OKI

SASM63K

User's Manual

Program Development Support Software

First Edition

August 1997

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 SASM63K, a structured assembler for the OLMS-63K series of 4-bit single-chip microcontrollers. The contents of this manual are written for developers with some experience using assembly language.

SASM63K operates under MS-DOS. It is supplied on a floppy disk.

Typographical conventions

Symbol	Meaning
CAPITALS	Items appearing in all capitals are to be typed in exactly as given.
italics	Items appearing in italics are not to be typed as given, but rather replaced with values giving the necessary information.
[ ]	The contents of the brackets are optional and may be omitted.
. . .	The item preceding the ellipsis may be repeated as many times as necessary.
value1 ~ *value2*	The tilde indicates a range spanning all values between the designated endpoints.
PROGRAM . . . PROGRAM	Vertically aligned dots indicate a program segment that has been omitted.
n2	This notation indicates a constant expression with a value from 0 to 3.
n4	This notation indicates a constant expression with a value from 0 to 15.
n8	This notation indicates a constant expression with a value from 0 to 255.
n16	This notation indicates a constant expression with a value from 0 to 65535.

TABLE OF CONTENTS

Chapter 1. Introduction

Chapter 2. Starting SASM63K

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 Reserved Words

Chapter 1

Introduction

This chapter describes the assembler's functions.

1.1 Functional Overview

SASM63K is a structured assembler for the OLMS-63K series of 4-bit single-chip microcontrollers. A structured assembler accepts a new type of assembly language that combines the coding ease of high-level languages with the high coding efficiency of assembly language. This combination greatly raises overall application program development efficiency.

SASM63K has the following features.

● **Replaces the ASM63KN assembler**

SASM63K is upward compatible with the ASM63KN assembler. It assembles all source programs previously written for ASM63KN. Programmers can incorporate the new features of SASM63K a few at a time into their previous programming style for an easy transition to SASM63K programming.

● **Adds extended instructions (SASM instructions)**

Extended instructions are special macros that combine native chip instructions to enhance working with data. An extended instruction can, for example, express a memory-to-memory transfer with a single statement. The coding style is also similar to high-level languages, so programs are much easier to read and understand.

● **Supports flow control blocks**

Programs can use the same IF, WHILE, and other flow control statements available in high-level languages for a structured programming approach that makes programs easier to maintain and update.

● **Adds preprocessor directives**

SASM63K adds preprocessor directives for macros and include files. Programmers can therefore define their own macros.

SASM63K assembles source files making reference to the contents of DCL files. A DCL file contains device-specific information for a particular microcontroller. Changing DCL files is all it takes to adapt SASM63K for a different member of the OLMS-63K series. A source file is a program written in OLMS-63K series assembly language.

The assembler's basic function is translating the mnemonic codes written in the source file into object code. These mnemonic codes are symbolic instructions assigned to individual machine language instructions.

SASM63K produces the following files from the source file: object files, a listing file, an error file, and an assembly source file.

The object files consist of two byte-divided HEX files containing the object code and Intel HEX format files containing the initialization data for external memory. (For further information on byte-divided HEX files and Intel HEX format files, see Chapter 9 "Output Files.")

The listing 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. In the absence of any specification to the contrary, this file goes to standard output, the screen.

The assembly source file contains the source code with preprocessor directives and extended instructions expanded. It may be assembled with ASM63KN Ver. 1.01 or higher as well as with SASM63K itself.

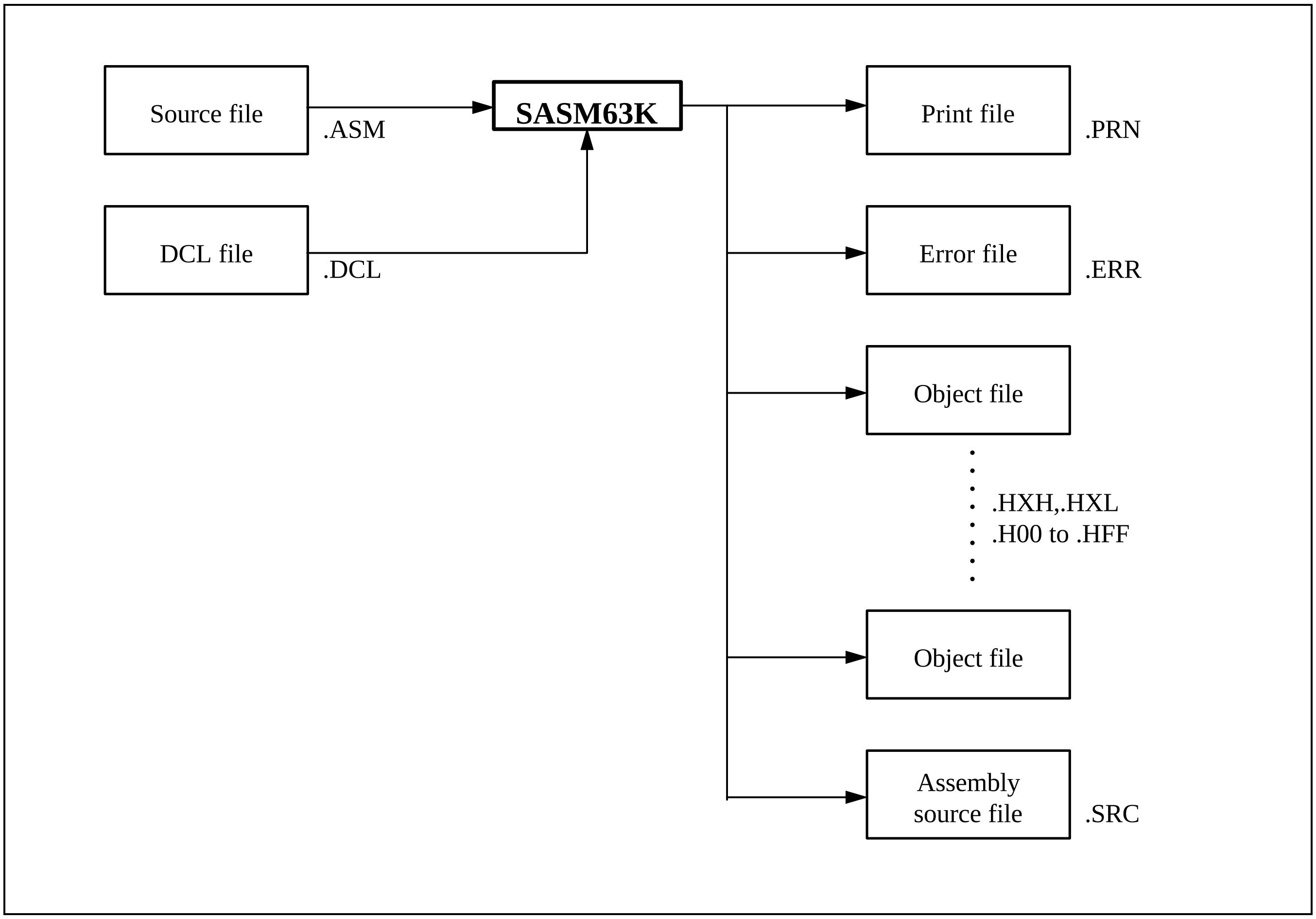


Figure 1.1 Input/Output Flow

1.2 Sample Program

Here is a small sample SASM63K program to start with.

```
TYPE (M63188)
TITLE "SASM63K Sample Program"

INCLUDE(SYMBOL.DEF)

DEFINE RESET_DATA 0H

MACRO INC_BCD()
        IF ( [] == 9 )
                [+] = 0
                IF ( [] == 9 )
                        [] = 0
                ELSE
                        []++
                ENDI
        ELSE
                []++
        ENDI
ENDM

        ORG     100H
MAIN :
        [CBR] = 15
        [\COUNT_DATA]   = RESET_DATA
        [\COUNT_DATA+1] = RESET_DATA
        CAL     DSP_LCD
        WHILE ( TRUE )
                IF ( _Q2HZ )
                        _Q2HZ = FALSE
                        HL = COUNT_DATA & 0FFH
                        INC_BCD()
                        CAL     DSP_LCD
                ENDI
        ENDW
```

This program uses many SASM instructions instead of the CPU's basic instructions. The [\ COUNT_DATA]=RESET_DATA on line 24 is one of these. This instruction assigns the value RESET_DATA to the address COUNT_DATA in the current bank. It corresponds to the basic instruction MOV 0F00H,#0H. SASM instructions are close to high-level language code, so programs are easier to write and read. Some SASM instructions expand into multiple basic instructions.

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

The macro definition on lines 8 to 19 and the macro call on line 31 are also preprocessor directives. In this example, the symbol INC_BCD is defined as a macro. At the point that it is called in line 31, it is expanded to the contents of lines 9 to 18. Macros can also take parameters and declare local labels.

The WHILE statement block on lines 27 to 34 causes the inner statements to be executed until the specified condition is no longer satisfied. Statements that like the WHILE statement control program flow are called flow control statements. This sample program contains an additional flow control statement, the IF statement block on lines 28 to 33.

1.3 DCL Files

A DCL file contains device-specific information for a particular microcontroller. Changing DCL files configures SASM63K for a different member of the OLMS-63K series. DCL files are text files. The DCL file must be specified with the TYPE directive.

1.3.1 File Name

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

DCL file name = devicename.DCL

1.3.2 DCL File Search

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

1. Search the directory contained in the DCL file specification in the TYPE directive (normally the current directory).

2. Search the directories contained in the PATH environment variable. If the DCL file specification contains an explicit path, however, this search is not performed.

Below are examples. Assume that the PATH environment variable has been defined as follows.

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

■ **Example 1** ■ TYPE (M63XXX) is specified.

1. The assembler searches for M63XXX.DCL in the current directory.

2. If (1) fails to find the file, the assembler searches in the following order based on the PATH environment variable.

   A:\BIN\M63XXX.DCL
   A:\M63XXX.DCL
   DCL\ M63XXX.DCL

■ **Example 2** ■ TYPE (DCL\ M63XXX) is specified .

1. The assembler searches for M63XXX.DCL in the DCL subdirectory of the current directory.

2. If (1) fails to find the file, the search terminates because the DCL file specification contains an explicit path specification (DCL\ ).

1.3.3 DCL File Contents

A DCL file contains the following device-specific information about the microcontroller.

(1) Program memory available

SASM63K uses this information to check the values of operands accessing the program memory space.

(2) Data memory available

SASM63K uses this information to check the values of operands accessing the data memory space and the bit address space. It also checks accesses to the SFR area to see whether the target addresses are, in fact, accessible.

There are the following types of information pertaining to data memory.

● Data area of target device
● SFR area of target device

(3) External memory available

SASM63K uses this information to check the values of operands accessing the external memory space.

(4) SFR area access attributes

SASM63K uses this information to check accesses to the SFR area.

(5) Data address symbols

These are predefined symbols specifying addresses to SASM63K. They may be used in place of addresses in operands.

(6) Permitted instruction mnemonics

SASM63K recognizes only the instruction mnemonics specified in the DCL file. Other instructions produce errors.

1.3.4 DCL63K.DOC

This manual covers all microcontrollers in the OLMS-63K series, referring the reader to the corresponding DCL file for device-specific information for the individual microcontroller.

The file DCL63K.DOC describes the information contained in the DCL file.

■ Note ■

Never attempt to rewrite the contents of a DCL file. Assembler results are not guaranteed if source files are assembled using a DCL file that has been modified.

1.3.5 Error Processing

The DCL file is processed in the assembler's first phase. This processing continues through to the end of the DCL file regardless of any errors. If the assembler detects an error during this processing, it displays an error message on the screen and continues processing the DCL file. Once it has finished processing the DCL file, it checks for errors.

If there have been any errors at all, the assembler aborts without performing any further processing. If there are no errors, it goes on to process the source file.

1.4 OLMS-63K Series Memory Spaces

The OLMS-63K series has three memory spaces:

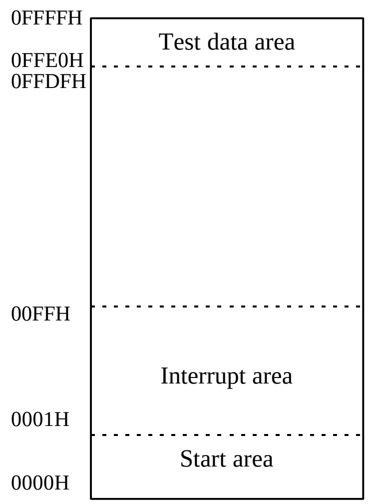
- Program memory
- Data memory
- External memory

Program memory is addressed in terms of 16-bit words; data memory, in terms of nybbles or bits; the external memory, in terms of bytes.

The address ranges for each memory space vary with the target microcontroller. (See Section 1.2 "DCL Files.")

1.4.1 Program Memory

The following shows the program memory space for the OLMS-63K series.

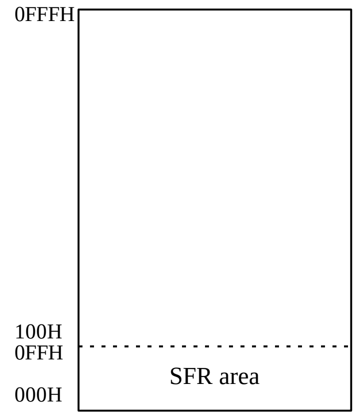


Sample Program Memory Space

For the address ranges for these areas, refer to the hardware manual for the target microcontroller.

1.4.2 Data Memory

The following shows the data memory space for the OLMS-63K series.

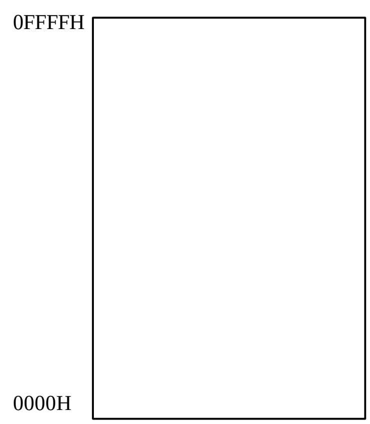


Sample Data Memory Space

For the address ranges for these areas, refer to the hardware manual for the target microcontroller.

1.4.3 External Memory

The following shows the external memory space for the OLMS-63K series.



Sample External Memory Space

For the address ranges for these areas, refer to the hardware manual for the target microcontroller.

1.4.4 Expanding External Memory

The OLMS-63K series supports an external memory space of 64 kilobytes. Some application programs, however, require more. One way of providing additional external memory space is to adapt the target microcontroller ports for use in expanding addresses.

SASM63K considers 64 kilobytes as a single bank. These banks are called external memory banks to distinguish them from data memory banks.

SASM63K provides the following functions for supporting the expansion of the external memory space to 16 megabytes.

- It generates separate HEX files for each external memory bank.

- The ORG directive includes support for specifying the bank for external memory bank initialization code. See Section 4.3.1 "ORG."

- Address symbols may be defined in external memory banks. See Section 4.1.6 "XDATA."

- The XBANK operator gives the external bank number for address symbols defined with the XDATA directive and labels in the external memory areas. See Section 3.5.6 "Special Operators.")

■ **Example** ■

The following Figure gives an example of an expanded memory configuration. The example uses Port B (PB) as the external memory bank selector.

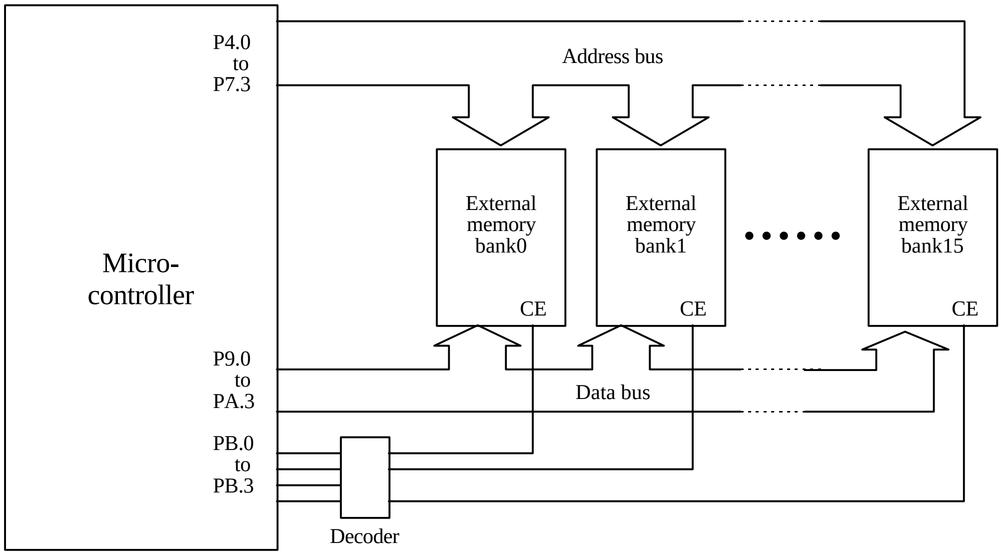


Figure 1.2 External Memory Expansion Example

The following code fragment shows how to access this expanded external memory.

```
        XSEG
        ORG     0H      ; Bank 0
XSYM00: DB      10H     ; Data specification
XSYM01: DB      20H     ;
XSYM02: DB      30H     ;
        .
        .
        .
        ORG     1:0H    ; Bank 1
XSYM10: DB      40H     ; Data specification
XSYM11: DB      50H     ;
XSYM12: DB      60H     ;
        .
        .
        .
        CSEG
        MOV     A,#XBANK XSYM11 ; Switch to external memory bank 1
        MOV     PB,A            ;
;
        MOVXB   [HL],XSYM11     ; Move the 50H value from byte 01H in the external memory
                                ; bank to the address in data memory contained in the HL
                                ; register.
```

1.5 Address Spaces and Segments

SASM63K divides memory spaces according to how their addresses are assigned. These logical spaces are called address spaces.

There are the following types of address spaces.

Address Spaces

Address Space	Description
CODE address space	Program memory space, addressed in word (16-bit) increments
DATA address space	Data memory space, addressed in nybble increments
BIT address space	Data memory space, addressed in bit increments
XDATA address space	External memory space, addressed in byte increments

The XDATA address space exists only when external memory is physically present.

When writing a program, the user must tell the assembler which address space the current portion of the program is using. The procedure uses segments, which are areas of contiguous addresses.

SASM63K retains the segment type as an attribute of each symbol defined in these address spaces.

The following summarizes the relationship between address space and segment type.

Address Space and Segment Type

Segment Type	Address Space
CSEG (CODE segment)	CODE address space
DSEG (DATA segment)	DATA address space
BSEG (BIT segment)	BIT address space
XSEG (XDATA segment)	XDATA address space

Chapter 2

Starting SASM63K

This chapter describes how to start SASM63K.

2.1 Starting Methods

There are two methods for starting SASM63K.

Method 1: `SASM63K` file_name [options]

Method 2: `SASM63K`

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 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.

`SASM63K` file_name [options]

The assembler loads and immediately starts assembly.

2.1.2 Starting Method 2

Type the following at the operating system prompt.

`SASM63K`

The assembler loads and displays its command prompt, an asterisk (*), on the console. Type the file name and options at this prompt.

file_name [options]

After this input, the assembler starts assembly.

Entering a blank response to this prompt displays a usage screen for the assembler. The assembler does not start assembly.

2.2 File Specifications

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

SASM63K allows files to be specified with the hierarchical directory structure supported by the operating system. The following is the general format for such specifications.

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

The maximum length for a single file specification is 50 characters. All characters beyond this limit are ignored.

The entire file name specification is not necessary; parts may be omitted. Table 2-1 gives the defaults that SASM63K uses for missing drive names, directories, file names, and extensions.

Table 2-1. File Types and Defaults

	Item			
File Type	**Drive**	**Directory**	**File Name**	**Extension**
Source file	Current drive	Current directory	No default	.ASM
Print file	Current drive	Current directory	Same as source file name	.PRN
Error file	Current drive	Current directory	Same as source file name	.ERR
Object file	Current drive	Current directory	Same as source file name	.HXH .HXL .H00 to HFF
Assembly source file	Current drive	Current directory	Same as source file name	.SRC

For example, if the source file specification is a file name including an extension (Figure 2-1(1)), SASM63K recognizes that as the file name. If there is no extension (Figure 2-1(2)), SASM63K assumes the extension .ASM. If the file name ends in a period (Figure 2-1(3)), SASM63K assumes a file name with no extension.

Specification		Interpretation
(1) TEXT.SRC	→	TEXT.SRC
(2) TEXT	→	TEXT.ASM
(3) TEXT.	→	TEXT

Figure 2-1 Interpretation of File Specifications

2.3 Options

SASM63K offers a variety of command line options for controlling assembler functions. All options start with a slash (/). This slash is followed by the option name. There must be no space between the slash and the option name. The option name can be in either upper or lower case. These options allow the input of some of the directives described in Section 4.5 "Listing Control" from the command line. The format differs, but the functions are exactly the same as the corresponding directives.

Options take precedence over directives. SASM63K does not generate an error message when an option overrides a directive in the source file.

As many options as needed may be specified in any order. Multiple options must be separated with one or more spaces or tabs.

Table 2-2 lists the options available for SASM63K. An asterisk in the default column indicates the default used when neither the option or its corresponding directive appears.

Table 2-2 Options

Option	Default	Corresponding directive	Function
/O [(file)]	*	OBJ	Generates object files
/NO		NOOBJ	Suppresses generation of object files
/PR [(file)]		PRN	Generates print file
/PR1 [(file)]		—	Generates print file. See Section 9.2 “Print File”
/NPR	*	NOPRN	Suppresses generation of print file
/S		SYM	Generates symbol list
/NS	*	NOSYM	Suppresses generation of symbol list
/R		REF	Cross reference list
/NR	*	NOREF	Suppresses generation of cross reference list
/E [(file)]		ERR	Redirects error message list to file
/NE	*	NOERR	Directs error message list to screen
/D		DEBUG	Generates debug information
/ND	*	NODEBUG	Suppresses generation of debug information
/A		—	Generates assembler source file
/NA	*	—	Suppresses generation of assembler source file

2.4 Exit Codes

SASM63K terminates by returning an exit code, a value indicating the termination state, to the operating system. The following Table lists the possibilities. An exit code of 0 or 1 indicates that the assembler processed the source file until it encountered an END directive or an end of file.

Table 2-3 Termination Codes

Termination code	Termination state
0	No errors
1	Error(s) in assembly
2	Abort due to fatal error

2.5 Examples of Starting SASM63K

This section illustrates the two starting methods described above, taking as an example assembly of the source file TEXT.ASM.

These examples assemble the source file TEXT.ASM in the directory \USR\MYDIR, generating listing and error files, but not generating a cross reference list.

■ **Example 1** ■ Starting Method 1

```
A>SASM63K    \USR\ MYDIR \TEXT /PR /E /NR
SASM63K Structured Macro Assembler, Ver.1.00
Copyright(c) 1995 Oki Electric Ind.Co.,Ltd., ALL RIGHTS RESERVED

pass1...
pass2...

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

■ **Example 2** ■ Starting Method 2

```
A>SASM63K
SASM63K Structured Macro Assembler, Ver.1.00
Copyright(c) 1995 Oki Electric Ind.Co.,Ltd., ALL RIGHTS RESERVED
*\USR\ MYDIR \TEXT /PR /E /NR

pass1...
pass2...

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

Chapter 3

Assembly Language Syntax

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

3.1 Characters Allowed in Programs

Source programs for SASM63K can use the following characters.

Letters: ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz

Digits: 0123456789

Symbols: ! ” # $ % & ’ ( ) * + , - . / : ; < = > ? @ [ ] ^ ~ \ |
tab space _ (underscore)

SASM63K accepts lower case letters, but internally converts them to upper case before processing. "TELEX" and "telex," for example, refer to the same symbol.

3.2 Structual Elements of Source Programs

An SASM63K 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 of spaces and tabs plus the carriage return.

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 symbols, operators, etc. They have no other special meaning.

The rest of this section describes the individual statement types.

3.2.1 Instruction Statements

SASM63K supports two types of instructions. Instructions found in the CPU's native instruction set are called basic instructions. Instructions that SASM63K expands to one or more basic instructions during assembly are called SASM instructions. Any instruction statement can have a label definition at the beginning. For further details, see Section 3.3 "Statement Format for Basic Instructions" and Chapter 5 "SASM Instructions."

3.2.2 Directive Statements

Directive statements give instructions to the assembler itself. Preprocessor instructions are included in this category. For further details, see Chapter 4 "Directives."

3.2.3 Control Statements

Control statements, or control blocks, control program flow. A control block consists of two or more statements. For details, see Chapter 7 "Control Statements."

3.3 Statement Format for Basic Instructions

Basic instruction statements consist of four fields: label definition, instruction, operand(s), and comment. The following gives the general format.

<u>LABEL:</u>	<u>MOV</u>	<u>10H,A</u>	<u>;comment</u>
label definition	instruction	operands	comment

Actual statements do not necessarily need to have all four fields present. The fields can start from any column, but must appear in the order given.

The rest of this section describes the individual fields.

3.3.1 Label Field

The label field defines a symbol whose value is the current address. A colon (:) must always follow the symbol.

This symbol may be referenced from anywhere in the program.

3.3.2 Instruction and Operand Fields

These fields code the basic instruction. When an operand field follows the instruction field, the two are delimited by spaces and tabs. A single statement cannot contain more than one instruction.

3.3.3 Comment Field

The comment field begins 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, instructions, and registers. They are broadly divided into user-defined symbols defined in the program and reserved word symbols predefined by SASM63K.

3.4.1 Reserved Word Symbols

Reserved word symbols are ones predefined by SASM63K. They include the following categories.

- Instructions
- Directives
- Registers
- Operators
- Device-specific addresses
- Special assembler symbols
- Special directive operands
- Symbols beginning with question marks

These reserved word symbols do not distinguish between upper and lower case. They are available to the user without having to be defined. They can only be used for their predefined purpose—that is, they cannot be used as labels or be redefined with directives for defining symbols.

The Appendix lists all reserved words other than those for device-specific addresses.

The rest of this section describes the individual reserved word categories.

3.4.1.1 Instructions

These are the microcontroller instructions recognized by the SASM63K assembler. For functional descriptions of these instructions, refer to the corresponding manual.

3.4.1.2 Directives

These are the directives for controlling the SASM63K assembler. For functional descriptions, see Chapter 4 "Directives."

3.4.1.3 Registers

These symbols stand for registers. They are used as operands to microcontroller instructions. They differ from the reserved words, described below, for indicating register addresses in that they indicate the registers themselves—that is, they do not have addresses. This category has the following members.

A	FLAG	HL	XY	RA
SP	PC			

3.4.1.4 Operators

These symbols stand for operators. They are used in expressions. For details on function and use, see Section 3.6 "Operators." This category has the following member.

XBANK

3.4.1.5 Device-Specific Addresses

These symbols stand for addresses specific to the target microcontroller. These correspond to the registers in the SFR area and bits in registers. These reserved words are defined in the DCL files.

The following address symbols are common to all microcontrollers in the OLMS-63K series.

H	L	X	Y	RA0
RA1	RA2	RA3	CBR	EBR

For other symbols, refer to the DCL file. For a description of the DCL file, refer to the file DCL63K.DOC.

3.4.1.6 Special Instruction Operands

These include the flag names used as instruction operands and addressing specifiers. The latter are specifiers that indicate the addressing type. For their specific meanings and specification methods, refer to the corresponding manual. This category has the following members.

C	E	Z	G

3.4.1.7 Special Directive Operands

These are symbols with special meanings in the context of directive operands. For their meaning and use, see Chapter 4 "Directives." This category has the following members.

BANK ANY

3.4.1.8 Symbols Starting with a Question Mark (?)

These are reserved words because SASM63K uses this type of symbol internally.

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 $? _

In order to distinguish user-defined symbols from integer constants, however, the first character cannot be a digit.

There is no restriction on the length of symbols, but only the first 31 characters are valid. All characters past that point are ignored.

User-defined symbols are defined as labels or with symbol definition directives. The following Table summarizes the methods for defining symbols.

Table 3-1 Methods for Defining User Symbols

User-defined symbol	Definition methods
User-defined symbol representing a number	EQU and SET directives
User-defined symbol representing an address	Label, CODE directive, DATA directive, BIT directive, XDATA directive

The definition for a user-defined symbol includes a value and, for one representing an address, a segment type. This segment type is CODE, DATA, BIT, or XDATA depending on the address space the symbol is defined in. User-defined symbols representing numbers do not have segment types.

Labels are defined as addresses. They must be followed by a colon (:) when defined.

Reserved words cannot be used as user-defined symbols.

Correct examples	Incorrect examples
_LOOP:	DATA: .... Same as directive
LOOP_1:	1ABC: ....First character is digit
$XYZ:	

3.4.3 Location Counter Symbol

SASM63K constantly tracks the address in the segment being assembled. The counter containing this address is called the location counter. The dollar sign ($) is a special symbol giving the location counter value for the current segment. It is called the location counter symbol.

■ **Example** ■

The following is an example of the use of the location counter symbol.

```
CSEG
JMP     $          ; Infinite loop
```

3.4.4 Symbol Scope and Overlapping Definitions

A symbol can normally be defined only once within a single file. The only exceptions are symbols defined and redefined with the SET directive. A symbol defined with the DEFINE directive can only be redefined if the new definition string exactly matches the old.

Parameters and local symbols defined within a macro definition are only valid within that macro. Ones defined within another macro are treated as separate symbols even if they have the same names.

3.5 Constants

3.5.1 Integer Constants

■ **Syntax** ■

```
ddigits
hdigitsH
odigitsO
odigitsQ
bdigitsB
```

■ **Function** ■

Integer constants are integers within the range expressible with 16 bits. Binary, octal, decimal, and hexadecimal representations are all supported. The radix is given by a suffix appended to the digits. Integers without a radix suffix are considered decimal.

hdigits is a string of hexadecimal digits; *ddigits*, a string of decimal digits; *odigits*, a string of octal digits; *bdigits*, a string of binary digits. To distinguish them from symbols, integer constants must start with a digit between 0 and 9. A hexadecimal integer starting with a letter (A-F) must therefore be prefixed with a zero.

For enhanced readability, underscores may be inserted anywhere within the string. They cannot, however, be used at the beginning of the string.

Table 3-2 Radix Specifiers

	Radix specifier	**Permissible characters**
Hexadecimal	H , h	0123456789ABCDEFabcdef_
Decimal		0123456789_
Octal	O , o , Q	01234567_
Binary	B , b	01_

The radix specifier can be upper or lower case. Hexadecimal digits can also be upper or lower case.

■ Example ■

The decimal number 256 takes the following forms when written as a hexadecimal, octal, and binary number.

	Notation
Hexadecimal	100H
Decimal	256
Octal	400O　　400Q
Binary	100000000B

Adding zeros to the beginning does not change the meaning of the integer constant. The following notations are all equivalent to decimal 256.

	Notation
Hexadecimal	00100H
Decimal	0256
Octal	000400O　　00400Q
Binary	00100000000B

Finally, here are some examples of the same decimal 256 written with underscores.

	Notation
Hexadecimal	1_00H　　1_0_0H
Binary	1_0000_0000_B

3.5.2 Character Constants

■ Syntax ■

'char'

■ Function ■

A character constant is the 1-byte code for the specified character. char is either a single character, an escape sequence that evaluates to one, or an empty string. The last evaluates to the byte 0H.

The escape sequence consists of a backslash followed by a character or a number. SASM63K supports the following escape sequences.

Syntax	Function
\nnn	nnn is an octal number with up to three digits. The result is the character with this octal value. The octal number must be between 0 and 255.
ch	*ch* is a character. The result is the code for the same character.

■ Examples ■

The following are examples of character constants. The column on the right gives the character value in hexadecimal notation.

Character constant	Value
' '	00H
'A'	41H
'\0'	00H
'\47'	27H
'\377'	0FFH
'\8'	38H
'\047'	27H
'\F'	46H
'\''	27H

3.5.3 String Constants

■ **Syntax** ■

```
"characters"
```

■ **Function** ■

A string constant is a string of characters enclosed in double quotation marks ("). characters represents this string. This string can mix escape sequences and single-byte characters. The maximum string length is 255 characters.

■ **Examples** ■

The following are examples of string constants used as operands for the DB directive. The comment fields contain the hexadecimal codes for the individual bytes.

```
DB  "STRING"        ;53H,54H,52H,49H,4EH,47H
DB  "\377\111\222"  ;0FFH,49H,92H
```

3.6 Operators

Operands for instructions and directives can use expressions made up of constants, symbols representing addresses, and numbers joined together with operators. This section describes the function of the operators offered by SASM63K. These operators include the following.

- Arithmetic operators
- Logical operators
- Bitwise logical operators
- Relational operators
- Dot operator
- Special operator

There are unary and binary operators. A unary operator takes an expression on its right. A binary operator takes an expression on both sides.

SASM63K expresses numbers internally as 16-bit unsigned integers. All operations are performed as 16-bit unsigned operations. Note that overflows during operations are ignored.

The following descriptions use *expression*, *expression1*, and *expression2* to indicate expressions.

3.6.1 Arithmetic Operators

These operators are for standard arithmetic operations.

Operator	Syntax	Function
+	*expression1* + *expression2*	Addition
	+ *expression*	Unary plus
-	*expression1* – *expression2*	Subtraction
	– *expression*	Negation (unary operator)
*	*expression1* * *expression2*	Multiplication
/	*expression1* / *expression2*	Division
%	*expression1* % *expression2*	Modulo arithmetic (the remainder when *expression1* is divided by *expression2*)

■ Examples ■

The following are some arithmetic expressions and their hexadecimal values.

Arithmetic expression	**Value**
1234H+80H	12B4H
1234H–80H	11B4H
1234H*80H	1A00H
1234H/80H	24H
1234H%80H	34H
+1234H	1234H
–1234H	0EDCCH

3.6.2 Logical Operators

The logical operators base their results on the true/false values of the expressions to the right and left (or simply to the right in the case of a unary operator). The result is always either 0 (false) or 1 (true).

Operator	Syntax	Function
&&	*expression1* && *expression2*	Returns 1 if both the left and right terms are nonzero; otherwise returns 0.
||	*expression1* || *expression2*	Returns 0 if both the left and right terms are zero; otherwise returns 1.
!	! *expression*	Returns 1 if the right term is 0; otherwise returns 0.

■ **Examples** ■

The following are some logical expressions and their values.

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 operands.

Operator	Syntax	Function
&	*expression1* & *expression2*	Logical AND
|	*expression1* | *expression2*	Logical OR
^	*expression1* ^ *expression2*	Exclusive OR
<<	*expression1* << *expression2*	Shift left term to the left by the number of bits specified by the right term. Zeros enter from the lowest bit.
>>	*expression1* >> *expression2*	Shift left term to the right by the number of bits specified by the right term. Zeros enter from the highest bit.
~	~ *expression*	Bit complement of right term.

■ **Examples** ■

The following are some bitwise logical expressions and their values.

<table>
<tr><th>Bitwise logical expression</th><th>Value</th></tr>
<tr><td>1234H&4321H</td><td>0220H</td></tr>
<tr><td>1234H | 4321H</td><td>5335H</td></tr>
<tr><td>1234H ^ 4321H</td><td>5115H</td></tr>
<tr><td>1234H<<1</td><td>2468H</td></tr>
<tr><td>1234H>>1</td><td>091AH</td></tr>
<tr><td>~1234H</td><td>0EDCBH</td></tr>
</table>

3.6.4 Relational Operators

Relational operators compare two expressions. They return 1 if the indicated relation is true and 0 otherwise.

Operator	Syntax	Function
>	*expression1* > *expression2*	Returns 1 if *expression1* is greater than *expression2*; otherwise returns 0.
>=	*expression1* >= *expression2*	Returns 1 if *expression1* is greater than or equal to *expression2*; otherwise returns 0.
<	*expression1* < *expression2*	Returns 1 if *expression1* is less than *expression2*; otherwise returns 0.
<=	*expression1* <= *expression2*	Returns 1 if *expression1* is less than or equal to *expression2*; otherwise returns 0.
==	*expression1* == *expression2*	Returns 1 if *expression1* is equal to *expression2*; otherwise returns 0.
!=	*expression1* != *expression2*	Returns 1 if *expression1* is not equal to *expression2*; otherwise returns 0.

The following are some relational expressions and their values.

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

3.6.5 Dot Operator

The dot operator produces a bit address from a data address and a bit offset.

Operator	Syntax	Function
.	*expression1* . *expression2*	The value is the result of the following expression. ((*expression1* << 2) + *expression2*)

expression1 gives the data address; expression2, the bit offset within that data byte.

SASM63K treats the dot operator as an arithmetic operator, not checking the range of either *expression1* or *expression2*.

■ Examples ■

The following code fragment gives examples of dot operator usage.

```
       DSEG
       ORG    10H
DSYM1:
       DS     1
       CSEG
BSYM1  BIT    DSYM1.0
BSYM2  BIT    DSYM1.1

       BSET   \BSYM1
       BSET   \BSYM2
       BSET   \DSYM1.1
```

3.6.6 Special Operator

This operator, when applied to a symbol, yields the external memory bank in which that symbol is defined. This XBANK operator can therefore only be applied to symbols with the XDATA segment attribute.

Operator	Syntax	Function
XBANK	XBANK*xdata_symbol*	Yields the external bank number for the symbol *xdata_symbol*, a symbol with the XDATA segment attribute.

There must be at least one space or tab between the XBANK operator and xdata_symbol.

■ Examples ■

The following code fragment gives examples of XBANK operator usage.

```
       XSEG
       ORG    2:300H
XSYM0:
       CSEG
       MOV    A,#XBANK XSYM0
```

3.6.7 Operator 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—except for those of precedence level 3, which are evaluated from the right.

Table 3-3 Operator Precedence

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

3.7 Comments

Comments have no effect on programs, so the programmer can freely annotate the program as desired. The format is to start with a semicolon (;) and to follow that with the comment itself.

■ Examples ■

```
       MOV    A,#2   ;A <- 2
       JMP    LOOP   ;GOTO LOOP
;-------------------------------
;SUB-PROGRAM
;-------------------------------
       ORG    100H
```

The above shows how statements can be coded with comments after instructions and operands or with comments alone. The characters used are not limited to those described in Section 3.1 "Characters Allowed in Programs."

3.8 Addressing Modes

This section describes the OLMS-63K series address mode syntax, the contents of such expressions, and limits on their usage.

This manual does not discuss the details of which instructions can be used with which addressing modes. For such details, refer to the hardware manual for the particular microcontroller. The discussion that follows makes use of the following notations.

Symbol	**Meaning**
immediate	A nybble-sized immediate value. An expression that evaluates to a value between −0FH and +0FH.
bit_offset	A bit position. An expression that evaluates to a value between 0 and 3.
data_address	An expression that yields an address in data memory.
data_bit_address	An expression that yields a bit address in data memory.
code_address	An expression that yields an address in program memory.

In all addressing modes, the expressions used as instruction operands can make forward references to symbols.

The addressing modes for the OLMS-63K Series fall into the following categories.

- ❑ Immediate addressing
  - ◆ 4-bit immediate addressing

- ❑ Register addressing
  - ◆ Register direct addressing

- ❑ Data memory addressing
  - ■ Direct addressing
    - ◆ 4 kilonybble direct addressing
    - ◆ SFR bank direct addressing
    - ◆ Current bank direct addressing
  - ■ Register indirect addressing
    - ◆ HL register indirect addressing
    - ◆ XY register indirect addressing
    - ◆ Extra bank HL register indirect addressing
    - ◆ Extra bank XY register indirect addressing
  - ■ Post-increment register indirect addressing
    - ◆ Post-increment HL register indirect addressing
    - ◆ Post-increment XY register indirect addressing
    - ◆ Post-increment extra bank HL register indirect addressing
    - ◆ Post-increment extra bank XY register indirect addressing

- ❑ Program memory addressing
  - ■ Direct addressing
    - ◆ 64 kiloword direct addressing
    - ◆ 4 kiloword page addressing
  - ■ Relative addressing
    - ◆ PC-relative addressing
  - ■ PC-based addressing
    - ◆ PC-based addressing
  - ■ Register indirect addressing
    - ◆ RA register indirect addressing

- ❑ External memory addressing
  - ■ Direct addressing
    - ◆ 64 kilobyte direct addressing
  - ■ Register indirect addressing
    - ◆ RA register indirect addressing

The following sections discuss the syntax, meaning, and usage of the various addressing modes. The examples use underlining to indicate the addressing mode under discussion.

3.8.1 Immediate Addressing

3.8.1.1 4-Bit Immediate Addressing

■ **Syntax** ■

```
#immediate
```

■ **Function** ■

The value accessed is that represented by the operand. *immediate* is an expression representing a value.

SASM63K allows both signed and unsigned forms of immediate addressing. *immediate* has a value between -0FH and +0FH.

■ **Examples** ■

```
MOV    [HL],#0AH
MOV    A,#-2
```

3.8.2 Register Addressing

3.8.2.1 Register Direct Addressing

■ **Syntax** ■

Word-sized:	`RA`	RA register
Byte-sized:	`HL`	HL register
	`XY`	XY register
Nybble-sized:	`A`	Accumulator
	`H`	H register
	`L`	L register
	`X`	X register
	`Y`	Y register
	`RA0`	RA0 register
	`RA1`	RA1 register
	`RA2`	RA2 register
	`RA3`	RA3 register
	`CBR`	Current bank register
	`EBR`	Extra bank register
	`FLAG`	Flag register
Bit-sized:	`G`	G flag
	`C`	Carry flag
	`Z`	Zero flag

■ **Function** ■

The value accessed is that in the register.

■ **Examples** ■

```
MOV     A,[HL]
MOV     H,#2
MOV     L,#5
MOV     X,#0FH
MOV     Y,#0AH
MOV     RA0,#0H
MOV     RA1,#01H
MOV     RA2,#2H
MOV     RA3,#5H
MOV     CBR,#4
MOV     EBR,#5
FCLR    FLAG
INCB    HL
INCB    XY
INCW    RA
FSET    G
FSET    C
FSET    Z
```

3.8.3 Data Memory Addressing

The data memory addressing modes specify the address of a program variable in data memory.

3.8.3.1 4 Kilonybble Direct Addressing

■ **Syntax** ■

data_address

■ **Function** ■

This mode directly specifies an address in the 4-kilonybble data memory. The value accessed is the contents of that address. *data_address* is an expression representing an address in data memory.

■ **Examples** ■

```
MOV     0FFH,A
MOV     A,200H
```

3.8.3.2 SFR Bank Direct Addressing

■ Syntax ■

```
data_address
```

■ Function ■

The value accessed is the contents of the specified address in the SFR bank. data_address is an expression representing an address in the SFR bank. *data_address* has a value between 0 and 0FFH.

■ Examples ■

```
ADD    3FH,A
SUB    20H,A
INC    0AFH
```

3.8.3.3 Current Bank Direct Addressing

■ Syntax ■

Nybble-sized: `\data_address`

Bit-sized: `\data_bit_address`

■ Function ■

The value accessed is the contents of the nybble or bit at the specified address in the current bank. The operand represents that address. *data_address* is an expression representing a nybble address in the current bank; *data_bit_address*, one representing a bit address in the current bank.

SASM63K determines whether the address refers to a nybble or a bit from the instruction mnemonic.

■ Notes ■

data_address is an address within the data address space; *data_bit_address*, one within the bit address space.

The programmer must keep track of whether the address specified by the operand is actually within the currently selected bank. SASM63K provides the USING BANK directive to check the bank number for current bank direct addressing.

This form of addressing can also be viewed as using an 8-bit offset (0-0FFH) within the current bank. Although SASM63K generates the correct code, it considers the address as one in bank 0.

■ **Examples** ■

```
INC     \3FFH
MOV     \220H,#2

BCLR    \200H.2
BSET    \20H.5
BTST    \4FFH
```

3.8.3.4 HL Register Indirect Addressing

■ **Syntax** ■

Byte-sized: `C:[HL]`
`[HL]`

Nybble-sized: `C:[HL]`
`[HL]`

Bit-sized: `C:[HL].`*`bit_offset`*
`[HL].`*`bit_offset`*

■ **Function** ■

This mode uses the contents of the current bank register (CBR) and the HL register to specify an address in data memory. The value accessed is the byte, nybble, or bit at the specified address.

■ **Examples** ■

```
MOVHB   C:[HL],3FFFH
MOVHB   [HL],3FFFH

MOV     C:[HL],A
MOV     [HL],A

BSET    C:[HL].2
BSET    [HL].2
```

3.8.3.5 XY Register Indirect Addressing

■ **Syntax** ■

Byte-sized:	`C:[XY]` `[XY]`
Nybble-sized:	`C:[XY]` `[XY]`
Bit-sized:	`C:[XY].`*`bit_offset`* `[XY].`*`bit_offset`*

■ **Function** ■

This mode uses the contents of the current bank register (CBR) and the XY register to specify an address in data memory. The value accessed is the byte, nybble, or bit at the specified address.

■ **Examples** ■

```
MOVHB  C:[XY],[RA]
MOVHB  [XY],[RA]

ADD    C:[XY],A
ADD    [XY],A

BCLR   C:[XY].2
BCLR   [XY].2
```

3.8.3.6 Extra Bank HL Register Indirect Addressing

■ **Syntax** ■

Byte-sized:	`E:[HL]`
Nybble-sized:	`E:[HL]`
Bit-sized:	`E:[HL].`*`bit_offset`*

■ **Function** ■

This mode uses the contents of the extra bank register (EBR) and the HL register to specify an address in data memory. The value accessed is the byte, nybble, or bit at the specified address.

■ **Examples** ■

```
MOVHB  E:[HL],[RA]

ROL    E:[HL]

BTST   E:[HL].2
```

3.8.3.7 Extra Bank XY Register Indirect Addressing

■ **Syntax** ■

Byte-sized: `E:[XY]`

Nybble-sized: `E:[XY]`

Bit-sized: `E:[XY].`*`bit_offset`*

■ **Function** ■

This mode uses the contents of the extra bank register (EBR) and the XY register to specify an address in data memory. The value accessed is the byte, nybble, or bit at the specified address.

■ **Examples** ■

```
MOVHB  E:[XY],[RA]

ROR    E:[XY]

BSET   E:[XY].0
```

3.8.3.8 Post-Increment HL Register Indirect Addressing

■ Syntax ■

Byte-sized: `C:[HL+]`
`[HL+]`

Nybble-sized: `C:[HL+]`
`[HL+]`

Bit-sized: `C:[HL+].`*`bit_offset`*
`[HL+].`*`bit_offset`*

■ Function ■

This mode uses the contents of the current bank register (CBR) and the HL register to specify an address in data memory. The value accessed is the byte, nybble, or bit at the specified address.

After the access, the HL register is incremented by 2 for byte-sized access or by 1 for nybble- or bit-sized access.

■ Examples ■

```
MOVHB  C:[HL+],[RA]
MOVHB  [HL+],[RA]

ROL    C:[HL+]
ROL    [HL+]

BTST   C:[HL+].2
BTST   [HL+].2
```

3.8.3.9 Post-Increment XY Register Indirect Addressing

■ Syntax ■

Byte-sized: `C:[XY+]`
`[XY+]`

Nybble-sized: `C:[XY+]`
`[XY+]`

Bit-sized: `C:[XY+].`*`bit_offset`*
`[XY+].`*`bit_offset`*

■ Function ■

This mode uses the contents of the current bank register (CBR) and the XY register to specify an address in data memory. The value accessed is the byte, nybble, or bit at the specified address.

After the access, the XY register is incremented by 2 for byte-sized access or by 1 for nybble- or bit-sized access.

■ Examples ■

```
MOVHB  C:[XY+],[RA]
MOVHB  [XY+],[RA]

ROL    C:[XY+]
ROL    [XY+]

BTST   C:[XY+].1
BTST   [XY+].1
```

3.8.3.10 Post-Increment Extra Bank HL Register Indirect Addressing

■ Syntax ■

Byte-sized: `E:[HL+]`

Nybble-sized: `E:[HL+]`

Bit-sized: `E:[HL+].`*`bit_offset`*

■ Function ■

This mode uses the contents of the extra bank register (EBR) and the HL register to specify an address in data memory. The value accessed is the byte, nybble, or bit at the specified address.

After the access, the HL register is incremented by 2 for byte-sized access or by 1 for nybble- or bit-sized access.

■ Examples ■

```
MOVHB  E:[HL+],[RA]

ROR    E:[HL+]

BSET   E:[HL+].3
```

3.8.3.11 Post-Increment Extra Bank XY Register Indirect Addressing

■ Syntax ■

Byte-sized: `E:[XY+]`

Nybble-sized: `E:[XY+]`

Bit-sized: `E:[XY+].`*`bit_offset`*

■ Function ■

This mode uses the contents of the extra bank register (EBR) and the XY register to specify an address in data memory. The value accessed is the byte, nybble, or bit at the specified address.

After the access, the XY register is incremented by 2 for byte-sized access or by 1 for nybble- or bit-sized access.

■ Examples ■

```
MOVHB  E:[XY+],[RA]

ROR    E:[XY+]

BSET   E:[XY+].3
```

3.8.4 Program Memory Addressing

Program memory addressing gives the jump target, call target, or address of data in program memory.

3.8.4.1 64 Kiloword Direct Addressing

3.8.4.1.1 Direct Table Addressing

■ **Syntax** ■

```
    code_address
```

■ **Function** ■

The value accessed is the contents of the byte at the specified address in program memory. *code_address* is an expression representing an address in program memory.

■ **Notes** ■

code_address must be within the addresses available for program memory.

■ **Examples** ■

```
        CSEG
CODE_TABLE:
        DW      1020H,3040H,5060H,7080H
        .
        .
        .
        MOVHB   [HL],CODE_TABLE
        MOVLB   [XY],CODE_TABLE
```

3.8.4.1.2 Direct Code Addressing

■ Syntax ■

```
        code_address
```

■ Function ■

This addressing mode directly specifies the jump target or call target for the LJMP and LCAL instructions. *code_address* is an expression representing an address in program memory.

■ Notes ■

code_address must be within the addresses available for program memory.

■ Examples ■

```
        LJMP   LABEL
        .
        .
        .
LABEL:
        .
```

3.8.4.2 4 Kiloword Page Addressing

■ **Syntax** ■

code_address

■ **Function** ■

This addressing mode specifies the jump target or call target for the JMP and CAL instructions. *code_address* is an expression representing an address in program memory.

■ **Notes** ■

code_address must be within the addresses available for program memory and also within the current 4-kiloword page.

■ **Examples** ■

```
       ORG    2000H
LABEL1:
       :
       JMP    LABEL1
       JMP    LABEL2          ; Error
       :
       ORG    3000H
       :
LABEL2:
```

3.8.4.3 PC-Relative Addressing

■ **Syntax** ■

```
code_address
```

■ **Function** ■

This addressing mode specifies the jump target for the SJMP and conditional branch instructions. *code_address* is an expression representing an address in program memory.

■ **Notes** ■

The difference between the address of the next instruction and *code_address* must be within the range between -128 and +127.

■ **Examples** ■

```
SJMP   LOOP
BC     NEXT
```

3.8.4.4 PC-Based Addressing

■ **Syntax** ■

```
PC+A
```

■ **Function** ■

This addressing mode specifies the jump target for the JMP instruction using the contents of the program counter and the accumulator.

■ **Examples** ■

```
JMP    PC+A
```

3.8.4.5 RA Register Indirect Addressing

■ Syntax ■

```
[RA]
```

■ Function ■

This addressing mode uses the contents of the RA register to specify an address in program memory. The value accessed is the contents of that address.

■ Examples ■

```
MOVHB  C:[HL],[RA]

MOVLB  E:[XY],[RA]
```

3.8.5 External Memory Addressing

The external memory addressing modes are for accessing data in external memory.

3.8.5.1 64 Kilobyte Direct Addressing

■ Syntax ■

xdata_address

■ Function ■

This addressing mode directly specifies an address in the external memory space. The value accessed is the byte at that address. *xdata_address* is an expression representing an address in external memory.

■ Notes ■

xdata_address must be within the addresses available for external memory.

■ **Examples** ■

```
       XSEG
XMEM_TABLE:
       DB     10H,20H,30H,40H,50H,60H,70H
       .
       CSEG
       .
       .
       MOVXB  [HL],XMEM_TABLE
```

3.8.5.2 RA Register Indirect Addressing

■ **Syntax** ■

```
       [RA]
```

■ **Function** ■

This addressing mode uses the contents of the RA register to specify an address in external memory. The value accessed is the byte at that address.

■ **Notes** ■

Although the syntax is the same as that for RA register indirect addressing of program memory, the value accessed differs. SASM63K determines whether the address refers to program memory or external memory from the instruction mnemonic.

■ **Examples** ■

```
       MOVXB  C:[HL],[RA]

       MOVXB  [RA],[XY]
```

Chapter 4

Directives

This chapter describes directives. Directives control the assembler, so, except for the DB and DW directives, 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    expression
symbol =      expression
```

■ Function ■

The EQU directive assigns the value of a constant expression to the specified *symbol*. The *expression* cannot contain forward references.

A symbol defined with EQU may not be redefined with the same program as a label or a new symbol. It is not given a segment type.

The *expression* must evaluate with the range between 0 and 0FFFFH.

■ Examples ■

```
         DSEG
         ORG      20H
BUF1:    DS       10

BUFSIZ   EQU      10
```

4.1.2 SET

■ Syntax ■

```
symbol SET    expression
```

■ Function ■

The SET directive has the same function as the EQU directive except that it permits redefinition of the symbol any number of times in the program with additional SET directives. It assigns the value of the expression to the specified *symbol*. The *expression* cannot contain forward references.

The *symbol* is assigned the value of the *expression* from the current SET directive. Unlike the EQU directive, which only assigns a single value to a symbol, the SET directive assigns a value that remains valid until changed by the next SET directive.

■ Examples ■

```
         .
         .
SYMBOL   SET      3
         MOV      A,#SYMBOL
         .
         .
         .
SYMBOL   SET      9
         MOV      A,#SYMBOL
```

The above code uses the SET directive to define the symbol SYMBOL. In the MOV instruction following the first SET directive, SYMBOL has the value 3. In the MOV instruction following the second SET directive, however, it has the value 9.

4.1.3 CODE

■ **Syntax** ■

```
symbol CODE   expression
```

■ **Function** ■

The CODE directive assigns a code address to the specified *symbol*. The *expression* must evaluate to a code address and cannot contain forward references.

The *symbol* is assigned the CSEG segment type.

Symbols defined with the CODE directive cannot be redefined.

■ **Examples** ■

```
CODESYM1 CODE     1000H
CODESYM2 CODE     2000H

         MOVHB    [HL],CODESYM1
         MOVLB    [XY],CODESYM2
```

4.1.4 DATA

■ **Syntax** ■

```
symbol DATA   expression
```

■ **Function** ■

The DATA directive assigns a RAM data address to the specified *symbol*. The *expression* must evaluate to a RAM data address and cannot contain forward references.

The *symbol* is assigned the DATA segment type.

Symbols defined with the DATA directive cannot be redefined.

■ **Examples** ■

```
DATASYM1 DATA     200H
DATASYM2 DATA     300H

         MOV      DATASYM1,A
         MOV      A,DATASYM2
```

4.1.5 BIT

■ Syntax ■

```
symbol BIT    expression
```

■ Function ■

The BIT directive assigns a bit address to the specified *symbol*. The *expression* must evaluate to a bit address and cannot contain forward references.

The *symbol* is assigned the BIT segment type.

Symbols defined with the BIT directive cannot be redefined.

■ Examples ■

```
BITSYM1  BIT      200H.1

         DSEG
         ORG      100H
LABEL:   DS       2
BITSYM2  BIT      LABEL.2
```

4.1.6 XDATA

■ Syntax ■

```
symbol XDATA  xbank_no:expression
symbol XDATA  expression
```

■ Function ■

The XDATA directive assigns an XDATA address to the specified *symbol*. The *xbank_no* expression must evaluate to an external memory bank number and cannot contain forward references. The address *expression* must evaluate to an external memory address and cannot contain forward references.

If only the address *expression* is present, the assembler uses external memory bank 0.

The symbol is assigned the XDATA segment type.

Symbols defined with the XDATA directive cannot be redefined.

■ Examples ■

```
XDATASYM1 XDATA   1:200H
XDATASYM2 XDATA   300H
          .
          .
          .
          MOVXB   XDATASYM1,[XY]
          MOVXB   [HL],XDATASYM2
```

4.2 Memory Segment Control

Memory segment control directives define the start of segments (address spaces). There are four segments: CODE, DATA, BIT, and XDATA.

Each segment has its own location counter. The location counter values have a one-to-one correspondence 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 a CODE segment. The assembler starts out in the CODE segment until it sees a CSEG, DSEG, BSEG, or XSEG directive. When CSEG is first used, the location counter becomes 0. The assembler updates the location counter with each ORG, DS, or DW directive and with each microcontroller instruction.

Upon the second or subsequent appearance of the CSEG directive, the location counter assumes the value that it had at the end of the preceding CODE segment.

■ Examples ■

```
CSEG
ORG      200H
MOV      A,#2
.
.
.
DSEG
```

4.2.2 DSEG

■ Syntax ■

```
        DSEG
```

■ Function ■

The DSEG directive defines the start of a DATA segment. This segment is for defining symbols in the data address space and for reserving space for variables. This directive can be used any number of times in a program. The assembler updates the location counter with each ORG or DS directive. When DSEG is first used, the location counter becomes 0.

Upon the second or subsequent appearance of the DSEG directive, the location counter assumes the value that it had at the end of the preceding DATA segment.

■ Examples ■

```
         DSEG
         ORG      100H
DATA_SYM:DS       5

         CSEG
         .
         .
         .
         MOV      DATA_SYM,A
```

4.2.3 BSEG

■ Syntax ■

```
      BSEG
```

■ Function ■

The BSEG directive defines the start of a BIT segment. This segment is for defining symbols in the bit address space and for reserving space for variables. This directive can be used any number of times in a program. The assembler updates the location counter with each ORG or DBIT directive. When BSEG is first used, the location counter becomes 0.

Upon the second or subsequent appearance of the BSEG directive, the location counter assumes the value that it had at the end of the preceding BIT segment.

■ **Examples** ■

```
         DSEG
         ORG      100H
DATA_SYM:DS       1

         BSEG
         ORG      DATA_SYM.0
BIT_SYM: DBIT     4
```

4.2.4 XSEG

■ **Syntax** ■

```
XSEG
```

■ **Function** ■

The XSEG directive defines the start of an XDATA segment. This directive can be used any number of times in a program. The assembler updates the location counter with each ORG, DB, DW, or DS directive. When XSEG is first used, the location counter becomes 0.

Upon the second or subsequent appearance of the XSEG directive, the location counter assumes the value that it had at the end of the preceding XDATA segment.

■ **Examples** ■

```
            XSEG
            ORG    100H
XDATA_TBL0: DB     0H,10H,20H,30H,40H,50H,60H,70H
            .
            .
            .
            CSEG
```

4.3 Location Counter Control

The location counter control directives modify the location counter for the current address space. They include the ORG, DS, and DBIT directives.

4.3.1 ORG

■ Syntax ■

```
ORG    address
ORG    xbank_no:address          (valid only in an XDATA segment)
```

■ Function ■

The ORG directive sets the location counter for the current segment to the specified *address*. This directive can be used in any segment.

The *address* expression cannot include forward references. It must evaluate to a valid address within the current address space.

The *xbank_no* expression must evaluate to an external memory bank number in the range between 0 and 255. If only the address expression is present for an XDATA segment, the assembler uses external memory bank 0.

The ORG directive may be used any number of times in a single program, but caution is advised since the assembler does not check for overlapping segments.

■ Examples ■

```
CSEG
ORG     100H
.
.
.
ORG     1:20H             ; Error
.
.
.
XSEG
ORG     200H              ; External memory bank 0
.
.
.
ORG     1:100H            ; External memory bank 1
```

4.3.2 DS

■ Syntax ■

```
[label:] DS   size
```

■ Function ■

The DS directive reserves a memory area with undefined contents. The size of this area, given by the expression *size*, is in words for the CODE segment, in bytes for the XDATA segment, and in nybbles for the DATA segment. The DS directive simply adds the value of the size expression to the location counter for the current segment. The address restrictions for the segment cannot, however, be exceeded.

The *size* expression gives the memory space size in words, bytes, or nybbles. It cannot contain forward references.

The DS directive is used for reserving space in the CODE, DATA, and XDATA segments.

■ Examples ■

```
         DSEG
         ORG      100H
BUFFER:  DS       10H                 ; Reserve 10H nybbles


         CSEG
         ORG      200H
FUNC1:
         MOV      CBR,#1
         MOV      A,BUFFER
```

The above example reserves a 10H-nybble space in the DATA segment.

4.3.3 DBIT

■ Syntax ■

```
[label:] DBIT size
```

■ Function ■

The DBIT directive reserves a memory area of the specified size in the BIT segment. Its adds the value of the *size* expression to the location counter for the BIT segment.

The *size* expression gives the memory space size in bits. It cannot contain forward references.

The DBIT directive is used for reserving space in the BIT segment.

■ Examples ■

```
         BSEG
         ORG      100H.0
FLAG1:   DBIT     4                  ; Reserve 4 bits

         CSEG
         ORG      300H
FUNC1:   MOV      CBR,#1
         BSET     \FLAG1
```

The above example reserves a 4-bit space in the BIT segment.

4.4 Data Definitions

Data definition directives initialize program or external memory to specified values. They are used to define data in program or external memory.

4.4.1 DB

■ **Syntax** ■

```
[label:] DB   expression [, expression]...
[label:] DB   string_constant
```

Function

The DB directive initializes external memory in 1-byte units. It can therefore only be used in the XDATA segment.

As operands, the directive takes either expressions or a string constant.

An *expression* may contain forward references. It must evaluate to single-byte data—that is, a value within the following ranges.

−0FFH to −1H (0FF01H to 0FFFFH)
0H to 0FFH

The values from the expressions are assigned to bytes starting at the location counter in the order that the expressions appear in the list.

■ **Examples** ■

```
         XSEG
         ORG      10H      ;
TABLE1:  DB       1,2,3,4,5,6,7,8,9

TABLE2:  DB       'A','B','C','D','E','F'

TABLE3:  DB       "string"
```

4.4.2 DW

■ Syntax ■

```
[label:] DW expression [, expression]...
```

■ Function ■

The DW directive initializes program or external memory in word units. It can therefore only be used in the CODE and XDATA segments.

As operands, the directive takes expressions.

An *expression* may contain forward references. It must evaluate to word (2-byte) data—that is, a value within the following ranges.

–0FFFFH to –1H (0001H to 0FFFFH)
0H to 0FFFFH

In the CODE segment, the values from the expressions are assigned to words starting at the location counter in the order that the expressions appear in the list. In the XDATA segment, the values from the expressions are assigned to bytes with the lower half of the word preceding the upper half starting at the location counter in the order that the expressions appear in the list.

■ Examples ■

```
         CSEG
         ORG      10H       ;
TABLE1:  DW       1,2,3     ; Defines words containing 0001H, 0002H, and 0003H.

         XSEG
         ORG      20H
TABLE2:  DW       4,5,6     ; Defines the byte sequence 04H 00H 05H 00H 06H 00H.
         DW       1234H     ; Defines the byte sequence 34H 12H.
```

4.5 Listing Control

Listing control directives affect the generation of the listing, object, and error files and the listing file format. They have absolutely no effect on the code generated as the result of assembly.

4.5.1 DATE

■ **Syntax** ■

```
DATE   "character_string"
```

■ **Function** ■

The DATE directive assigns the date to be inserted in the listing file header. In the absence of such a specification, the assembler uses the date of assembly obtained from the operating system. The *character_string* may be up to 25 characters long. Any characters beyond this limit are ignored.

If the file contains more than one DATE directive, only the last one is effective.

■ **Examples** ■

```
DATE     "Apr 1, 1995"
```

4.5.2 TITLE

■ **Syntax** ■

```
TITLE  "character_string"
```

■ **Function** ■

The TITLE directive assigns the title to be inserted in the listing file header. In the absence of such a specification, the assembler leaves the title blank. The *character_string* may be up to 70 characters long. Any characters beyond this limit are ignored.

If the file contains more than one TITLE directive, only the last one is effective.

■ **Examples** ■

```
TITLE    "Sample Program"
```

4.5.3 PAGE

■ **Syntax** ■

```
PAGE   [page_length, page_width]
```

■ **Function** ■

The PAGE directive specifies a new page in the listing file. The page dimension parameters are provided only for compatibility with ASM63KN and are ignored by SASM63K.

4.5.4 OBJ/NOOBJ

■ **Syntax** ■

```
OBJ [(object_file)]
NOOBJ
```

■ **Function** ■

The OBJ directive specifies the generation of an *object_file* with the specified name. In the absence of a name specification, the assembler uses the default output specification. The assembler ignores any extension in the file name specification because it generates multiple object files. For the default file name and the default extensions for the resulting files, see Section 2.2 "File Specifications."

The NOOBJ directive suppresses object file output.

A file may contain multiple OBJ and NOOBJ directives, but only the first one has any effect.

The default directive is OBJ.

■ **Examples** ■

```
OBJ(SAMPLE)
```

4.5.5 PRN/NOPRN

■ **Syntax** ■

```
PRN [(print_file)]
NOPRN
```

■ **Function** ■

The PRN directive specifies the generation of a listing file with the specified name. For the default file name and the defaults for omitted portions of the file name, see Section 2.2 "File Specifications."

The NOPRN directive suppresses listing file output.

A file may contain multiple PRN and NOPRN directives, but only the first one has any effect.

The default directive is NOPRN.

■ **Examples** ■

```
PRN(SAMPLE.PRN)
```

4.5.6 ERR/NOERR

■ Syntax ■

```
ERR [(error_file)]
NOERR
```

■ Function ■

The ERR directive specifies the output of error messages to an *error_file* with the specified name. For the default file name and the defaults for omitted portions of the file name, see Section 2.2 "File Specifications."

The NOERR directive tells SASM63K to send error messages to the standard output (screen).

A file may contain multiple ERR and NOERR directives, but only the first one has any effect.

In the absence of an ERR directive, the error messages go to the screen. The default directive is NOERR.

■ Examples ■

```
ERR(SAMPLE.ERR)
```

4.5.7 SYM/NOSYM

■ **Syntax** ■

```
SYM
NOSYM
```

■ **Function** ■

The SYM directive adds a symbol list to the listing file. This list provides information on user-defined symbols used in the program.

The NOSYM directive suppresses the output of this list.

A file may contain multiple SYM and NOSYM directives, but only the first one has any effect.

If a NOPRN directive is in effect, the SYM directive is ignored. The default directive is NOSYM.

■ **Examples** ■

```
PRN(SAMPLE.LST)
SYM
```

4.5.8 REF/NOREF

■ **Syntax** ■

```
REF
NOREF
```

■ **Function** ■

The REF directive adds a cross reference list to the listing file. This list provides information on user-defined symbols and the line numbers where they are used in the program.

The NOREF directive suppresses the output of this list.

A file may contain multiple REF and NOREF directives, but only the first one has any effect.

If a NOPRN directive is in effect, the REF directive is ignored. The default directive is NOREF.

■ **Examples** ■

```
PRN(SAMPLE.LST)
REF
```

4.5.9 DEBUG/NODEBUG

■ Syntax ■

```
DEBUG
NODEBUG
```

■ Function ■

The DEBUG directive adds debugging and symbol information to an object file. For the object file receiving this information, see Section 6.1.2 "Debugging Information."

The NODEBUG directive suppresses the output of this information.

A file may contain multiple DEBUG and NODEBUG directives, but only the first one has any effect.

If a NOOBJ directive is in effect, the DEBUG directive is ignored. The default directive is NODEBUG.

■ Examples ■

```
OBJ(SAMPLE)
DEBUG
```

4.5.10 LIST/NOLIST

■ Syntax ■

```
LIST
NOLIST
```

■ Function ■

The LIST directive specifies the output of assembly listing information beginning from the next line in the source code. Output continues up until the next NOLIST directive.

The NOLIST directive suppresses listing output until the next LIST directive.

SASM63K starts assembling a file with listing output on. In the absence of any LIST and NOLIST directives, it therefore produces an assembly listing of the entire program.

If a NOPRN directive is in effect, LIST directives are ignored.

■ Examples ■

```
PRN
NOLIST              ; Assembly listing stops from next line.
.
.
.
LIST                ; Assembly listing resumes from next line.
.
.
.
```

4.6 Checking CBR Bank Number

4.6.1 USING BANK

■ Syntax ■

```
USING BANK status
```

■ Function ■

The USING BANK directive informs SASM63K of the bank number in the current bank register (CBR).

SASM63K then checks the bank number assumed with the USING BANK directive against the bank number actually specified for current bank direct addressing and issues a warning if they do not match.

status can be either of the following.

status	**Description**
bank_no	Expression giving a bank number
ANY	Disables current bank checking. (Default)

bank_no is an expression representing a data memory bank number.

If a bank number is specified, SASM63K checks it against the bank number used in current bank direct addressing. The ANY specification disables checking. Prior to the first USING BANK directive, the setting is ANY.

■ Warning ■

The USING BANK directive tells SASM63K to assume that the current bank register contains a specific value. The assembler does not generate object code for checking this assumption. It is up to the programmer to provide the microcontroller instructions for setting the hardware.

The scope of a USING BANK directive runs from the next source statement to the next USING BANK directive. Note that this scope has nothing to do with actual program flow.

■ **Examples** ■

```
         DSEG
         ORG      200H
DATA2:   DS       20H

         ORG      300H
DATA3:   DS       20H
         .
         .
         .
         CSEG
         USING    BANK    2H        ; Assume bank 2 is current bank.
         MOV      CBR,#(DATA2>>8)&0FH

         MOV      \DATA2,#3         ; No problem.
         MOV      \DATA3,#1         ; Warning generated.
         .
         .
         .
         USING    BANK    3H        ; Assume bank 3 is current bank.
         MOV      CBR,#(DATA3>>8)&0FH

         MOV      \DATA3,#2         ; No problem.
         MOV      \DATA2,#5         ; Warning generated.
         .
         .
         .
```

The first USING BANK directive has the operand 2H, so the assembler assumes 2H for the current bank. The next two MOV instructions use current bank direct addressing for their operands, but DATA3 is in data memory bank 3, so there is a mismatch. SASM63K therefore flags the second MOV instruction with a warning message.

The second USING BANK directive has the operand 3H, so the assembler assumes 3H for the current bank. The MOV instruction accessing DATA2 therefore produces a warning message.

4.7 Assembler Control

4.7.1 TYPE

■ Syntax ■

```
TYPE (dcl_name)
```

■ Function ■

The TYPE directive specifies the DCL file name for the target microcontroller. It causes the assembler to read in a file with the specified base name and the extension .DCL. The base name of the DCL file for a particular microcontroller is the name of the microcontroller with the MSM prefix shortened to just M. For the MSM63184 microcontroller, for example, use the base name M63184 with the TYPE directive.

The assembler reads the DCL file specified by the TYPE directive and sets itself up for the particular device. The TYPE directive must therefore appear at the start of a program.

dcl_name can contain an explicit path specification. If it does, however, the assembler does not use the PATH environment variable to search for DCL files.

dcl_name cannot contain an extension. DCL files are limited to the extension .DCL.

The TYPE directive must appear before any instructions and before any of the following directives.

EQU SET CODE DATA XDATA BIT DB DW DS DBIT ORG DEFINE MACRO

If the TYPE directive violates any of the above rules, the assembler aborts.

For details on DCL files, see section 1.3 "DCL Files."

■ Examples ■

```
;------------------
; TEST PROGRAM
;------------------
    TYPE   (M63XXX)
    CSEG
    .
    .
    .
```

4.7.2 END

■ Syntax ■

```
END
```

■ Function ■

The END directive indicates the end of the program. SASM63K assembles everything up to the END directive.

The END directive takes neither a label nor operands.

■ Examples ■

```
TYPE     (M63XXX)
.
.
.
END
MOV
```

In the above example, the END directive is followed by a statement with a syntax error. SASM63K does not generate an error message, however, since it simply ignores the statement.

4.8 Preprocessor Directives

4.8.1 INCLUDE

■ Syntax ■

```
INCLUDE (include_file)
```

■ Function ■

The assembler replaces the INCLUDE directive with the contents of the specified file.

The contents of include_file are processed just as if they were present in the current file. Files expanded with the INCLUDE directive may themselves contain INCLUDE directives.

■ Examples ■

```
;---------------
; SOURCE FILE
;---------------
     INCLUDE  (DEFINE.H)

     CSEG
     .
     .
     .
```

```
;---------------
; INCLUDE FILE  DEFINE.H
;---------------
     TYPE  (M63XXX)

     SYM
     REF
```

This example uses an include file to store listing file control directives.

4.8.2 DEFINE

■ **Syntax** ■

```
DEFINE symbol text
```

■ **Function** ■

The DEFINE directive assigns the specified text string to the specified *symbol*. Whenever the assembler encounters the *symbol* in the source program, it replaces the symbol with the *text* string.

There must be at least one space or tab between the *symbol* and the *text* string. Such spaces and tabs are not included in the text string. The *text* string may be a string of any characters. It is terminated by a carriage return or a semicolon.

The same symbol cannot appear in more than one DEFINE directive. The only exceptions to this rule are DEFINE directives containing *text* strings identical to the original definition.

■ **Examples** ■

```
DEFINE   FLAG     [30H].0
```

4.8.3 SUBR

■ Syntax ■

```
SUBR   symbol [LOCAL([local_label,...])]
.
.
.
ENDSUB
```

■ Function ■

The SUBR directive causes the statements between SUBR and ENDSUB to be assembled if the *symbol* has previously been referenced. If the symbol has not been referenced, the statements up to the ENDSUB are ignored.

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

The statement block between SUBR and ENDSUB can contain local labels. Those labels are declared as a comma-delimited list in parentheses after the keyword LOCAL. Since each local label name is replaced with a unique name guaranteed not to overlap other names, different SUBR statement blocks may use the same local labels.

The SUBR directive cannot appear between the SUBR and ENDSUB directives.

For a specific example of the use of the SUBR directive, see Chapter 10 "Sample Program."

■ **Examples** ■

```
    CAL SUB1

SUBR      SUB1      LOCAL(LAB1,LAB2)
LAB1:
    [HL]=1
LAB2:
    .
    .
ENDSUB

SUBR      SUB2      LOCAL(LAB1)
    [HL]=1
LAB1:
    .
    .
ENDSUB

    CAL SUB2        ; This statement results in an error because the label SUB2 is
; undefined.
```

In the above example, SUB1 is assembled because it has been previously referenced. SUB2 is not assembled because its reference follows the definition. Although LAB1 appears to be defined twice, there is no error message because both times it is declared LOCAL.

4.8.4 REFER

■ **Syntax** ■

```
REFER symbol
```

■ **Function** ■

The REFER directive causes the expansion of the SUBR block for the symbol regardless of whether the symbol has been referenced. In other words, the symbol appearing in a REFER directive is regarded as a referenced symbol.

■ **Examples** ■

```
REFER    SUB2

SUBR     SUB2
     [HL]=1
     .
     .
     .
ENDSUB
```

In the above example, the symbol SUB2 appears in a REFER directive, so the statement block between SUBR and ENDSUB is assembled.

4.8.5 Macro Definitions

■ Syntax ■

```
MACRO  symbol([parameter,...]) [LOCAL([local_label,...])]
.
.
.
ENDM
```

■ Function ■

A macro assigns a series of statements to a single symbol so that the programmer can then substitute that symbol for that series of statements. The assembler expands the macro each time that it finds the symbol defined for that macro. A macro can also have parameters so that different expansions of the macro can result in different text strings. When label definitions are needed for statements within a macro, these labels must be declared local so that they assume unique names for each macro expansion.

The macro definition starts with the MACRO directive, the *symbol* for the macro, and, in parentheses, a comma-delimited list of parameters for the macro. The parentheses are obligatory even when there are no parameters. The parameters inside them are symbols. If there are label definitions within the macro, their names appear in a comma-delimited list inside parentheses after the keyword LOCAL.

Parameter names and *local_label* names are valid only within the macro definition. They cannot be referenced elsewhere. As a result, however, parameters and local labels with identical names can be used within other macro definitions.

■ Examples ■

(1) Example of the simplest type of macro definition

```
MACRO FCLR_FLAG()
  FCLR   C
  FCLR   G
  FCLR   Z
ENDM
```

(2) Example of macro with a parameter

```
MACRO ROLB(adr)
  ROL    \adr
  ROL    \adr+1
ENDM
```

(3) Example of macro with local label

```
MACRO DECB_HL() LOCAL(label1)
  DEC    L
  BNC    label1
  DEC    H
label1:          ; This label is replaced with a unique label for each macro expression.
ENDM
```

4.8.6 Macro Calls

■ Syntax ■

```
macro_name([argument,...])
```

■ Function ■

macro_name must be the name of a previously defined macro. If the macro takes arguments, they appear as a comma-delimited list in parentheses following this name. The parentheses are required even when there are no arguments.

Arguments are arbitrary strings terminated by a comma or a right parenthesis. They replace the corresponding parameters in the macro definition.

■ Examples ■

These examples call the macros defined in the preceding section.

(1) FCLR_FLAG()

This call expands to the following instructions.

```
FCLR     C
FCLR     G
FCLR     Z
```

(2) ROLB(20H)

The macro definition included the parameter adr, so anywhere that the symbol adr appears in the macro, it is replaced by the string "20H." The call therefore expands to the following instructions.

```
        ROL      \20H
        ROL      \20H+1
```

Note: When a right parenthesis, comma or backslash is needed within an argument string, precede it with a backslash .

```
        ROLB((20H+1\))
```

(3) DECB_HL()

The assembler expands this call, giving a new name to the local label label1.

```
        DEC      L
        BNC      ?00001
        DEC      H
?00001:
```

4.9 Optimized Branch Directives

OLMS-63K provides several jump instructions and subroutine call instructions. If GJMP or GCAL directives are used instead of directly coding the microcontroller instructions, then SASM63K will convert them to the optimal instructions corresponding to the address value of the branch destination or distance to the branch destination.

Short branches are ones between -128 and +127 bytes relative to the program counter. Long branches are ones within the same 4-kiloword page. Far branches can be anywhere in the code memory space.

4.9.1 Optimization of Jump Instructions

■ **Syntax** ■

```
GJMP   symbol
```

■ **Function** ■

The GJMP directive produces an unconditional jump.

The operand *symbol* gives the branch destination. SASM63K converts this directive to the optimal jump variant - short, long, or far - for the branch destination *symbol*. For the rules used in making this selection, see Section 4.9.4 "Conversion Rules." For the variants selected, see Section 4.9.5 "Directive Expansions."

symbol can be either a code segment label or the location counter symbol ($). Expressions are not allowed.

■ **Examples** ■

```
        CSEG
        ORG     200h
LABEL1:
        GJMP    LABEL1         ;Converts to SJMP instruction.
        .
        .
        .
        ORG     300H
        GJMP    LABEL1         ;Converts to JMP instruction.
        .
        .
        .
        ORG     1000H
        GJMP    LABEL1         ;Converts to LJMP instruction.
```

In the above example, the first GJMP directive is within the range between -128 and+127 bytes of LABEL1, so converts to an SJMP instruction. The second GJMP directive falls outside this range, but is still within the same 4-kiloword page as LABEL1, so converts to a JMP instruction. The third GJMP directive falls outside both ranges, so converts to an LJMP instruction.

4.9.2 Optimization of Conditional Jump Instructions

■ **Syntax** ■

```
GBC    symbol
GBLT   symbol
GBNC   symbol
GBGE   symbol
GBLE   symbol
GBGT   symbol
GBG    symbol
GBNG   symbol
GBZ    symbol
GBEQ   symbol
GBNZ   symbol
GBNE   symbol
```

■ **Function** ■

These directives produce conditional jumps.

The jump conditions are the same as those for the instructions produced by dropping the initial G from the directive names. For further details on jump conditions, see the Instruction Manual.

The operand *symbol* gives the branch destination. SASM63K converts this directive to the optimal conditional jump variant - short, long, or far - for the branch destination *symbol*. For the rules used in making this selection, see Section 4.9.4 "Conversion Rules." For the variants selected, see Section 4.9.5 "Directive Expansions."

symbol can be either a code segment label or the location counter symbol ($). Expressions are not allowed.

4.9.3 Optimization of Call Instructions

■ Syntax ■

```
GCAL   symbol
```

■ Function ■

The GCAL directive produces a subroutine call.

The operand *symbol* gives the branch destination. SASM63K converts this directive to the optimal call variant - long or far - for the branch destination *symbol*. For the rules used in making this selection, see Section 4.9.4 "Conversion Rules." For the variants selected, see Section 4.9.5 "Directive Expansions."

symbol must be a code segment label. Expressions are not allowed.

■ Examples ■

```
         CSEG
         ORG      200H
SUB1:
         .
         .
         .
         ORG      250H
         GCAL     SUB1                ;Converts to a CAL instruction.
         .
         .
         .
         ORG      1000H
         GCAL     SUB1                ;Converts to an LCAL instruction.
```

In the above example, the first GCAL directive is within the same 4-kiloword page as the subroutine entry point, so converts to a CAL instruction. The second GCAL directive is not within this range, so converts to an LCAL instruction.

4.9.4 Conversion Rules

An optimized branch directive converts to one of up to three different variants.

Short branch
Long branch
Far branch

The following are the conditions that must be met for each variant.

■Short branch

A short branch is one between -128 and +127 bytes relative to the program counter. For a directive to convert to a short branch, the following conditions must be met.

(1) The branch destination must be within the range between -128 and +127 bytes of the program counter.
(2) There must be no ORG directives between the branching address and the branch destination. In other words, both addresses must be within the same contiguous address region.

■Long branch

A long branch is one within the same 4-kiloword page but out of range for a short branch.

■Far branch

A far branch is one that is ineligible for conversion to a short or long branch.

4.9.5 Directive Expansions

The following chart lists the expansions for the optimized branch directives. It uses the symbol dest for the branch destination and the symbol next for the address following the directive.

Directive	Short branch		Long branch		Far branch	
GJMP	SJMP	dest	JMP	dest	LJMP	dest
GCAL	-		CAL	dest	LCAL	dest
GBC	BC	dest	BNC JMP	next dest	BNC LJMP	next dest
GBLT	BLT	dest	BGE JMP	next dest	BGE LJMP	next dest
GBNC	BNC	dest	BC JMP	next dest	BC LJMP	next dest
GBGE	BGE	dest	BLT JMP	next dest	BLT LJMP	next dest
GBLE	BLE	dest	BGT JMP	next dest	BGT LJMP	next dest
GBGT	BGT	dest	BLE JMP	next dest	BLE LJMP	next dest
GBG	BNG SJMP	next dest	BNG JMP	next dest	BNG LJMP	next dest
GBNG	BNG	dest	BNG SJMP skip: JMP	skip next dest	BNG SJMP skip: LJMP	skip next dest
GBZ	BZ	dest	BNZ JMP	next dest	BNZ LJMP	next dest
GBEQ	BEQ	dest	BNE JMP	next dest	BNE LJMP	next dest
GBNZ	BNZ	dest	BZ JMP	next dest	BZ LJMP	next dest
GBNE	BNE	dest	BEQ JMP	next dest	BEQ LJMP	next dest

Chapter 5

SASM Instructions

This chapter describes SASM instructions. SASM instructions are extended instructions that are more object oriented, easier to read, and easier to code than device instructions. See Chapter 6 for the details of the individual instructions.

5.1 SASM Instruction Syntax

SASM instructions are instructions that combine multiple native CPU instructions (hereinafter called basic instructions) to further enhance the object orientation of data. Coding is similar to high-level languages, so programs are easier to write and read. SASM instructions have the following statement syntax.

LABEL:	C,	[HL]	+=	3
Label definition	Option	Data object	Operator	Data object

The statement ends at the carriage return just as regular statements do. Options and label definitions are specified as necessary.

5.1.1 Data Objects

Data objects are the operands for calculations. They are either nybbles or bits. The following tables list the data objects of each type.

■ Nybble data objects

Data Object	Meaning
[n8]	Data nybble at address n8 in SFR area
[\ n8]	Data nybble at address n8 in bank given by current bank register
C:[HL]	Data nybble at address in register pair HL in bank given by current bank register
C:[XY]	Data nybble at address in register pair XY in bank given by current bank register
E:[HL]	Data nybble at address in register pair HL in bank given by extra bank register
E:[XY]	Data nybble at address in register pair XY in bank given by extra bank register
C:[HL+]	Data nybble at address in register pair HL in bank given by current bank register. Register pair incremented after access.
C:[XY+]	Data nybble at address in register pair XY in bank given by current bank register. Register pair incremented after access.
E:[HL+]	Data nybble at address in register pair HL in bank given by extra bank register. Register pair incremented after access.
E:[XY+]	Data nybble at address in register pair XY in bank given by extra bank register. Register pair incremented after access.
n4	Integer constant with value between 0 and 15

■ Bit data objects

Data Object	Meaning
[n8].n2	Bit number n2+1 (from the least significant bit) in data nybble at address n8 in SFR area
[\n8].n2	Bit number n2+1 (from the least significant bit) in data nybble at address n8 in bank given by current bank register
C:[HL].n2	Bit number n2+1 (from the least significant bit) in data nybble at address in register pair HL in bank given by current bank register
C:[XY].n2	Bit number n2+1 (from the least significant bit) in data nybble at address in register pair XY in bank given by current bank register
E:[HL].n2	Bit number n2+1 (from the least significant bit) in data nybble at address in register pair HL in bank given by extra bank register
E:[XY].n2	Bit number n2+1 (from the least significant bit) in data nybble at address in register pair XY in bank given by extra bank register
C:[HL+].n2	Bit number n2+1 (from the least significant bit) in data nybble at address in register pair HL in bank given by current bank register. Register pair incremented after access.
C:[XY+].n2	Bit number n2+1 (from the least significant bit) in data nybble at address in register pair XY in bank given by current bank register. Register pair incremented after access.
E:[HL+].n2	Bit number n2+1 (from the least significant bit) in data nybble at address in register pair HL in bank given by extra bank register. Register pair incremented after access.
E:[XY+].n2	Bit number n2+1 (from the least significant bit) in data nybble at address in register pair XY in bank given by extra bank register. Register pair incremented after access.

C: and HL can be omitted, as the following examples show.

■ Examples ■

[]	Same as C:[HL]
E:[]	Same as E:[HL]
[XY]	Same as C:[XY]
[+]	Same as C:[HL+]
[XY+]	Same as C:[XY+]
[].2	Same as C:[HL].2

5.1.2 Operators

This section lists the operators available for calculations with data objects. Some operators require two data objects; others, only one. When they take two data objects, the sizes of the two must match.

There are sometimes restrictions on the data objects that can be used with operators. For details, see Chapter 6 "SASM Instruction Details."

■ Transfer operators

Operator	Syntax	Meaning	Type
=	obj1 = obj2 bit_obj = TRUE bit_obj = FALSE	Assigns contents of obj2 to obj1. Sets bit_obj to 1. Clears bit_obj to 0.	Nybble Bit Bit
<>	obj1 <> obj2	Exchanges the contents of obj1 and obj2.	Nybble

■ Arithmetic operators

<table>
<tr><th>Operator</th><th>Syntax</th><th>Meaning</th><th>Type</th></tr>
<tr><td>+=</td><td>obj1 += obj2</td><td>Adds obj1 and obj2 and assigns the result to obj1.</td><td>Nybble</td></tr>
<tr><td>-=</td><td>obj1 -= obj2</td><td>Subtracts obj2 from obj1 and assigns the result to obj1.</td><td>Nybble</td></tr>
<tr><td>&=</td><td>obj1 &= obj2</td><td>ANDs obj1 and obj2 and assigns the result to obj1.</td><td>Nybble</td></tr>
<tr><td>|=</td><td>obj1 |= obj2</td><td>ORs obj1 and obj2 and assigns the result to obj1.</td><td>Nybble</td></tr>
<tr><td>^=</td><td>obj1 ^= obj2</td><td>XORs obj1 and obj2 and assigns the result to obj1.</td><td>Nybble</td></tr>
<tr><td>>></td><td>obj1 >> obj2</td><td>Shifts obj1 right by the value of obj2.</td><td>Nybble</td></tr>
<tr><td><<</td><td>obj1 << obj2</td><td>Shifts obj1 left by the value of obj2.</td><td>Nybble</td></tr>
<tr><td>++</td><td>obj1 ++</td><td>Increments obj1.</td><td>Nybble</td></tr>
<tr><td>– –</td><td>obj1– –</td><td>Decrements obj1.</td><td>Nybble</td></tr>
</table>

5.1.3 Options

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

An instruction can use multiple options, separated by commas. The order is irrelevant. The same option must not appear more than once.

The following tables describe the available options.

■ C (carry) option

Instruction	Function
Addition (+=)	Perform addition with carry. If the addition generates an overflow, set the carry flag.
Subtraction (-=)	Perform subtraction with carry. If the subtraction generates a borrow, set the carry flag.
Right shift (>>)	Perform right rotate through carry flag.
Left shift (<<)	Perform left rotate through carry flag.
Other	No effect.

■ D (decimal adjust) option

Instruction	Function
Addition (+=)	Perform decimal-adjusted addition with carry. If the addition generates an overflow, set the carry flag.
Subtraction (-=)	Perform decimal-adjusted subtraction with carry. If the subtraction generates a borrow, set the carry flag.
Other	No effect.

■ Examples ■

```
C,[80H]+=[]     ; Addition with carry
D,[XY]-=[]      ; Decimal-adjusted subtraction with carry
```

5.1.4 Limits on Data Objects

There are limits on operator and data object combinations. The tables below list these restrictions on data objects. Instructions also impose further limits. For complete details, see Chapter 6 "SASM Instruction Details."

■ Nybble-sized data objects

[n8], [\n8], C:[HL], C:[XY], E:[HL], E:[XY], C:[HL+], C:[XY+], E:[HL+], E:[XY+], n4

obj1 op obj2

Instruction type	op	obj1	obj2
Transfer	=	All except n4.	All
	<>	All except n4.	All except n4.
Arithmetic	+=	All except n4.	All
	-=	All except n4.	All
	&=	All except n4.	All
	|=	All except n4.	All
	^=	All except n4.	All
	>>	All except n4.	n4
	<<	All except n4.	n4
	++	All except n4.	Not applicable
	--	All except n4.	Not applicable

■ Bit-sized data objects

[n8].n2, [\ n8].n2, C:[HL].n2, C:[XY].n2, E:[HL].n2, E:[XY].n2, C:[HL+].n2,
C:[XY+].n2, E:[HL+].n2, E:[XY+].n2

obj1 op TRUE

obj1 op FALSE

Instruction type	op	obj1
Transfer	=	All

5.1.5 Special Instructions

The following table lists the SASM instructions, recognized by SASM63K, that do not fit into the categories discussed above.

Syntax	Meaning	Type
HL = n8	Assigns n8 to the contents of the HL register pair	Byte
XY = n8	Assigns n8 to the contents of the XY register pair	Byte
RA = n16	Assigns n16 to the contents of the RA register	Word
HL++	Increments the contents of the HL register pair	Byte
XY++	Increments the contents of the XY register pair	Byte
RA++	Increments the contents of the RA register	Word

5.1.6 SASM Instruction Expansion

The operation of SASM instructions is performed by expanding them into their basic instruction equivalents. The instruction [] = 1, for example, always expands into MOV C:[HL],#1. When a SASM instruction expands into multiple basic instructions and requires temporary register storage, it uses the A register or the carry flag. The programmer should always be mindful, therefore, that the use of SASM instructions can alter the contents of A register or the carry flag.

To examine exactly how SASM instructions are expanded, generate an assembly source file by running the assembler with the /A command line option. Examining this assembly source file also tells how the other flags (Z and G) are modified by the expansion of SASM instructions.

Chapter 6

SASM Instruction Details

This chapter describes the SASM instructions in detail. Be sure to read this chapter if you plan to use SASM instructions.

6.1 Nybble Assignments

■ **Syntax** ■

```
obj1 = obj2
```

■ **Function** ■

Assigns contents of obj2 to obj1.

■ **Data Objects Allowed** ■

obj1: [n8] [\n8] [HL] [XY] E:[HL] E:[XY] [HL+] [XY+] E:[HL+] E:[XY+]
obj2: [n8] [\n8] [HL] [XY] E:[HL] E:[XY] [HL+] [XY+] E:[HL+] E:[XY+] n4

■ **Options** ■

None

■ **Flags** ■

Z Indeterminate.

C No change.

G If a data object uses post-increment addressing, this flag is set to 1 if the corresponding register (HL or XY) overflows as the result of the increment and is cleared to 0 otherwise. If there is no post-increment addressing, the flag does not change.

■ **Examples** ■

Source	**Expansion**	
[ ] = 1	MOV	C:[HL],#01H
[70H] = [XY]	MOV MOV	A,C:[XY] 070H,A
[+] = 4	MOV	C:[HL+],#04H

6.2 Nybble Exchanges

■ Syntax ■

```
obj1 <> obj2
```

■ Function ■

Exchanges contents of obj2 and obj1.

■ Data Objects Allowed ■

obj1: [n8] [\n8] [HL] [XY] E:[HL] E:[XY] [HL+] [XY+] E:[HL+] E:[XY+]
obj2: [n8] [\n8] [HL] [XY] E:[HL] E:[XY] [HL+] [XY+] E:[HL+] E:[XY+]

■ Options ■

None

■ Flags ■

Z Indeterminate.

C No change. G If a data object uses post-increment addressing, this flag is set to 1 if the corresponding register (HL or XY) overflows as the result of the increment and is cleared to 0 otherwise. If there is no post-increment addressing, the flag does not change.

■ Examples ■

Source	Expansion	
[70H] <> [HL]	MOV XCH MOV	A,070H A,C:[HL] 070H,A
[HL] <> [XY]	MOV XCH MOV	A,C:[HL] A,C:[XY] C:[HL],A
[\20H] <> E:[HL+]	OR XCH XCH	\ 020H,#0 A,E:[HL+] A ,\ 020H

6.3 Nybble Additions and Subtractions

■ Syntax ■

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

■ Function ■

(1) Adds obj1 and obj2 and assigns the result to obj1.
(2) Subtracts obj2 from obj1 and assigns the result to obj1.

■ Data Objects Allowed ■

obj1: [n8] [\n8] [HL] [XY] E:[HL] E:[XY] [HL+] [XY+] E:[HL+] E:[XY+]
obj2: [n8] [\n8] [HL] [XY] E:[HL] E:[XY] [HL+] [XY+] E:[HL+] E:[XY+] n4

■ Options ■

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

D Perform addition or subtraction with carry and decimal adjust. If the result generates an overflow or borrow, set the carry flag.

■ Flags ■

(1)

Z This flag is set to 1 if the result of the addition is zero and is cleared to 0 otherwise.

C This flag is set to 1 if the result generates an overflow and is cleared to 0 otherwise.

G If a data object uses post-increment addressing, this flag is set to 1 if the corresponding register (HL or XY) overflows as the result of the increment and is cleared to 0 otherwise. If there is no post-increment addressing, the flag does not change.

(2)

Z   This flag is set to 1 if the result of the subtraction is zero and is cleared to 0 otherwise.

C   This flag is set to 1 if the result generates a borrow and is cleared to 0 otherwise.

G   If a data object uses post-increment addressing, this flag is set to 1 if the corresponding register (HL or XY) overflows as the result of the increment and is cleared to 0 otherwise. If there is no post-increment addressing, the flag does not change.

■ Examples ■

Source	Expansion	
[HL] += 4	ADD	C:[HL],#04H
C,[XY+] += 3	MOV ADC	A,#03H C:[XY+],A
D, [\ 20H] += E:[XY+]	MOV ADCD	A,E:[XY+] \ 020H,A
E:[XY+] -= 8	SUB	E:[XY+],#08H
E:[HL] -= [\30H]	OR SUB	\ 030H,#0 E:[HL],A

6.4 Nybble Logical Operations

■ **Syntax** ■

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

■ **Function** ■

(1) ANDs obj1 and obj2 and assigns the result to obj1.
(2) ORs obj1 and obj2 and assigns the result to obj1.
(3) XORs obj1 and obj2 and assigns the result to obj1.

■ **Data Objects Allowed** ■

obj1: [n8] [\n8] [HL] [XY] E:[HL] E:[XY] [HL+] [XY+] E:[HL+] E:[XY+]
obj2: [n8] [\n8] [HL] [XY] E:[HL] E:[XY] [HL+] [XY+] E:[HL+] E:[XY+] n4

■ **Options** ■

None

■ **Flags** ■

Z This flag is set to 1 if the result is zero and is cleared to 0 otherwise.

C No change.

G If a data object uses post-increment addressing, this flag is set to 1 if the corresponding register (HL or XY) overflows as the result of the increment and is cleared to 0 otherwise. If there is no post-increment addressing, the flag does not change.

■ **Examples** ■

Source	Expansion	
[HL] &= [\20H]	OR AND	\020H,#0 C:[HL],A
[XY] |= 0AH	OR	C:[XY],#0AH
[HL+] ^= 3	XOR	C:[HL+],#03H

6.5 Nybble Shifts

■ Syntax ■

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

■ Function ■

(1) Shifts obj1 right by the value of obj2.
(2) Shifts obj1 left by the value of obj2.

■ Data Objects Allowed ■

obj1: [n8] [\n8] [HL] [XY] E:[HL] E:[XY] [HL+] [XY+] E:[HL+] E:[XY+]
obj2: n2

■ Options ■

C

(1) Rotate through carry: The carry moves into the top bit of obj1, and the bottom bit of obj1 moves into the carry bit.

(2) Rotate through carry: The carry moves into the bottom bit of obj1, and the top bit of obj1 moves into the carry bit.

■ Flags ■

(1)

Z This flag is set to 1 if the result of the shift is zero and is cleared to 0 otherwise.

C This flag holds the last LSB shifted.

G If a data object uses post-increment addressing, this flag is set to 1 if the corresponding register (HL or XY) overflows as the result of the increment and is cleared to 0 otherwise. If there is no post-increment addressing, the flag does not change.

(2)

Z This flag is set to 1 if the result of the shift is zero and is cleared to 0 otherwise.

C This flag holds the last MSB shifted.

G If a data object uses post-increment addressing, this flag is set to 1 if the corresponding register (HL or XY) overflows as the result of the increment and is cleared to 0 otherwise. If there is no post-increment addressing, the flag does not change.

■ Examples ■

Source	Expansion	
[HL] >> 1	FCLR ROR	C C:[HL]
C,[HL] >> 1	ROR	C:[HL]
C,[XY] >> 2	ROR ROR	C:[XY] C:[XY]
[\30H] << 2	FCLR ROL FCLR ROL	C \ 030H C \ 030H

6.6 Nybble Increments and Decrements

■ Syntax ■

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

■ Function ■

(1) Increments obj1.
(2) Decrements obj1.

■ Data Objects Allowed ■

obj1: [n8] [\n8] [HL] [XY] E:[HL] E:[XY] [HL+] [XY+] E:[HL+] E:[XY+]

■ Option ■

None

■ Flags ■

(1)

Z This flag is set to 1 if the result of the increment is zero and is cleared to 0 otherwise.

C This flag is set to 1 if the result generates an overflow and is cleared to 0 otherwise.

G If a data object uses post-increment addressing, this flag is set to 1 if the corresponding register (HL or XY) overflows as the result of the increment and is cleared to 0 otherwise. If there is no post-increment addressing, the flag does not change.

(2)

Z This flag is set to 1 if the result of the decrement is zero and is cleared to 0 otherwise.

C This flag is set to 1 if the result generates a borrow and is cleared to 0 otherwise.

G If a data object uses post-increment addressing, this flag is set to 1 if the corresponding register (HL or XY) overflows as the result of the increment and is cleared to 0 otherwise. If there is no post-increment addressing, the flag does not change.

■ **Examples** ■

Source	Expansion	
[HL] ++	INC	C:[HL]
[\70H] ++	INC	\ 070H
[XY] – –	DEC	C:[XY]
[30H] – –	DEC	030H

6.7 Bit Assignments

■ Syntax ■

```
obj1 = obj2
```

■ Function ■

Assigns contents of obj2 to obj1.

■ Data Objects Allowed ■

obj1: [n8].n2 [\ n8].n2 [HL].n2 [XY].n2 E:[HL].n2 E:[XY].n2 [HL+].n2 [XY+].n2 E:[HL+].n2 E:[XY+].n2
obj2: TRUE FALSE

■ Options ■

None

■ Flags ■

Z Indeterminate.

C No change.

G If a data object uses post-increment addressing, this flag is set to 1 if the corresponding register (HL or XY) overflows as the result of the increment and is cleared to 0 otherwise. If there is no post-increment addressing, the flag does not change.

■ Examples ■

Source	Expansion	
[HL].2 = TRUE	BSET	C:[HL].2
[20H].2 = TRUE	MOV OR	A,#(1<<2) 020H,A
E:[XY+].1 = FALSE	BCLR	E:[XY+].1

6.8 Special Instructions

■ **Syntax** ■

(1) `HL = n8`
(2) `XY = n8`
(3) `RA = n16`
(4) `HL++`
(5) `XY++`
(6) `RA++`

■ **Function** ■

(1) Assigns n8 to the contents of the HL register pair. n8 is a constant between 0 and 0FFH.
(2) Assigns n8 to the contents of the XY register pair. n8 is a constant between 0 and 0FFH.
(3) Assigns n16 to the contents of the RA register. n16 is a constant between 0 and 0FFFFH.
(4) Increments the contents of the HL register pair.
(5) Increments the contents of the XY register pair.
(6) Increments the contents of the RA register.

■ **Options** ■

None

■ **Flags** ■

(1) Flags do not change.

(2) Flags do not change.

(3) Flags do not change.

(4)

Z No change.

C No change.

G This flag is set to 1 if the HL register overflows as the result of the increment and is cleared to 0 otherwise.

(5)

Z No change.

C No change.

G This flag is set to 1 if the XY register overflows as the result of the increment and is cleared to 0 otherwise.

(6)

Z No change.

C No change.

G This flag is set to 1 if the RA register overflows as the result of the increment and is cleared to 0 otherwise.

■ Examples ■

Source	Expansion	
HL = 5	MOV MOV	H,#((05H>>4)&0FH) L,#(05H&0FH)
XY = 10H	MOV MOV	H,#((010H>>4)&0FH) L,#(010H&0FH)
RA = 10FFH	MOV MOV MOV MOV	RA0,#(010FFH&0FH) RA1,#((010FFH>>4)&0FH) RA2,#((010FFH>>8)&0FH) RA3,#((010FFH>>12)&0FH)
HL ++	INCB	HL
XY ++	INCB	XY
RA ++	INCW	RA

Chapter 7

Control Statements

This chapter describes the control statements available for controlling program flow.

7.1 Bit Expressions

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

Bit expressions cannot stand on their own. They are always used as part of a flow control statement.

7.1.1 Structural Elements of Bit Expressions

The following table summarizes the basic bit expressions.

Syntax	Description
obj1 == obj2	1 if obj1 is equal to obj2; 0 otherwise.
obj1 != obj2	1 if obj1 is not equal to obj2; 0 otherwise.
obj1 > obj2	1 if obj1 is greater than obj2; 0 otherwise.
obj1 >= obj2	1 if obj1 is greater than or equal to obj2; 0 otherwise.
obj1 < obj2	1 if obj1 is less than obj2; 0 otherwise.
obj1 <= obj2	1 if obj1 is less than or equal to obj2; 0 otherwise.
bit_obj == TRUE	1 if bit_obj is 1; 0 otherwise.
bit_obj == FALSE	1 if bit_obj is 0; 0 otherwise.
bit_obj != TRUE	1 if bit_obj is not 1; 0 otherwise.
bit_obj != FALSE	1 if bit_obj is not 0; 0 otherwise.
bit_obj	1 if bit_obj is 1; 0 otherwise.
TRUE	Always 1.
FALSE	Always 0.

obj1 and obj2 are nybble-sized data objects from the list in Section 5.1.1 "Data Objects." They cannot, however, use post-increment addressing. obj1 cannot be a constant.

bit_obj is a bit-sized data object from the list in Section 5.1.1 "Data Objects" or one of the flags: C (carry flag), Z (zero flag), or G (G flag). It cannot, however, use post-increment addressing.

7.1.2 Operators in Bit Expressions

Combining bit expressions with operators yields another bit expression. The following table lists the operators available for building bit expressions.

<table>
<tr><th>Operator</th><th>Syntax</th><th>Description</th></tr>
<tr><td>!</td><td>!bit_expr1</td><td>1 if bit_expr1 is 0; 0 otherwise.</td></tr>
<tr><td>&&</td><td>bit_expr1 && bit_expr2</td><td>1 if both bit_expr1 and bit_expr2 are 1; 0 otherwise.</td></tr>
<tr><td>||</td><td>bit_expr1 || bit_expr2</td><td>0 if both bit_expr1 and bit_expr2 are 0; 1 otherwise.</td></tr>
</table>

When joining bit expressions with the operators in the above table, place parentheses around the component expressions as leaving off the parentheses sometimes produces syntax errors. Such syntax errors result when the component expression contains a constant expression.

■ **Examples** ■

[]==3 && [xy]==1 This form, without parentheses, produces an error message.

([]==3) && ([xy]==1) The parentheses ensure proper evaluation.

The operators have the following order of precedence.

<table>
<tr><th>Precedence</th><th>Operator</th></tr>
<tr><td>1</td><td>( )</td></tr>
<tr><td>2</td><td>!</td></tr>
<tr><td>3</td><td>&&</td></tr>
<tr><td>4</td><td>||</td></tr>
</table>

7.2 Control Statement Types

The expansions of control statements always include jump instructions. Control statements therefore offer a choice of four variants to match the variety of jump instructions available: short, long, far, and optimized.

The following table describes the relationship between the flow control statement type and the jump instruction used in the expansion.

Flow control statement type	Jump instruction	Description
Short	SJMP	PC-relative jump. The difference between the address of the next instruction and the jump target must be within the range between -128 and +127.
Long	JMP	Jump within the same 4-kiloword page. The branch destination must be within the same 4-kiloword page as the branching address.
Far	LJMP	Far jump. The jump target can be anywhere in the program memory.
Optimized	SJMP, JMP, or LJMP	The assembler automatically selects the optimal jump variant.

The following flow control statements are available.

- IF-ELSE-ELSEIF statement
- WHILE statement
- REPEAT-UNTIL statement
- SWITCH-CASE statement
- FOR statement
- BREAK statement
- CONTINUE statement

Since the BREAK and CONTINUE statements are used within the other flow control statements—that is, they never appear alone—they take their type from the surrounding flow control statement and do not have different notations for the various types.

7.3 IF-ELSE-ELSEIF Statement

■ **Syntax** ■

```
IF/SIF/LIF/FIF bit_expression
       .
       .
       .
  [ELSEIF bit_expression]
       .
       .
       .
  [ELSE]
       .
       .
       .
ENDI
```

The Optimized type uses IF; the short type, SIF; the long type, LIF; the far type, FIF.

■ **Function** ■

The result of evaluating the bit expression controls which statement block is executed.

ELSEIF and ELSE are optional, but the IF and ENDI must always be present. There can be multiple ELSEIF clauses.

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

If the bit expression is false, the process is repeated for the bit expressions after each ELSEIF. If one is true, the statement block following the ELSEIF is executed. If none are true, the statement block following the ELSE is executed.

■ Example 1 ■

```
IF ( [\30].1 == TRUE )
        [HL] = 3
ENDI
```

If the second bit from the bottom of the nybble data addressed by [\30H] is 1, the statement [HL] = 3 is executed. If the bit is not 1, however, nothing happens.

■ Example 2 ■

```
IF ([HL] != 0) && ([HL] == [XY])
        [HL]++
ELSE
        [XY]++
ENDI
```

If [HL] is not zero and [HL] equals [XY], the statement [HL]++ is executed. Otherwise, the statement [XY]++ is executed.

7.4 WHILE Statement

■ **Syntax** ■

```
WHILE/SWHILE/LWHILE/FWHILE bit_expression
       .
       .
       .
ENDW
```

The Optimized type uses WHILE; the short type, SWHILE; the long type, LWHILE; the far type, FWHILE.

■ **Function** ■

The WHILE statement causes the statement block between the WHILE and the ENDW to be repeatedly executed while the bit expression is true. The WHILE statement differs from the REPEAT-UNTIL statement described below in that the condition is checked before entering the first iteration. If the bit expression is false at the start, therefore, control exits the WHILE statement block without executing the statements even once.

■ **Example** ■

```
WHILE ( [HL]==0 )
        HL++
ENDW
```

While [HL] == 0, the HL register is incremented.

7.5 REPEAT-UNTIL Statement

■ **Syntax** ■

```
REPEAT/SREPEAT/LREPEAT/FREPEAT
              .
              .
              .
UNTIL bit_expression
```

The Optimized type uses REPEAT; the short type, SREPEAT; the long type, LREPEAT;the far type, FREPEAT.

■ **Function** ■

The REPEAT-UNTIL statement causes the statement block between the REPEAT and the UNTIL to be repeatedly executed while the bit expression is true. The REPEAT-UNTIL statement differs from the WHILE statement in that the statement block is executed once before the condition is checked. Even if the bit expression is false at the start, the REPEAT-UNTIL statement block is executed once.

■ **Example** ■

```
REPEAT
        XY++
        [HL] += 2
UNTIL ([HL] != [30H])
```

The statement block between the REPEAT and the UNTIL is executed. It is then repeated until [HL] is equal to [30H].

7.6 SWITCH-CASE Statement

■ **Syntax** ■

```
SWITCH/SSWITCH/LSWITCH/FSWITCH obj
CASE constant_expression
        .
        .
        .
   [CASE constant_expression]
        .
        .
        .
   [DEFAULT]
        .
        .
        .
ENDS
```

obj is one of the following nybble-sized data objects.

[n8] [\n8] [HL] [XY] E:[HL] E:[XY]

The Optimized type uses SWITCH; the short type, SSWITCH; the long type, LSWITCH; the far type, FSWITCH.

■ **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 expressions after each CASE. If there is a match, the statements following that CASE are executed, and control exits the entire SWITCH statement. If there are no matches, the statements after the DEFAULT are executed instead. If there is no DEFAULT, nothing is executed. Multiple statements can appear after each CASE. There can also be none.

■ **Example** ■

```
SWITCH [\30H]
CASE 1
         [\WORK] = 5
CASE 2
         [\WORK] = 0AH
CASE 3
         [\WORK] = 3
DEFAULT
         [\WORK] = 0
ENDS
```

The above assigns to [\WORK] a value that depends on the value in [\30H]. The value assigned is 5 for a value of 1 in [\30H], 0AH for 2, 3 for 3, and 0 otherwise.

7.7 FOR Statement

■ Syntax ■

```
FOR/SFOR/LFOR/FFOR obj = constant_expression1, constant_expression2
              .
              .
              .
ENDF
```

obj is one of the following nybble-sized data objects.

[n8] [\n8] [HL] [XY] E:[HL] E:[XY]

The Optimized type uses FOR; the short type, SFOR; the long type, LFOR; the far type, FFOR.

■ Function ■

The FOR statement initializes obj to constant_expression1 and repeatedly executes the statement block between the FOR and the ENDF, incrementing obj each time, until obj equals constant_expression2. If obj overflows before it reaches the value constant_expression2, control exits the FOR statement block.

■ Example ■

```
FOR [L] = 0, 0FH
        [HL] = 0
ENDF
```

The statement block between the FOR and the ENDF is executed repeatedly for values of [L] from 0 through 0FH.

■ Warning ■

Using [HL+], [XY+], E:[HL+], or E:[XY+] for obj produces unpredictable results.

7.8 BREAK Statement

■ Syntax ■

```
BREAK
```

■ Function ■

The BREAK statement is used inside SWITCH statement blocks and iteration blocks for the FOR, WHILE, and REPEAT statements. It causes control to exit the innermost statement block and pass to the statement following the end of the corresponding statement block.

■ Example ■

```
WHILE ([HL+] == 0)
        [XY+] = 0
        [\20H]++
        IF ([\20H] == 0AH)
                BREAK
        ENDI
ENDW
```

When [\20H] equals 0AH, control exits the WHILE statement's iteration block, regardless of the truth value of the WHILE statement's iteration condition.

7.9 CONTINUE Statement

■ **Syntax** ■

```
CONTINUE
```

■ **Function** ■

The CONTINUE statement is used inside SWITCH statement blocks and iteration blocks for the FOR, WHILE, and REPEAT statements. It causes control to pass to the end of the innermost statement block. After execution of the CONTINUE statement, the iteration condition is checked.

■ **Example** ■

```
WHILE ([\20H]!=0AH)
        [\20H]++
        IF ([\20H] == 5)
                CONTINUE
        ENDI
        [XY+]=[\20H]
ENDW
```

If [\20H] is equal to 5, the statement [XY+] = [\20H] is skipped, and control passes to the check of the iteration condition [\20H] != 0AH.

Chapter 8

Error Messages

This chapter describes the SASM63K error messages.

8.1 Syntax Errors

Syntax errors detected during analysis of the source file result in the output of error messages to the screen or error file.

The following table lists the error codes, error messages, and their meanings.

Code	Error message
E01	**mnemonic not allowed** Instruction mnemonics can only appear in a CODE segment.
E02	**bad syntax** This error occurs at the start of analysis. There is no recognizable instruction.
E03	**newline in string** A character string contains a newline code (CR-LF).
E04	**unexpected EOF** A character constant or character string is missing the closing quotation mark.
E06	**operand not allowed** There is an operand in a directive that does not support operands.
E07	**xdata segment only** This directive can only appear in an external memory segment.
E09	**bad character XX(hex)** The source file contains an illegal character.
E10	**bad character in string or character constant. XX(hex)** A string or character constant contains an illegal character.
E11	**redefinition XXXXXX** The indicated symbol has been redefined.
E12	**undefined XXXXXX** There is no definition for the indicated symbol.
E13	**bad operand** There is an error in operand syntax. If the statement is a microcontroller instruction, the cause could be a mistake in the addressing mode specification or too many or too few operands. If the statement is a directive, the operands probably do not match the directive syntax.

Code	Error message
E14	**out of range** The operand value is outside the range allowed.
E15	**bad const** There is a mistake in the notation for a constant.
E17	**string or character constant. too long** A string or character constant is longer than the specified limit.
E18	**divide by zero** The denominator in an expression is zero.
E19	**bad location** The location specified is outside the allowed range.
E20	**invalid instruction** The program uses an instruction that is not defined with a DCL file #INSTRUCTION statement.
E21	**absolute expression required** The operand must be a constant expression.
E22	**segment type mismatch** The segment types do not match.
E23	**external memory not exist** The target microcontroller does not support external memory.
E24	**declaration duplicated** There is more than one TYPE directive.
E27	**out of SFR limit** There is a mistake in the SFR address specified in the DCL file. If this message arises with a DCL file supplied by Oki Electric Industry, there is most likely something wrong with the file. Please report the problem to us.
E32	**cannot use this object** This instruction does not accept this data object.
E33	**code segment only** This instruction can only appear in a CODE segment.

Code	Error message
E34	**missing ENDM** A macro definition is missing the closing ENDM.
E35	**name required** A name is required.
E36	**missing statement XXXXXX** The indicated statement is missing.
E37	**not macro name** This name is not a macro name.
E38	**macro definition not found** There is no macro definition for the called macro.
E39	**missing)** The statement is missing a closing parenthesis.
E40	**parameters mismatch** The macro call has the wrong number of arguments.

8.2 Warning Messages

Warning messages are generated for statements that are syntactically correct, but for which the assembly results may not be reliable. The output format for the listing file, screen, and error file is the same as for syntax errors.

The following table lists the warning codes, warning messages, and their meanings.

Code	Warning message
W01	**out of using bank** The operand's bank number does not match that given with the USING BANK directive.
W02	**illegal sfr read** The statement contains an invalid access to the SFR area. This warning arises when an instruction attempts to read from an SFR area that does not allow reads.
W03	**illegal sfr write** The statement contains an invalid access to the SFR area. This warning arises when an instruction attempts to write to an SFR area that does not allow writes.
W04	**do not put out code** This SASM instruction does not generate object code.

8.3 Fatal Errors

Fatal errors interfere with SASM63K execution. When it detects a fatal error, SASM63K displays the corresponding message on the screen and aborts execution.

Code	Fatal error message
F01	**file not found** The source file was not found.
F02	**file can't create** The assembler was unable to create an output file.
F04	**memory is not enough** There is not enough memory to continue processing. One possible cause of this error is that the source program defines too many symbols. If you have any TSR programs resident, try removing them. If you are using the /R or /S command line options, try removing them. If the error persists after these measures, the current version of SASM63K is unable to process your file. Take steps to reduce the number of symbols.
F05	**line overflow** The number of source statements exceeds 65,534.
F06	**bad syntax in command line** There is an error in the command line options.
F07	**DCL file not found** The DCL file was not found.
F08	**error(s) found in DCL file** The DCL file contains one or more syntax errors. Since assembler results cannot be guaranteed, the assembler aborts with this message. This error should not occur if you use the original DCL files supplied by Oki Electric Industry.
F09	**TYPE directive required** The TYPE directive is missing.

Code	Fatal error message
F11	**too many include or macro nesting levels** The program nests include files or macros too deeply.
F12	**I/O error writing file** The source level debug file could not be written to.
F13	**file can’t open** The file cannot be open.
F14	**file can’t close** The .file cannot be close.
F15	**file can’t read** The file cannot be read..

Chapter 9

Output Files

This chapter describes the four types of output files produced by SASM63K: object files, print files, error files, and assembly source files.

9.1 Object Files

Object files contain object code and optional debugging information.

The following table list the two different types of object files.

HEX file type	Memory space	Extensions
Byte-divided HEX files	Program memory	.HXH and .HXL
Intel HEX format files	External memory	.H00 to .HFF

The first type of files are Intel HEX format files with Oki extensions. Such extensions are necessary because the 8-bit Intel HEX format does not support the 16-bit data width of the OLMS-63K Series' program memory. The second type are Intel HEX format files.

The debugging information is for use with a symbolic debugger running the emulator. It appears at the beginning of the byte-divided HEX file with the extension .HXH.

If the assembler detects even one syntax error, it generates no object files.

If the microcontroller does not have external memory, the assembler produces no Intel HEX format files.

If the microcontroller has external memory, the assembler always produces an Intel HEX format file with the extension .H00.

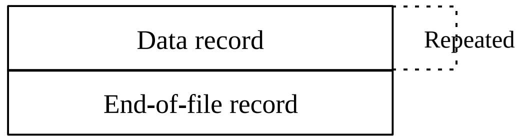
The assembler produces Intel HEX format files with extensions .H01 through .HFF when the source program contains ORG directives specifying external memory banks. The extension on the Intel HEX format file reflects the bank number specified with the ORG instruction. If an ORG directive specifies external memory bank 3, for example, the corresponding Intel HEX format file has the extension .H03.

The following sections describe the formats of these HEX files and the debugging information. They first give the file structure and then describe the record format. The record descriptions give sample output, divide it into fields, and then describe the fields.

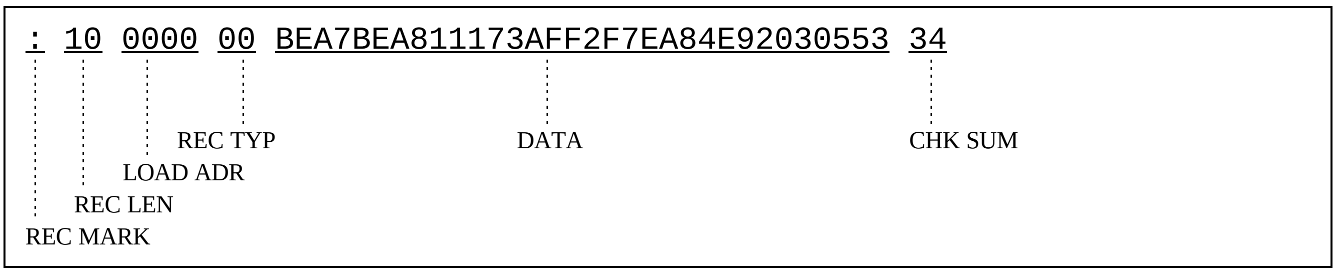
9.1.1 Byte-Divided HEX Files

The byte-divided HEX files divide the words containing the object code into separate files containing the high and low bytes.

● Structure of byte-divided HEX files

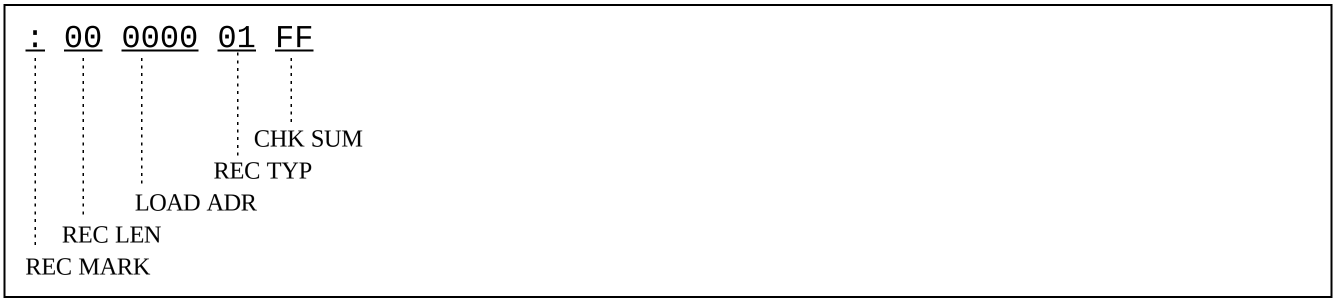


● Data records



Field	Description
REC MARK	Colon character (:)
REC LEN	Number of object code bytes stored in the DATA field
LOAD ADR	Load address for the first byte of object code in the DATA field
REC TYP	Always “00” for a data record
DATA	Object code field. The upper byte file (with extension .HXH) contains the upper bytes of the object code; the lower byte file (with extension .HXL), the lower bytes.
CHK SUM	Checksum

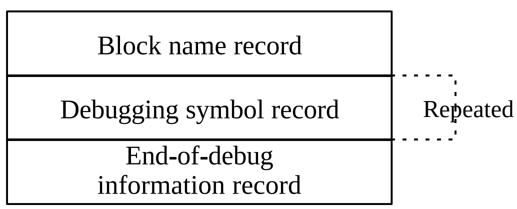
● **End-of-file record**



Field	Description
REC MARK	Colon character (:)
REC LEN	Always “00”
LOAD ADR	Always “0000”
REC TYP	Always “01” for a end-of-file record
CHK SUM	Always “FF”

9.1.2 Debugging Information

Specifying the /D command line option or inserting a DEBUG directive into the source program causes the assembler to output debugging information to the upper byte member (with extension .HXH) of the two byte-divided HEX files.



● Block name record

```
1 MODULE
:       :
:       BLOCK NAME
REC MARK
```

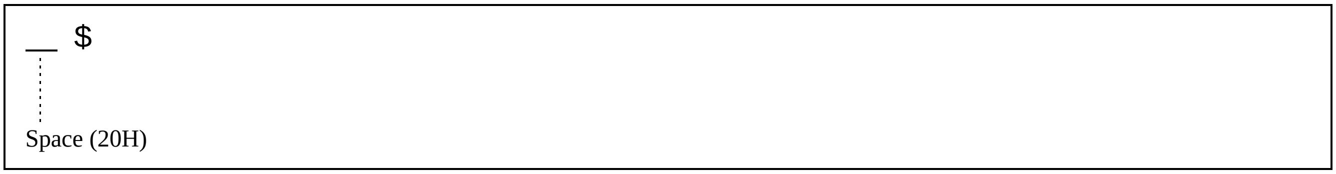
Field	Description
REC MARK	Character “1” for a block name record
BLOCK NAME	Symbol giving the module name

● Debug symbol record

```
0 DEBUGSYM 10H X F
:    :      :  : :
:    :      :  : BANK
:    :      :  SEG
:    :      VALUE
:    SYMBOL
REC MARK
```

Field	Description
REC MARK	Character “0” for a debugging symbol record
SYMBOL	User-defined symbol
VALUE	Value for symbol in hexadecimal
SEG	Symbol type as determined from how it was defined: C Symbol allocated in CODE address space D Symbol allocated in DATA address space X Symbol allocated in XDATA address space B Symbol allocated in BIT address space N Symbol defined with EQU or SET directive
BANK	External memory bank number (0-FFH) for symbol. This field only appears when the SEG field contains X.

● **End-of-debug information record**

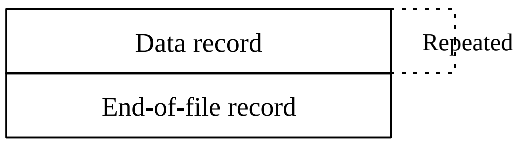


This record indicates the end of the debugging information. It consists solely of a space and a dollar sign (24H).

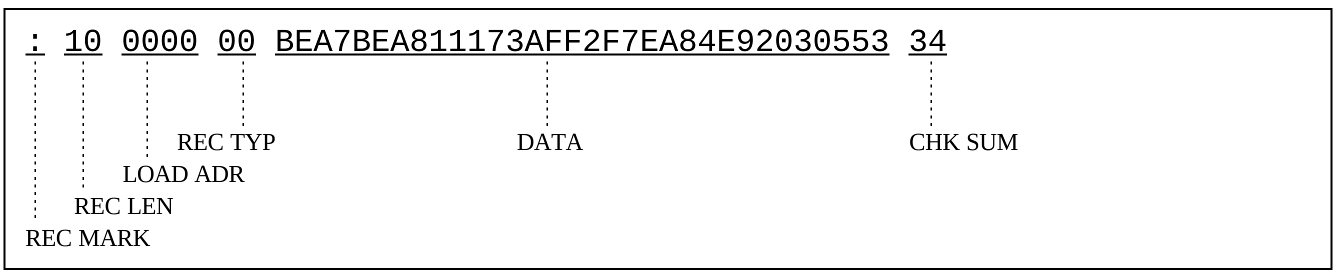
9.1.3 Intel HEX Format Files

The Intel HEX format files contain external memory initialization data in Intel HEX format.

● **Structure of Intel HEX format file**



● **Data records**



Field	Description
REC MARK	Colon character (:) (3AH)
REC LEN	Number of external memory initialization data bytes stored in the DATA field
LOAD ADR	Load address for the first byte of initialization data in the DATA field
REC TYP	Always “00” for a data record
DATA	External memory initialization data
CHK SUM	Checksum

● End-of-file record

```
: 00 0000 01 FF
:  :    :  :  :
:  :    :  :  CHK SUM
:  :    :  REC TYP
:  :    LOAD ADR
:  REC LEN
REC MARK
```

Field	Description
REC MARK	Colon character (:) (3AH)
REC LEN	Always “00”
LOAD ADR	Always “0000”
REC TYP	Always “01” for a end-of-file record
CHK SUM	Always “FF”

9.2 Print File

The print file is a sequential file of variable-length records separated by carriage returns. The disk output file has the same name as the source file, but the extension .PRN.

There are two types of output formats for print file. User chooses one of them by /PR or /PR1 option. When the /PR option is specified , SASM63K generates a print file which includes the expanded source of SASM instructions, control statements, Branch directives, DB/DW directives and preprocessor directives. When the /PR1 option is specified , SASM63K generates a print file which does not include the expanded source.

The following is a sample print file. The numbers down the left side refer to the notes that follow.

```
(1)  << SASM63K >> Structured-Macro-Assembler, Ver.2.21
(2)    page : 1
(3)    file : SAMPLE1.ASM
(4)    date : 97 02/19 Wed. [18:40]
(5)   title : Sample program
(6)   Loc  Code          Line  Source statements

(7)   ******************** SAMPLE1.ASM ********************
                            1: TYPE ( M63184 )
                            2: TITLE "Sample program"
                            3:
                            4: INCLUDE ( SYMBOL .DEF )
      ******************** SYMBOL.DEF ********************
                            1: NOLIST
      ******************** SAMPLE1.ASM ********************
                            5:
       0100                 6:        ORG    100H
       0100                 7: START:
                            8:        HL = 0
(8)    0100 0130                      MOV    H ,#((00H>>4)&0FH)
       0101 0120                      MOV    L ,#(00H&0FH)
                            9:        XY = 0
       0102 0110                      MOV    X ,#((00H>>4)&0FH)
       0103 0100                      MOV    Y ,#(00H&0FH)
                           10:        [CBR] = 2
       0104 0032                      MOV    CBR ,#02H
                           11:        USING BANK 2
                           12:        [\200H] = 0FH
       0105 4F00                      MOV    \0200H ,#0FH
                           13:
                           14:        WHILE ([\200H]!=0)
       0106                    ?00001:
       0106 A000                      CMP    \0200H ,#00H
       0107 0C02                      BEQ    ?00002
                           15:               [\300H]--
(9)    0108 3100                      DEC    \300H
      SAMPLE1.ASM(15) : Warning : W01 : out of using bank
```

The following notes refer to the numbers on the above print file.

(1) The file starts off by giving the assembler version number.

(2) This line gives the page number.

(3) This line gives the source file name.

(4) This line gives the date specified with the DATE directive. If there is no DATE directive, the assembler uses the date from the operating system.

(5) This line gives the title specified with the TITLE directive.

(6) This line gives the headings for the source code listing. The fields have the following meanings.

Field	Meaning
Loc	Location counter value as 4-digit hexadecimal number.
Code	Object code in hexadecimal. In the CODE segment, this is in (16-bit) words; in the XDATA segment, it is in bytes.
Statement	Assembly language statements: instructions; DB, DW, DS, DBIT, and ORG directives; label definitions, etc. Constant expressions are shown after evaluation.
Line	Source file line number in decimal.
Source	Source statement.

(7) These lines give the source file name.

(8) These lines show the expanded source of SASM instruction. When the print file is generated by /PR option or PRN directive, the expanded source of SASM instructions , control statements, Branch directive, DB/DW directives and preprocessor directives are output at the field of Source statements. When the print file is generated by /PR1 option, the expanded source isn't output.

(9) These lines show the format of error and warning messages.

9.3 Cross Reference List

The cross reference list gives both symbol information and a reference table. It shows where each symbol is defined and referenced. Symbols are listed in alphabetical order.

The cross reference list follows the assembly list in the output. Symbols written in lower case in the source program are converted to upper case for output.

The following is an example of a cross reference list.

```
                                                  (2)
(1)        ---- cross reference list ----          :
 :                                                 :
 :.........BIT_SYM0...................... SYMBOL.DEF(2)
(3)...............  SAMPLE1.ASM(25)
           START......................... SAMPLE1.ASM(7)
           VAL1.......................... SAMPLE1.ASM(22)
                    SAMPLE1.ASM(44)
                    SAMPLE1.ASM(57)
           VAL2.......................... undefined symbol
                    SAMPLE1.ASM(47)
           XDATA_SYM0.................... SYMBOL.DEF(3)
                    SAMPLE1.ASM(32)
```

The following table describes the individual fields.

Field	Description
(1)	Symbol.
(2)	File name and line number (decimal) where symbol defined. If the symbol is undefined, the notation "undefined symbol" appears instead.
(3)	File name and line number (decimal) where symbol referenced.

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                            atr    value
BIT_SYM0....................... BIT     07FF
START.......................... CODE    0100
VAL1........................... NUMBER  0001
VAL2........................... UNDEF   0000
XDATA_SYM0..................... XDATA   0010
```

The following describes the individual fields.

The name field gives the name of the symbol.

The atr field gives the type of the symbol. This type depends on how the symbol was defined.

Type	Description
CODE	Symbol allocated in CODE space
DATA	Symbol allocated in DATA space
BIT	Symbol allocated in BIT space
XDATA	Symbol allocated in XDATA space
NUMBER	Symbol defined with EQU or SET directive
UNDEF	Undefined symbol

9.5 Error File

The error file contains error messages and the statement that generated them. The error message appears before the corresponding source statement.

For the meanings of the error messages, see Chapter 8 "Error Messages."

The following is an example of error message output.

```
    (1)     (2)     (3)      (4)       (5)
     :       :       :        :         :
SAMPLE.ASM(24) : Error : E12 : undefined UNDEF_SYM
   DW  UNDEF_SYM.......
                      :
                     (6)
```

Field	Description
(1)	Source file name.
(2)	Line number of statement generating error.
(3)	Error level: Error or warning.
(4)	Error code.
(5)	Error message. For a listing of error codes and error messages, see Chapter 8 "Error Messages."
(6)	Source statement.

9.6 Assembly Source File

The assembly source file is the SASM63K source file converted to a source file suitable for input to the ASM63KN assembler.

The assembly source file can be assembled with ASM63KN Ver. 1.01 or higher or by SASM63K itself. When assembling with SASM63K, makes sure that the output file does not have the same name as the input file.

The following is an example of an assembly source file.

```
;<<SASM63K>> Structured-Macro-Assembler, Ver.2.21
;file :SAMPLE2.ASM

       TYPE (M63184)
       TITLE "Sample program"
;SAMPLE1.ASM(3):
;SAMPLE1.ASM(4):       INCLUDE (SYMBOL.DEF)
       NOLIST
       BIT_SYM0       BIT    01FFH.3
       XDATA_SYM0       XDATA   10H
       LIST
;SAMPLE1.ASM(5):
;SAMPLE1.ASM(6):       DEFINE RESET_DATA 0
;SAMPLE1.ASM(7):       MACRO FCLR_FLAG()
;SAMPLE1.ASM(8):             FCLR    G
;SAMPLE1.ASM(9):             FCLR    C
;SAMPLE1.ASM(10):            FCLR    Z
;SAMPLE1.ASM(11):      ENDM
;SAMPLE1.ASM(12):
       ORG    100H
START:
;SAMPLE1.ASM(15):      HL = 0
       MOV    H,#((00H>>4)&0FH)
       MOV    L,#(00H&0FH)

;SAMPLE1.ASM(16):      XY = 0
       MOV    X,#((00H>>4)&0FH)
       MOV    Y,#(00H&0FH)

;SAMPLE1.ASM(17):
;SAMPLE1.ASM(18):      [CBR] = 2
       MOV    CBR,#02H

;SAMPLE1.ASM(19):      [\ 200H] = 0FH
       MOV    \0200H,#0FH

;SAMPLE1.ASM(20):      WHILE( [\ 200H] != 0 )
?00001:
       CMP    \0200H,#00H
       BEQ    ?00002
```

Chapter 10

Sample Program

This chapter describes a sample application program developed with SASM63K.

10.1 Sample Program Specifications

This chapter describes an application program for SASM63K. If you use all or part of this sample program, be sure to take into account other conditions and debug it.

10.1.1 Sample Program Function

The sample program is a timer program that counts from 00:00 to 99:59 in one-second increments.

10.1.2 Program Specifications

The minutes and seconds data are stored in five nybbles in bank 15 of data memory. (See Figure 10-1.) The LCD display uses LCD driver outputs COM1 to COM8 and SEG0 to SEG19. (See Figure 10-2.)

To count time, the program checks the 2-Hz interrupt request flag in bank 0 of data memory. (See Figure 10-3.) If the flag is 1, the program increments the half-second count. When this count reaches 2, the program clears it and increments the seconds count, which is stored as two BCD digits. When the seconds count reaches 60, the program increments the minutes count, another two BCD digits.

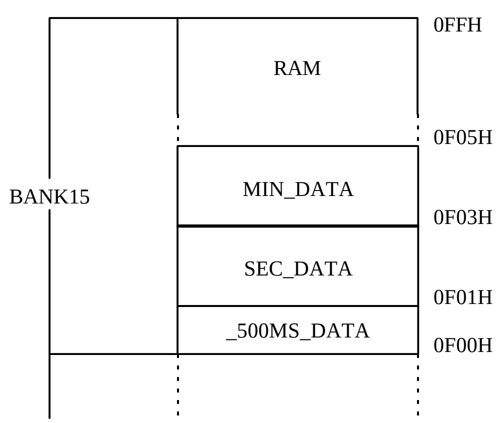


Figure 10-1   Data Storage Area

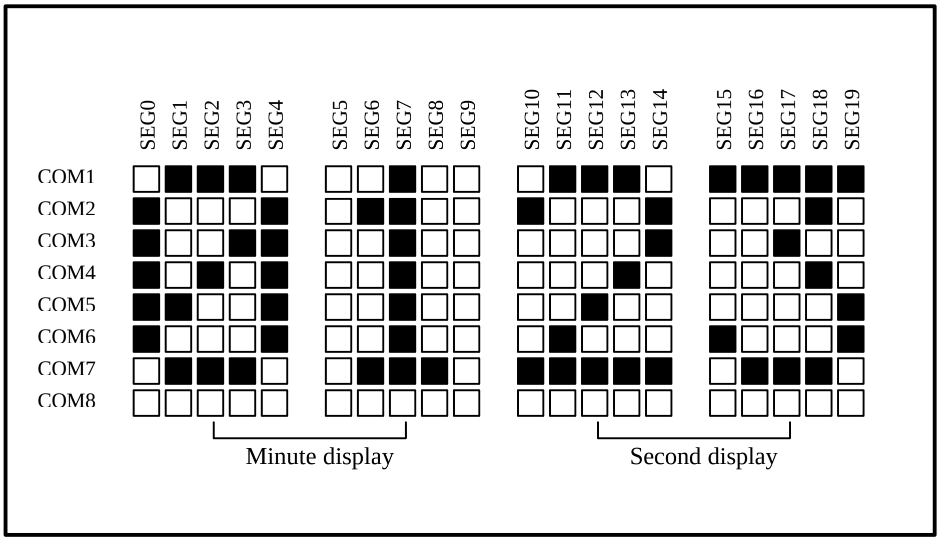


Figure 10-2 LCD Driver Outputs for Time Display

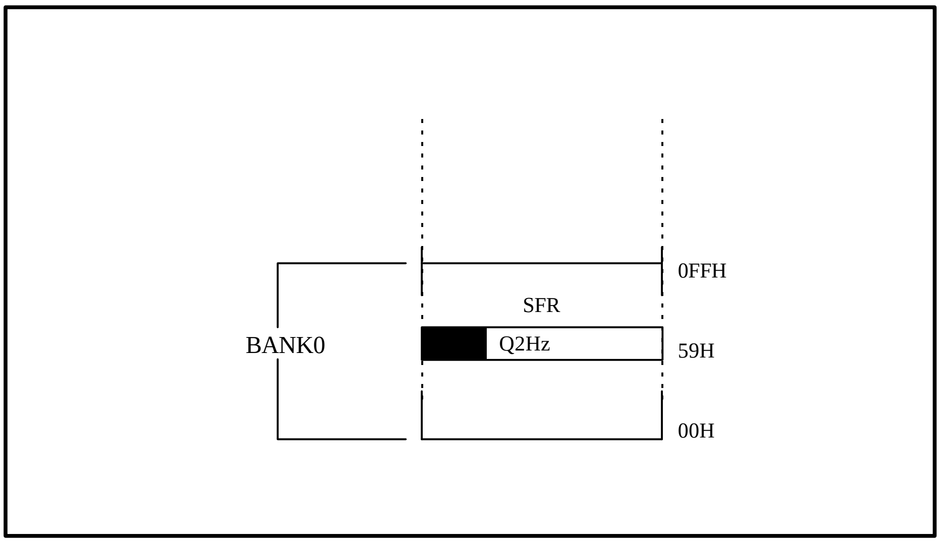


Figure 10-3 2-Hz Interrupt Request Flag

10.2 File Organization

The sample program is contained in the following six files.

1. MAIN.ASM
2. SFRSBL.DEF
3. DATSBL.DEF
4. MACRO.DEF
5. SUB.DEF
6. TABLE.DEF

The last five files are included in the MAIN.ASM file.

Each file is described below.

(1) MAIN.ASM

This contains the core processing routines. It specifies the target device with a TYPE directive, includes the subordinate files, and provides the main procedure.

Lines 14-41 use a WHILE statement to construct an infinite loop that increments and displays the timer. Lines 42-47 are the display routine. The main routine is coded using structured programming, an important SASM63K feature.

(2) SFRSBL.DEF

This file defines SFR symbols.

(3) DATSBL.DEF

This file defines data symbols.

(4) MACRO.DEF

This file defines a macro. The user may find it convenient to collect macros in a library. Note how the ability to make labels local eliminates the worry about defining the same label name twice.

(5) SUB.DEF

This file contains subroutines. These subroutines are all defined using the SUB directive, so, if this file is included at the end of MAIN.ASM, only the subroutines actually referenced will be expanded.

The user may find it convenient to collect subroutines in a library. Note how the ability to make labels local eliminates the worry about defining the same label name twice.

(6) TABLE.DEF

This file defines table data.

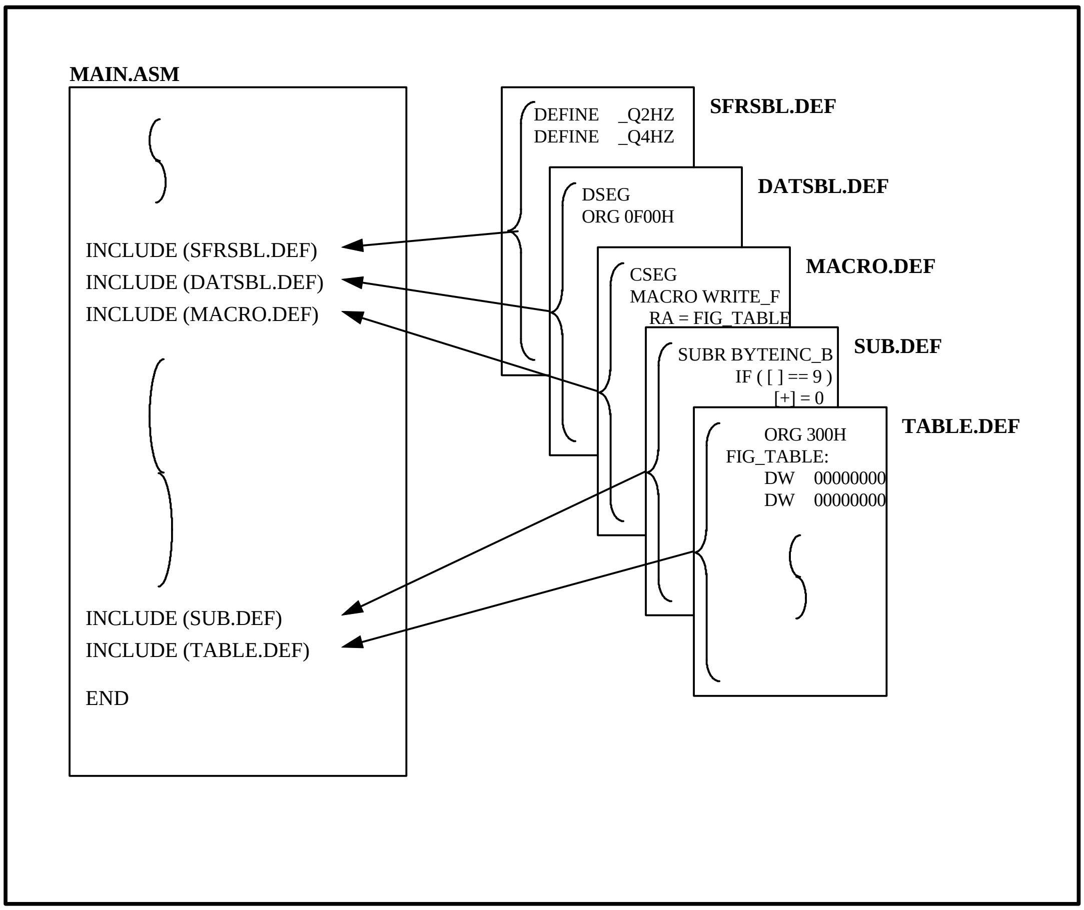


Figure 10-4 Program File Organization

■ MAIN.ASM (1)

```
;************************************************************
;******   SASM63K Sample Program                       ******
;******        for MSM63188                            ******
;******                                                ******
;******   Copyright 1995 OKI ELECTRIC INDUSTRY Co.,LTD.  ******
;************************************************************
TYPE (M63188)
TITLE "SASM63K Sample Program"
INCLUDE(SFRSBL.DEF)
INCLUDE(DATSBL.DEF)
INCLUDE(MACRO.DEF)
        CSEG
        ORG    0H
MAIN:
        [DSPCNT] = 8
        [DSPCON0] = 1
        [CBR] = 15
        [\_500MS_DATA] = 0
        [\SEC_DATA]    = 0
        [\SEC_DATA+1]  = 0
        [\MIN_DATA]    = 0
        [\MIN_DATA+1]  = 0
        CAL    DSP_LCD
        WHILE(TRUE)
                IF(_Q2HZ)
                        _Q2HZ = FALSE
                        HL = _500MS_DATA & 0FFH
                        [\_500MS_DATA] ++
                        IF ([\_500MS_DATA] == 2)
                                [\_500MS_DATA] = 0H
                                HL = SEC_DATA  & 0FFH
                                CAL    BYTEINC_BCD
                                IF ([\SEC_DATA+1] == 6)
                                        [\SEC_DATA+1] = 0
                                        HL = MIN_DATA & 0FFH
                                        CAL    BYTEINC_BCD
                                ENDI
                                CAL    DSP_LCD
                        ENDI
                ENDI
        ENDW
```

■ MAIN.ASM (2)

```
DSP_LCD:
        WRITE_FIG(DSPR0,MIN_DATA+1)
        WRITE_FIG(DSPR20,MIN_DATA)
        WRITE_FIG(DSPR40,SEC_DATA+1)
        WRITE_FIG(DSPR60,SEC_DATA)
        RT

INCLUDE(SUB.DEF)
INCLUDE(TABLE.DEF)
END
```

■ SFRSBL.DEF

```
DEFINE _Q2HZ   [IRQ4].3
DEFINE _Q4HZ   [IRQ4].2
DEFINE _Q16HZ  [IRQ4].1
DEFINE _Q32HZ  [IRQ4].0
DEFINE _E2HZ   [IE4].3
DEFINE _E4HZ   [IE4].2
DEFINE _E16HZ  [IE4].1
DEFINE _E32HZ  [IE4].0
DEFINE DSPR0   100H
DEFINE DSPR20  114H
DEFINE DSPR40  128H
DEFINE DSPR60  13CH
```

■ DATSBL.DEF

```
        DSEG
        ORG    0F00H
_500MS_DATA:    DS     1
SEC_DATA:       DS     2
MIN_DATA:       DS     2
```

■ MACRO.DEF

```
CSEG

MACRO WRITE_FIG(_A,_B)
       RA = FIG_TABLE
       FOR [X] = 0,4
              [RA0] += [\ _B]
              C,[RA1] += 0
       ENDF
       [CBR] = 1
       HL = _A & 0FFH
       FOR [X] = 0,4
              MOVLB   [HL],[RA]
              HL++
              HL++
              HL++
              HL++
              RA++
       ENDF
       [CBR] = 15
ENDM
```

■ SUB.DEF

```
SUBR BYTEINC_BCD
       IF ( [] == 9 )
              [+] = 0
              IF( [] == 9 )
                     [] = 0
              ELSE
                     []++
              ENDI
       ELSE
              []++
       ENDI
       RT
ENDSUB

SUBR BYTEDEC_BCD
       IF ( [] == 0 )
              [+] = 9
              IF ( [] == 0 )
                     [] = 9
              ELSE
                     [] --
              ENDI
       ELSE
              [] --
       ENDI
       RT
ENDSUB
```

■ TABLE.DEF (1)

```
        ORG     300H
FIG_TABLE:
;0
        DW      00000000_00111110B
        DW      00000000_01010001B
        DW      00000000_01001001B
        DW      00000000_01000101B
        DW      00000000_00111110B
;1
        DW      00000000_00000000B
        DW      00000000_01000010B
        DW      00000000_01111111B
        DW      00000000_01000000B
        DW      00000000_00000000B
;2
        DW      00000000_01000010B
        DW      00000000_01100001B
        DW      00000000_01010001B
        DW      00000000_01001001B
        DW      00000000_01000110B
;3
        DW      00000000_00100001B
        DW      00000000_01000001B
        DW      00000000_01000101B
        DW      00000000_01001011B
        DW      00000000_00110001B
;4
        DW      00000000_00011000B
        DW      00000000_00010100B
        DW      00000000_00010010B
        DW      00000000_01111111B
        DW      00000000_00010000B
;5
        DW      00000000_00100111B
        DW      00000000_01000101B
        DW      00000000_01000101B
        DW      00000000_01000101B
        DW      00000000_00111001B
```

■ TABLE.DEF (2)

```
;6
        DW      00000000_00111100B
        DW      00000000_01001010B
        DW      00000000_01001001B
        DW      00000000_01001001B
        DW      00000000_00110000B
;7
        DW      00000000_00000001B
        DW      00000000_01110001B
        DW      00000000_00001001B
        DW      00000000_00000101B
        DW      00000000_00000011B
;8
        DW      00000000_00110110B
        DW      00000000_01001001B
        DW      00000000_01001001B
        DW      00000000_01001001B
        DW      00000000_00110110B
;9
        DW      00000000_00000110B
        DW      00000000_01001001B
        DW      00000000_01001001B
        DW      00000000_00101001B
        DW      00000000_00011110B
```

Appendices

- Reserved Words

Reserved Words

A.1 Basic Instructions

ADC	ADCD	ADCJ	ADD	AND	BC	BCLR
BEQ	BGE	BGT	BLE	BLT	BMOV	BNC
BNE	BNG	BNOT	BNZ	BSET	BTST	BZ
CAL	CMP	DEC	DI	EI	FCLR	FSET
HALT	INC	INCB	INCW	JMP	LCAL	LJMP
MCLR	MMOV	MNOT	MOV	MOVHB	MOVLB	MOVXB
MSA	MSET	MTST	NOP	OR	POP	PUSH
ROL	ROR	RT	RTI	RTNMI	SBC	SBCD
SBCJ	SJMP	SUB	XCH	XOR		

A.2 Directives

ANY	BIT	BSEG	BANK	CODE	CSEG	DATA	
DATE	DB	DBIT	DEBUG	DEFINE	DS	DSEG	
DW	END	ENDM	ENDSUB	EQU	ERR	INCLUDE	
LINE	LIST	LOCAL	MACRO	NODEBUG		NOERR	NOLIST
NOOBJ	NOPRN	NOREF	NOSYM	OBJ	ORG	PAGE	
PRN	REF	REFER	SET	SUBR	SYM	TITLE	
TYPE	USING	XDATA	XSEG				

A.3 Registers

A	C	E	FLAG	G	HL	PC
RA	XY	Z				

A.4 Operators

XBANK

A.5 Control Statements

BREAK CASE CONTINUE DEFAULT ELSE ELSEIF
ENDF ENDI ENDS ENDW FFOR FIF
FOR FREPEAT FSWITCH FWHILE IF LFOR
LIF LREPEAT LSWITCH LWHILE REPEAT SFOR
SIF SREPEAT SWHILE SSWITCH SWITCH UNTIL
WHILE

A.6 Data Objects

TRUE FALSE

A.7 SASM Instruction Options

D

A.8 Addresses

The reserved words used for addresses are defined in the DCL file.