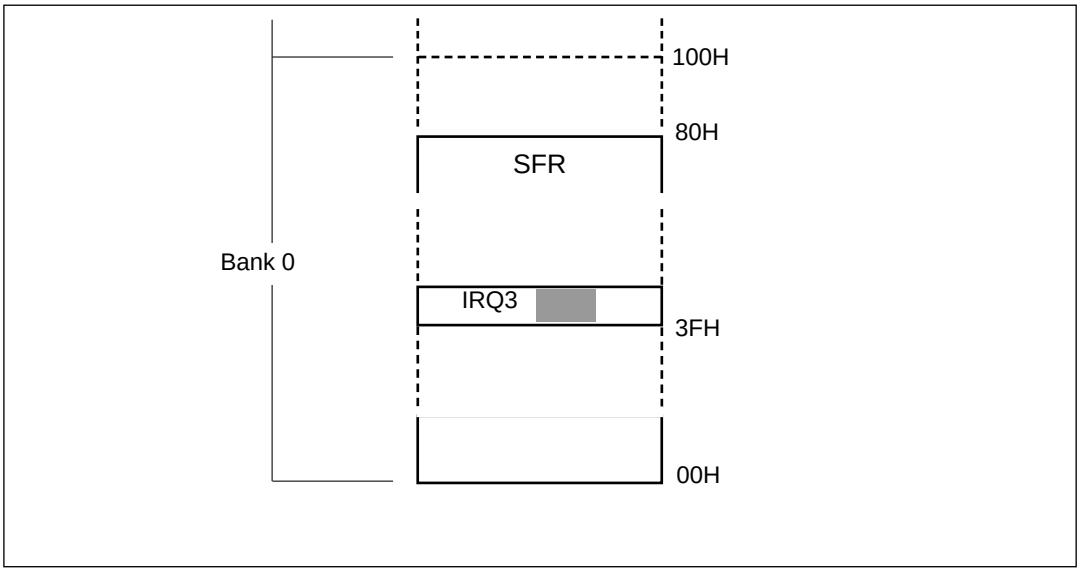


Figure 10-3 1-Hz Interrupt Request Flag

10.2 File Configuration

The sample program is configured from the following 7 files.

(1) MAIN.ASM
(2) SFRSBL.DEF
(3) DATSBL.DEF
(4) MACRO.DEF
(5) LCD.DEF
(6) SUB.DEF
(7) TABLE.DEF

Files (2) to (7) are all included in MAIN.ASM (Figure 10-4).

Each file is described below.

(1) MAIN.ASM

MAIN.ASM is the core program for processing. It specifies the target device with a TYPE directive, includes the sub-files, and codes the main procedure.

Lines 17-38 use a WHILE statement to construct an infinite loop that increments and displays the timer. Lines 40-50 are the display routine. The main procedure is coded using structured programming, one of SASM64K's feature.

(2) SFRSBL.DEF

SFRSBL.DEF defines SFR symbols.

(3) DATSBL.DEF

DATSBL.DEF defines data symbols.

(4) MACRO.DEF

MACRO.DEF defines macros. The user may find it convenient to collect macros in a library. The ability to define local labels eliminates the worry about defining the same label name twice.

(5) LCD.DEF

LCD.DEF defines LCD definition data.

(6) SUB.DEF

SUB.DEF defines subroutines. All subroutines are defined using the SUB directive, so if this file is included at the end of the MAIN.ASM, then only the subroutines actually referenced will be expanded. The user may find it convenient to collect subroutines in a library. The ability to define local labels eliminates the worry about defining the same label name twice.

(7) TABLE.DEF

TABLE.DEF defines table data.

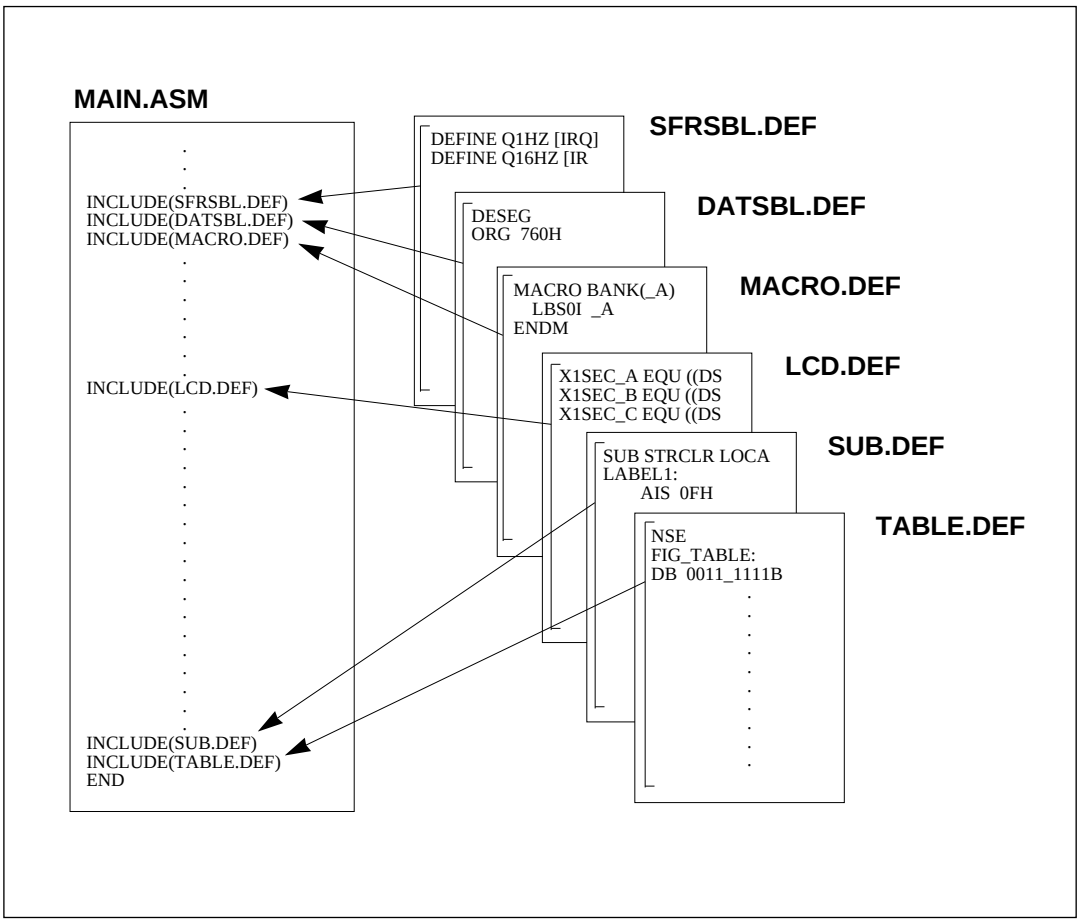


Figure 10-4 Program File Configuration

MAIN.ASM

```
 1: ;*****************************************************************
 2: ;******     SASM64K Sample Program                             *****
 3: ;******        for MSM64153                                    *****
 4: ;******                                                        *****
 5: ;******     Copyright 1994  OKI ELECTRIC INDUSTRY Co.,LTD.     *****
 6: ;*****************************************************************
 7: TYPE (M64153)
 8: TITLE "SASM64K Sample Program"
 9: INCLUDE(SFRSBL.DEF)
10: INCLUDE(DATSBL.DEF)
11: INCLUDE(MACRO.DEF)
12:     CSEG
13:     ORG    0H
14:     GOTO MAIN
15: INCLUDE(LCD.DEF)
16:     ORG    0E0H
17: MAIN:
18:     RBE
19:     RBC
20:     BANK(7)
21:     BYTE[offset SEC_DATA] = 00H
22:     BYTE[offset MIN_DATA] = 00H
23:     BCAL DSP_LCD
24:     WHILE(1)
25:     BANK(0)
26:         IF(Q1HZ)
27:             Q1HZ = 0
28:             BANK(7)
29:             HL = offset SEC_DATA
30:             BCAL BYTEINC_BCD
31:             IF(BYTE[]==60H)
32:                 BYTE[] = 00H
33:                 L = low MIN_DATA
34:                 BCAL BYTEINC_BCD
35:             ENDI
36:             BCAL DSP_LCD
37:         ENDI
38:     ENDW
39: DSP_LCD:
40:     BANK(0)
41:     STRN = 0
42:     BANK(7)
43:     DSPR_SET(high FIG_TABLE,[offset SEC_DATA]  ,DSPR2)
44:     DSPR_SET(     X        ,[offset SEC_DATA+1],DSPR4)
45:     DSPR_SET(     X        ,[offset MIN_DATA]  ,DSPR6)
46:     DSPR_SET(     X        ,[offset MIN_DATA+1],DSPR8)
47:     BANK(0)
48:     STRN = 1
49:     BANK(7)
50:     RT
51:
52: INCLUDE(SUB.DEF)
53: INCLUDE(TABLE.DEF)
54: END
```

SFRSBL.DEF

```
 1: DEFINE   Q1HZ      [IRQ3][1]
 2: DEFINE   Q16HZ     [IRQ2][3]
 3: DEFINE   Q32HZ     [IRQ2][2]
 4: DEFINE   Q128HZ    [IRQ2][1]
 5: DEFINE   E1HZ      [IE3][1]
 6: DEFINE   E16HZ     [IE2][3]
 7: DEFINE   E32HZ     [IE2][2]
 8: DEFINE   E128HZ    [IE2][1]
 9: DEFINE   DTRN1     [DSPCON0][3]
10: DEFINE   DTRN0     [DSPCON0][2]
11: DEFINE   ONOFF     [DSPCON0][1]
12: DEFINE   DUTY      [DSPCON0][0]
13: DEFINE   HTRN      [DSPCON1][1]
14: DEFINE   STRN      [DSPCON1][0]
```

DATSBL.DEF

```
 1:     DSEG
 2:     ORG    760H
 3: SEC_DATA:      DS 2
 4: MIN_DATA:      DS 2
```

MACRO.DEF

```
 1: MACRO BANK(_A)
 2:     LBS0I  _A
 3: ENDM
 4:
 5: MACRO DSPR_SET(_A,_B,_C)
 6:     X = _A
 7:     Y = _B
 8:     HL = _C
 9:     BANK(0)
10:     LMTB  bnk FIG_TABLE
11:     BANK(7)
12: ENDM
```

LCD.DEF

```
 1: X1SEC_A   EQU ((DSPR2-40H)<<2)+0
 2: X1SEC_B   EQU ((DSPR2-40H)<<2)+1
 3: X1SEC_C   EQU ((DSPR2-40H)<<2)+2
 4: X1SEC_D   EQU ((DSPR2-40H)<<2)+3
 5: X1SEC_E   EQU ((DSPR3-40H)<<2)+0
 6: X1SEC_F   EQU ((DSPR3-40H)<<2)+1
 7: X1SEC_G   EQU ((DSPR3-40H)<<2)+2
 8: X10SEC_A  EQU ((DSPR4-40H)<<2)+0
 9: X10SEC_B  EQU ((DSPR4-40H)<<2)+1
10: X10SEC_C  EQU ((DSPR4-40H)<<2)+2
11: X10SEC_D  EQU ((DSPR4-40H)<<2)+3
12: X10SEC_E  EQU ((DSPR5-40H)<<2)+0
13: X10SEC_F  EQU ((DSPR5-40H)<<2)+1
14: X10SEC_G  EQU ((DSPR5-40H)<<2)+2
15: X1MIN_A   EQU ((DSPR6-40H)<<2)+0
16: X1MIN_B   EQU ((DSPR6-40H)<<2)+1
17: X1MIN_C   EQU ((DSPR6-40H)<<2)+2
18: X1MIN_D   EQU ((DSPR6-40H)<<2)+3
19: X1MIN_E   EQU ((DSPR7-40H)<<2)+0
20: X1MIN_F   EQU ((DSPR7-40H)<<2)+1
21: X1MIN_G   EQU ((DSPR7-40H)<<2)+2
22: X10MIN_A  EQU ((DSPR8-40H)<<2)+0
23: X10MIN_B  EQU ((DSPR8-40H)<<2)+1
24: X10MIN_C  EQU ((DSPR8-40H)<<2)+2
25: X10MIN_D  EQU ((DSPR8-40H)<<2)+3
26: X10MIN_E  EQU ((DSPR9-40H)<<2)+0
27: X10MIN_F  EQU ((DSPR9-40H)<<2)+1
28: X10MIN_G  EQU ((DSPR9-40H)<<2)+2
29: ORG    50H
30: DB 00         ,X10MIN_E       ,X10MIN_G       ,X10MIN_F
31: DB X10MIN_D   ,X10MIN_C       ,X10MIN_B       ,X10MIN_A
32: DB 00         ,X1MIN_E        ,X1MIN_G        ,X1MIN_F
33: DB X1MIN_D    ,X1MIN_C        ,X1MIN_B        ,X1MIN_A
34: DB 00         ,X10SEC_E       ,X10SEC_G       ,X10SEC_F
35: DB X10SEC_D   ,X10SEC_C       ,X10SEC_B       ,X10SEC_A
36: DB 00         ,X1SEC_E        ,X1SEC_G        ,X1SEC_F
37: DB X1SEC_D    ,X1SEC_C        ,X1SEC_B        ,X1SEC_A
38: DB 0FFH
```

SUB.DEF

```
 1: SUB STRCLR LOCAL(LABEL1)
 2: LABEL1:
 3:     AIS    0FH
 4:     RT
 5:     [] = 0H
 6:     NS,HL ++
 7:     GOTO  LABEL1
 8: ENDSUB
 9:
10: SUB BYTEINC_BCD
11:     IF(BYTE[]==99H)
12:         BYTE[] = 00H
13:     ELSEIF([]==9H)
14:         BYTE[] += 07H
15:     ELSE
16:         NS,[] ++
17:     ENDI
18:     RT
19: ENDSUB
20:
21: SUB BYTEDEC_BCD
22:     IF(BYTE[]==00H)
23:         BYTE[] = 99H
24:     ELSEIF([]==0H)
25:         BYTE[] -= 07H
26:     ELSE
27:         [] --
28:     ENDI
29:     RT
30: ENDSUB
```

TABLE.DEF

```
 1: NSE
 2: FIG_TABLE:
 3: DB 0011_1111B    ;0
 4: DB 0000_0110B    ;1
 5: DB 0101_1011B    ;2
 6: DB 0100_1111B    ;3
 7: DB 0110_0110B    ;4
 8: DB 0110_1101B    ;5
 9: DB 0111_1101B    ;6
10: DB 0000_0111B    ;7
11: DB 0111_1111B    ;8
12: DB 0110_1111B    ;9
```

Appendices

- Basic Instructions
- Directives
- Operators
- Spscial Assembler Symbols
- Control Statements
- Data Address Symbols

Basic Instructions

ADC	ADCB	ADCS	ADS	ADSB	AIS	AND
ANDI	CAB	CAI	CAL	LCAL	CAM	CAMB
CAMD	CLI	CMA	CMI	CZP	DCA	DCH
DCL	DCM	DCMD	DCX	DCY	EOR	EORI
INA	INH	INL	INM	INMD	INX	INY
JA	JCP	JM	JP	LJP	LAB	LAH
LAI	LAL	LALB	LAM	LAMB	LAMD	LAMDB
LAMM	LAM+	LAM–	LAX	LAY	LAYB	LBA
LBAI	LBS0I	LBS1I	LHA	LHI	LHLI	LLA
LLAB	LLI	LMA	LMAB	LMAD	LMADB	LMA+
LMA–	LMBI	LMI	LMTB	LXA	LXI	LXYI
LYA	LYAB	LYI	NOP	OR	ORI	POP
PUSH	RAL	RAR	RBC	RBE	RC	RMB
RMBD	RT	RTI	RTS	SBC	SBE	SC
SMB	SMBD	SUBC	SUBCB	SUBCS	SUBS	SUBSB
TAB	TC	TMB	TMBD	XAB	XAM	XAMB
XAMD	XAMDB	XAMM	XAM+	XAM–		

Directives

B	BCAL	CODE	CSEG	DATA
DATE	DB	DEFINE	DS	DSEG
DW	END	ENDM	ENDSUB	EQU
ERR	GOTO	INCLUDE	LIST	LOCAL
MACRO	NOERR	NOLIST	NOOBJ	NOPRN
NOREF	NOSYM	NSE	OBJ	ORG
PAGE	PRN	REF	REFER	SET
SUB	SYM	TITLE	TYPE	

Operators

*	/	%	+	-	≪	≫
&	|	^	>=	>	<	<=
~	!	= =	!=	& &	‖	<>
+ =	- =	& =	|=	^=	OFFSET	BNK
LOW	HIGH					

Special Assembler Symbols

A	B	BA	BSR	BYTE
C	H	HL	L	NIBBLE
NOSKIP	NS	SKIP	X	XY
Y	@	$		

Control Statements

BREAK	CASE	CONTINUE	DEFAULT
FOR	IF	ELSE	ELSEIF
ENDF	ENDI	ENDS	ENDW
REPEAT	SWITCH	WHILE	UNTIL

Data Address Symbols

Refer to the following file.

- README.JPN (Japanese version)
- README.ENG (English version)