OKI

MAC66K Assembler Package User's Manual

Program Development Support Software

Relocatable Assembler	RAS66K
Linker	RL66K
Librarian	LIB66K
Object Converter	OH66K

THIRD EDITION
ISSUE DATE:Nov. 1993

TABLE OF CONTENTS

Chapter 1

Introduction

The MAC66K Assembler Package is software for developing OLMS-66K Series assembly language programs.

This manual describes the following software in the MAC66K Assembler Package:

Relocatable Assembler	RAS66K
Linker	RL66K
Librarian	LIB66K
Object Converter	OH66K

This chapter explains various information needed to read the rest of the manual. Read this chapter first before moving on to other chapters.

1.1 About The MAC66K Assembler Package

Thank you for your purchase of the MAC66K Assembler Package. This package contains software needed to create assembly language programs for the one-chip microcontroller OLMS-66K Series.

● **Package Contents**

1 Floppy Disk
1 MAC66K Assembler Package User’s Manual (this manual)
1 Macroprocessor MP User’s Manual

The floppy disk contains the following software.

● **Software**

Relocatable Assembler RAS66K

RAS66K generates an object file from a source file coded in assembly language. The object file will contain object code corresponding to the source file, as well as information needed for linking and debugging. RAS66K also generates a print file and error file.

Linker RL66K

RL66K links one or more object modules and generates a single absolute object file. It also generates a map file, which shows public symbols and segment allocation.

Librarian LIB66K

LIB66K is software for creating and managing library files. A library file is a single grouping of multiple object files which are used by RL66K.

Object Converter OH66K

OH66K converts an absolute object file generated by RL66K or RAS66K into a HEX file.

Macroprocessor MP

MP is software that parses macros coded in a source file and expands them to corresponding text.

1.2 System Requirements

The MAC66K Assembler Package software requires the following environment to operate.

Operating system	:	MS-DOS (version 3.1 or higher)
Host computer	:	Personal computer running MS-DOS
Free memory	:	At least 180 Kbytes

In the explanations to follow, MS-DOS is referred to simply as DOS.

1.3 About This Manual

This manual describes the software in the MAC66K Assembler Package. However, for MP refer to the Macroprocessor MP User’s Manual.

This manual assumes that the reader is familiar with assembly language and DOS, and that he can create and edit assembly language source files. This manual is not an introductory text.

An overview of each chapter is given below.

Chapter 1. Introduction

Chapter 1 is this chapter.

Chapter 2. Installation And Usage

Chapter 2 explains how to install the MAC66K Assembler Package and provides an overview of program development.

Chapter 3. Basic Programming Knowledge

Chapter 3 discusses basic knowledge needed to develop programs with the MAC66K Assembler Package.

Chapter 4. RAS66K

Chapter 4 explains how to use the relocatable assembler RAS66K and describes RAS66K assembly language.

Chapter 5. RL66K

Chapter 5 explains how to use the linker RL66K.

Chapter 6. LIB66K

Chapter 6 explains how to use the librarian LIB66K.

Chapter 7. OH66K

Chapter 7 explains how to use the object converter OH66K.

Chapter 8. Absolute Print File Generation

Chapter 7 explains how to generate absolute print files.

Appendices

The appendices provide charts of directives, and reserved words.

1.4 Related Documents

In addition to this manual, the MAC66K Assembler Package provides the following separate documents. Refer to them as needed.

- Macroprocessor MP User's Manual

  Manual for the macroprocessor MP.

- MAC66K.DOC

  Text file containing recent information not included in this manual.

- DCL66K.DOC

  Text file containing an explanation of DCL files used by RAS66K.

MAC66K.DOC and DCL66K.DOC are text-format files on the provided floppy disk. They can be referred with the DOS TYPE command or with your own editor.

In addition to the above documents, the MAC66K Assembler Package also contains some related documents. These related documents may be a microcontroller hardware manual, an instruction manual, an emulator or simulator manual for use in debugging, etc. The MAC66K.DOC file describes which related documents are included, so please refer to it.

Points in this manual where you should refer to these related documents will be indicated by "Please refer to related documents."

1.5 CPU Core

OLMS-66K one-chip microcontrollers are constructed from common CPUs with different I/O peripherals and different memory capacities.

The common CPU is called the CPU core, or simply the core. Multiple CPU cores exist in the OLMS-66K Series. Refer to MAC66K.DOC for the most recent information about devices and their CPU cores.

● **Examples in this manual**

The explanations of this manual include numerous program examples. These examples use OLMS-66K Series microcontroller instructions. Note that these may not be usable with your target microcontroller. You can confirm which instructions are usable with which CPU core by checking Appendix B, “List Of Reserved Words.”

1.6 Symbol Usage In This Manual

To make explanations easier to understand, this manual makes use of several types of symbols. The symbols and their meanings are listed below.

Symbol	Explanation
`SAMPLE`	These characters indicate messages displayed to the screen, command line input examples, or examples of generated list files.
`CAPITALS`	Upper-case letters indicate that the characters should be input as shown.
`italics`	Italics indicate that the characters should not be input as shown, but instead that they should be substituted with needed information.
`[ ]`	The contents of brackets are input as needed. They may be omitted.
`...`	The contents immediately preceding the ellipses may be repeated as necessary.
`{choice1|choice2}`	The braces contain choices separated by vertical bars, one of which should be input. Unless the choices are enclosed by brackets [ ], one of them must be input.
value1 to value2	The value will be equal to or between *value 1* and *value 2*.
Ctrl+C	Press the “Ctrl” key and “C” key simultaneously.
`PROGRAM` `.` `.` `.` `PROGRAM`	A vertical line of dots indicates a partial omission of a program example.

When an “H” is appended to the end of a number in this manual, that value will be hexadecimal. For example, 1234H will indicate the hexadecimal 1234.

1.7 Changes From Previous MAC66K Assembler Package Ver. 2.XX

This manual explains the use of MAC66K Assembler Package Ver.4.XX. The MAC66K Assembler Package Ver. 4.XX is software that supports all devices in the OLMS-66K Series, or in other words, supports all CPU cores nX-8/100, 200, 300, 400, and 500. It is an upgrade from the MAC66K Assembler Package Ver. 2.XX, which supported the nX-8/200, 300, and 400 cores.

This section describes the changes and additions to the previous version (Ver. 2.XX) for its users. New users who are starting with the latest version (Ver. 4.XX) may skip this section.

The major changes and additions to the previous version (Ver. 2.XX) are as follows.

● **nX-8/100 and 500 core support**

Ver. 4.XX adds new support for the CPU cores nX-8/100 and 500. Assembly language specifications that woud not clearly support the complex architecture of the nX-8/500 core have been clarified. Therefore the check functions of RAS66K and RL66K have been stregthened.

● **Object file format changes**

The format of object files has changed. Accordingly, object files generated by the previous version of RAS66K cannot be linked with the new version of RL66K or converted to HEX files with the new version of OH66K. Creation and management of libraries using LIB66K is still the same.

The following sections explain the changes and additions to the previous versions of each software in the package (RAS66K, RL66K, OH66K, LIB66K). The old version of the software is expressed as “old *xxx*” and the new version as “new *xxx*.”

1.7.1 RAS66K

■ Starting RAS66K

The new RAS66K supports only the following format of starting command line.

> RAS66K *source_file* [*options*]

The following input will display RAS66K usage and terminate.

> RAS66K

■ Options

The format of old RAS66K options was the same as for directives, but new RAS66K options are specified as a slash (/) followed by a one or two-character specification. For example, to generate a cross-reference list and symbol list, input the following.

```
RAS66K foo /S /R
```

In addition, the following options have been added.

/W [*type*]	Perform warning checks of the specified type.
/NW [*type*]	Disable warning checks of the specified type.
/CD	Recognize upper-case and lower-case distinctions.
/NCD	Ignore upper-case and lower-case distinctions.
/CC	Read C source level debugging information file.
/I*include_path*	Specifies include file path.
/V[*buffer_size*]	When source file is read, save it in a memory buffer of *buffer_size* bytes. This speeds up assembly during floppy disk-based operation.
/X	Generate an EXTRN declaration file.

For details, refer to Section 4.3.2, “Option Specifications.”

■ DCL file search order and search locations

The old RAS66K searched for DCL files in the following order.

1. Current directory
2. Directory specified by PATH environment variable

However, the new RAS66K searches in the order below.

1. Current directory
2. Directory containing RAS66K.EXE
3. Directory specified by DCL environment variable

■ Print file format

Print file format has changed. Refer to Section 4.13, "Print Files," for the new RAS66K print file format.

■ EXTRN declaration files

The new RAS66K can generate an EXTRN declaration file if the /X option is specified. The EXTRN declaration file codes EXTRN declarations corresponding to the public symbols in the source program. The usage types of public symbols and external symbols must completely match with the new RL66K, so EXTRN declaration files could be used to prevent link errors from coding mistakes and type mismatches. For details, refer to Section 4.14, "EXTRN Declaration Files."

■ Upper-case and lower-case distinctions

Upper-case and lower-case distinctions of letters in symbols defined in the source program can be specified with the new RAS66K.

If /CD is specified, then upper-case and lower-case distinctions are recognized.

If /NCD is specified, then upper-case and lower-case distinctions are ignored.

If neither option is specified, then upper-case and lower-case distinctions are ignored.

■ Address constants

The new RAS66K allows coding of addresses that include physical segment addresses. For example, an address with physical segment address 3 and offset address 1000H would be coded as follows.

```
3:1000H
```

Refer to Section 3.2.1, "Overview Of Memory Space," regarding physical segment addresses. Refer to Section 4.8.2.2, "Address Constants," regarding address constants.

■ New operators

The new RAS66K adds the following operators.

SEG *expression*	Returns the physical segment address of an address expression
OFFSET *expression*	Returns the offset address of an address expression.
PAGE *expression*	Returns the page number of an address expression.
BPOS *expression*	Returns the bit offset (0-7) of a bit address expression.
SIZE *segment_symbol*	Returns the segment size.

For details, refer to Section 4.9.2.6, “Special Operators.”

■ Expression attributes

The new RAS66K strictly manages the inheritance of attributes in expressions. As a result, calculation error checking is considerably more severe than in the old RAS66K. Calculations that are contradictory will cause warnings. For details, refer to Section 4.9.1.1, “Meaning Of Attributes Of Expressions.”

■ Stack segment

The new RAS66K now allows a stack segment to be defined. The stack segment is a special relocatable segment for use as a stack area. It is defined as follows.

STACKSEG *stack_size*

For details, refer to Section 4.5.4, “Stack Segment.”

■ Segment groups

Segment groups have meaning for nX-8/300 and 500 cores, which have multiple physical segments.

The new RAS66K can allocate several relocatable segments to a single physical segment. This is called segment grouping. Segments in the same group can be accessed without switching physical segments (LRB bits 13-15 for the 300 core, and DSR for the 500 core).

Segment groups are defined as follows.

GROUP [#*physical_segment*] *segment_symbol* [*segment_symbol*]...

For details, refer to Section 4.12.9, “Segment Group Definition.”

■ DSR checks

DSR checks have meaning for nX-8/300 and 500 cores, which have multiple physical segments. The DSR check function checks whether or not the currently selected RAM physical segment (LRB bits 13-15 for the 300 core, and DSR for the 500 core) matches the physical segment addresses coded in operands.

DSR is the name of the register that indicates the RAM segment address for the nX-8/500 CPU. RAS66K assembly language uses the same term as for MSM663XX.

The current RAM physical segment is set by coding one of the following.

USING DSREG *address*

or

USING DSREG #*physical_segment*

For details, refer to Section 4.10.4, “Physical Segment Address Checks.”

■ SFR access attribute checks

The new RAS66K checks whether or not accesses to SFRs are correct. Incorrect accesses will cause warnings. SFR access attributes are defined in DCL files. For details, refer to Section 4.10.6, “Special Function Register Access Checks.”

■ Communal symbols

Communal symbols are symbols in a communal area of memory common to multiple modules. They are defined as follows.

symbol COMM *segment_type size* [*relocation_type*]

The *symbol* will represent the first address. For details, refer to Section 4.12.13, “Creating Programs From Multiple Source Files.”

■ Conditional assembly functions

The new RAS66K adds three directives for conditional assembly.

IF *expression*

IFDEF *symbol*

IFNDEF *symbol*

For details, refer to Section 4.7.1, “Using Conditional Assembly.”

■ Macro functions

A string can be assigned to a symbol by defining a macro with the DEFINE directive. By using macros, you can replace instructions with different names or reduce the amount of repetitive code you must write. Macros are defined as follows.

DEFINE *symbol* “*string*”

For details, refer to Section 4.7.2, “Using Macros.”

1.7.2 RL66K

■ Starting RL66K

The start command input has changed with the introduction of a semicolon (;), which means all further input is to be omitted. This also slightly changes how response files are written. For details, refer to Section 5.3, "Using RL66K."

■ Options

The following options have been added to the new RL66K.

/ORDER (*segment_name*)	Control allocation order of segments of the same precedence.
/CM (*address*)	Set maximum address of program memory space.
/DM (*address*)	Set maximum address of data memory space.
/STACK (*size*)	Change stack size.
/CC	Automatically search for standard libraries LONG.LIB and FLOAT.LIB.
/S	Outputs public symbol list to map file.
/NS	Do not output public symbol list to map file.
/A [*abl_file*]	Generate an ABL file.
/NA	Do not generate an ABL file.
/SD	Output CDB debugging information.
/NSD	Do not output CDB debugging information.

Also, physical segment addresses can be specified with /CODE, /DATA, /BIT, /EDATA, and /EBIT directives. However, this is only allowed when the physical segment attribute is ANY for the nX-8/300 and 500 cores, which have multiple physical segments.

For details, refer to Section 5.4, "RL66K Options."

■ Library file search

The new RL66K introduces the environment variable LIB66K for setting the directory that contains library files. When library file specifications do not include paths, the library files are searched for in the following order.

1. Current directory
2. Directory set by environment variable LIB66K

■ Map file format

The map file format has greatly changed. Refer to Section 5.6, “Map Files,” regarding RL66K map file format.

■ Allocation order of segments with same precedence

The old RL66K allocated segments with the same precedence in the order that modules were input, but the order is not defined by the new RL66K. To control allocation, use the /ORDER option.

■ Restriction on symbol usage

The old RL66K restricted the total number of symbols that it could handle, but the new RL66K is limited only by main memory. In this case, symbols are segment symbols, communal symbols, and public symbols.

■ Segment symbol matching

The old RL66K matches segment symbols and external symbols, but the new RL66K does not match segment symbols and other symbols.

■ Stack segment

The new assembler package introduces a new special relocatable segment called the stack segment. The stack segment is allocated to memory with special processing different than that of ordinary relocatable segments. For details, refer to Section 5.5.6, “Reserving The Stack Area.”

■ Segment grouping

Segment grouping has meaning for nX-8/300 and 500 cores, which have multiple physical segments.

The new RL66K can allocate multiple relocatable segments with the same group ID to a single physical segment. This is called a segment group. For details, refer to Section 5.5.5, “Segment Groups.”

■ Dynamic segments

The new assembler package introduces new special relocatable segments called dynamic segments. Dynamic segments is allocated to memory with special processing different than that of ordinary relocatable segments. For details, refer to Section 5.5.4.4, “Allocation of Areas With Special Attributes.”

1.7.3 LIB66K

■ Options

The new LIB66K adds the following options to restrict which modules can be registered when creating a new library file.

/*cpu_core*	Restrict registered modules to those of the core specified by *cpu_core*.
/*memory_model*	Restrict registered modules to those of the memory model specified by *memory_model*.

For details, refer to the description of the *options* field in Section 6.2.1, “Command Line Execution.”

■ Additional functions

In addition to module addition, deletion, replacement, and copying, the new LIB66K adds a new extraction function. Extraction saves a particular module from a library file into its own file, and then deletes the module from the library file. The operation symbol for executing extraction is the ampersand (&).

In addition, a function for linking library files has been added. For details, refer to Section 6.3.3, “Adding Library Files.”

1.7.4 OH66K

■ Options

The option to indicate output of debugging information was /S in the old OH66K, but that has been changed to /D in the new OH66K. In the new OH66K, the /S option specifies that the output file is to be in Motorola S format.

For details, refer to the *options* field in Section 7.2.1, “Command Line Conversion.”

Chapter 2

Installation And Usage

This chapter explains how to install the MAC66K Assembler Package and provides an overview of program development.

2.1 Introduction

This chapter explains how to install and use the MAC66K Assembler Package. It first describes how to install the MAC66K Assembler Package and to set the operating environment. It then explains module programming and the flow of program development using the MAC66K Assembler Package. Finally, it gives a basic introduction to the various software in the package.

You must understand DOS in order to install the MAC66K Assembler Package. If you are not familiar with DOS, then please read the DOS User’s Manual or an introductory text on DOS before beginning installation.

2.2 Disk Contents

Before beginning installation, verify the contents of the floppy disk provided in the MAC66K Assembler Package. This floppy disk contains the following files.

File Name	Description
RAS66K.EXE	Relocatable Assembler RAS66K executable file.
RL66K.EXE	Linker RL66K executable file.
LIB66K.EXE	Librarian LIB66K executable file.
OH66K.EXE	Object Converter OH66K executable file.
MP.EXE	Macroprocessor MP executable file.
M66XXX.DCL	DCL file. This is a text file used by assembling with RAS66K. The 66XXX actually indicates the OLMS-66K Series device name, such as 66301 or 66507.
MAC66K.DOC	Text file that includes a list of all files on the floppy disk and additions or changes made after this manual was written. If the contents of your floppy disk differs from the list of files, then please contact Oki Electric.
DCL66K.DOC	Text file that explains how to read DCL files.

The floppy disk may also include additional files. If so, they will be explained in the MAC66K.DOC file. The MAC66K.DOC file can be displayed using the DOS MORE command as shown below, or it can be referred with your own editor.

```
MORE < MAC66K.DOC
```

2.3 Installation

The MAC66K Assembler Package software is run from either floppy disk or hard disk. You use the software directly off your purchased floppy disk. However, a hard disk provides faster execution and larger capacity compared to a floppy disk, so you may be more pleased using a hard disk.

To install the MAC66K Assembler Package, perform the following steps.

Step 1. Make a back-up copy of the original floppy disk.

You can use the original floppy disk provided to perform the installation. However, if for some reason the floppy disk contents are damaged, installation may become impossible. Therefore, please make a back-up copy of this floppy disk before starting installation. This floppy disk is not copy protected. When you finish making the back-up copy, store the original disk in a safe place and use the back-up copy for the rest of the installation procedure.

Step 2. Decide where you will copy the files.

If copying to a hard disk, then you should create a new directory in which to copy the files. If copying to a floppy disk, then it does not matter whether or not you create a directory.

Step 3. Copy the files.

Copy all needed files to the directory decided upon in Step 2 with the DOS COPY command. Files that have extensions other than ".EXE" or ".DCL" are not necessary for software execution, so you can delete them from the directory after you have verified their contents.

Step 4. Set the DOS environment.

- Set the environment variable PATH

  If you will invoke the MAC66K Assembler Package from a directory other than the installation directory, then set the environment variable PATH to the installation path.

  Refer to the DOS User's Manual for an explanation of how to set the environment variable PATH.

- Update the CONFIG.SYS file

  Update the the FILES and BUFFERS commands coded in the DOS system configuration file CONFIG.SYS.

The FILES command sets the maximum number of files that can be open simultaneously when a program is running. If this number is less than the actual number of files that will be open simultaneously during program operation, then the program cannot operate correctly.

The BUFFERS command sets the number of areas to hold data temporarily. If this number is small, then program processing speed may be slowed.

The number of FILES and BUFFERS should be set to about 20 when using the MAC66K Assembler Package.

Refer to the DOS User’s Manual for details on CONFIG.SYS, FILES, and BUFFERS.

2.4 Environment Variables

Some environment variables are used in the MAC66K Assembler Package software. Table 2-1 shows these environment variables. These environment variables do not have to be set, but should be set as needed.

Table 2-1. Environment Variables Used By Software

Environment Variable	Description
DCL	The environment variable DCL is used when RAS66K searches for a DCL file. When the DCL file is not in the current directory or in the directory that contains RAS66K.EXE, RAS66K will use the environment variable DCL to search for it. For details, refer to Section 4.4.1, “DCL File Specification.”
LIB66K	The environment variable LIB66K is used when RL66K searches for a library file. When the library file is not in the current directory, RL66K will use the environment variable LIB66K to search fot it. For details, refer to Section 5.3.1.4, “*libraries* Field.”

2.5 Program Development Flow

This section describes the flow of developing assembly language programs using the MAC66K Assembler Package. Program debugging is not explained in this manual, so please refer to the manual of the debugger that you are using.

Figure 2-1 shows the program development flow. In this figure, squares indicate files. If a file has only one default extension, then that extension will be shown in the square. Ovals indicate software. Diamonds indicate decision points in the flow.

Read through the following description while referring to the corresponding numbers in the figure.

(1) Write the program using a commercial text editor. Call the file that contains the program code the source file (.ASM file).

(2) If the source file contains macros, then expand them using MP to generate an expanded source file (.Q file).

(3) Assemble the source file using RAS66K to generate an object file (.OBJ file). Also generate a print file (.PRN file). Error messages can also be output to a file.

(4) The object file (.OBJ file) can be registered in a library file (.LIB file) using LIB66K. Library files can be used as input to RL66K. A list file (.L66 file) can also be generated, which lists all object files and public symbols registered in a library file.

(5) Link all object files comprising one program using RL66K to generate one object file (.ABS file). RL66K resolves external references between object files and allocates logical segments to memory. It also generates a map file (.M66 file).

(6) Convert the object file (.ABS file) to a HEX file using OH66K. Refer to Chapter 7, “OH66K,” regarding types and formats of HEX files.

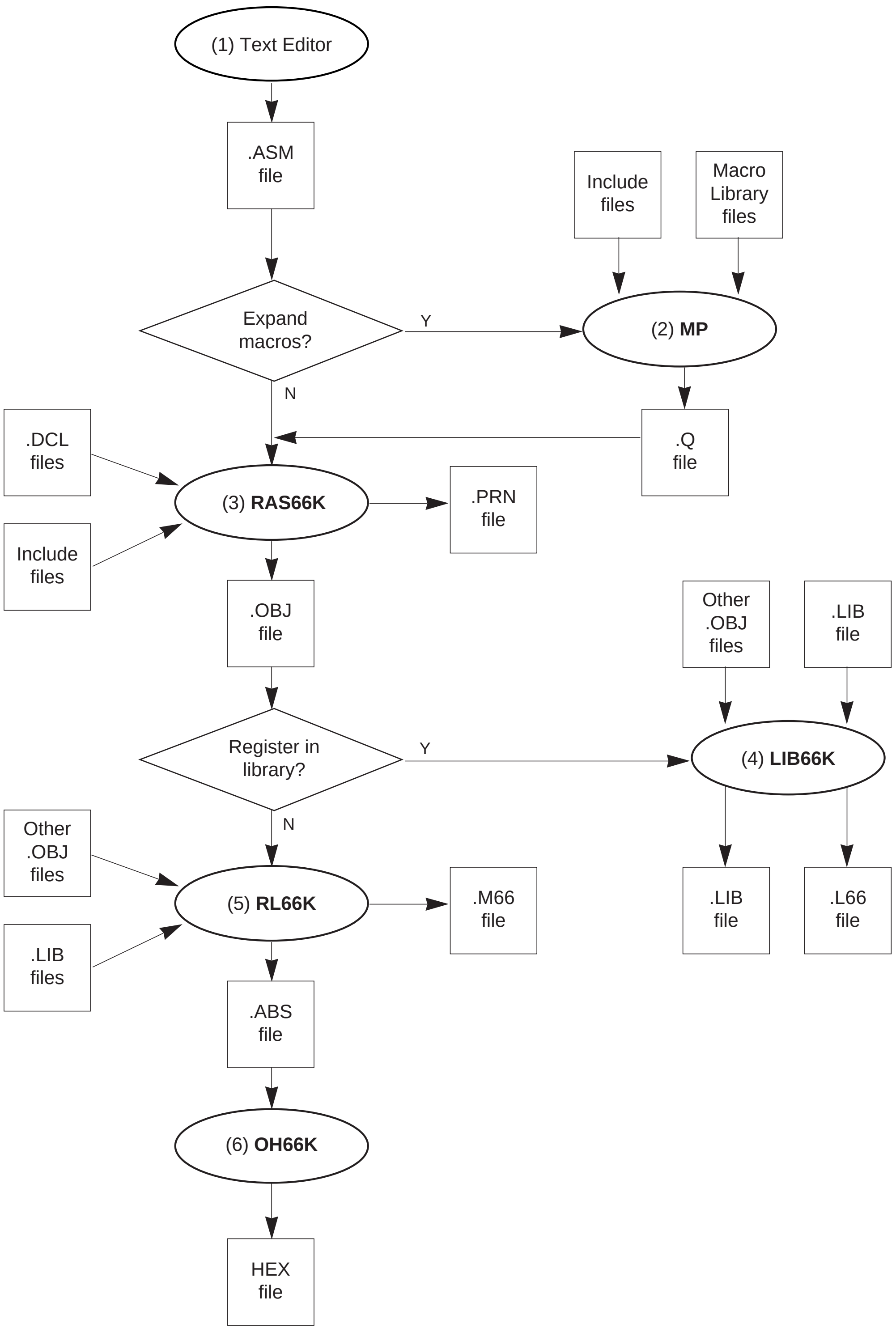


Figure 2-1. Program Development Flow

2.6 Module Programming

When developing a very large application program, one generally splits the program into several functional modules. The program is developed in modules. This section explains how to develop programs in modules using the MAC66K Assembler Package.

Consider developing a program divided into three functions. These three functions will each become a module. The development process is shown below. This process corresponds to the figure on the next page.

(1) Program the function of one module in one source file. This will result in the creation of three source files. (Assume the names of the three source files are MAIN.ASM, SUB.ASM, and STANDARD.ASM.)

(2) Assemble each source file with RAS66K. This will generate object files (MAIN.OBJ, SUB. OBJ, and STANDARD.OBJ).

(3) Register the STANDARD.OBJ file in a library file using LIB66K.

(4) Use RL66K to extract STANDARD module registered in the library file, link it with MAIN.OBJ and SUB.OBJ, and generate an .ABS file.

This is an example of programming by implementing a single source file from the various modules. The MAC66K Assembler Package handles an object file generated by one assembly as a single module.

In order to explain more concretely, focus on the module implemented as source file STANDARD.ASM in this example. RAS66K generates the object file STANDARD.OBJ from the module. LIB66K registers the object file STANDARD.OBJ in a library file. RL66K extracts STANDARD module from the library file.

The ability to register STANDARD.OBJ in a library file and then to extract STANDARD.OBJ from the library file and link it is critical to developing programs in modules. This is because the reason for developing programs in modules is not only to make programming easier, but also to allow modules to be reused. By programming a module as one source file, each object file created by assembly corresponds to one module. These modules can then be registered in library files and later extracted from the library files and linked.

Therefore the MAC66K Assembler Package handles an object file as a single module. This means that an object file alone or an object file registered in a library file is called an object module. Object modules are sometimes called modules for short.

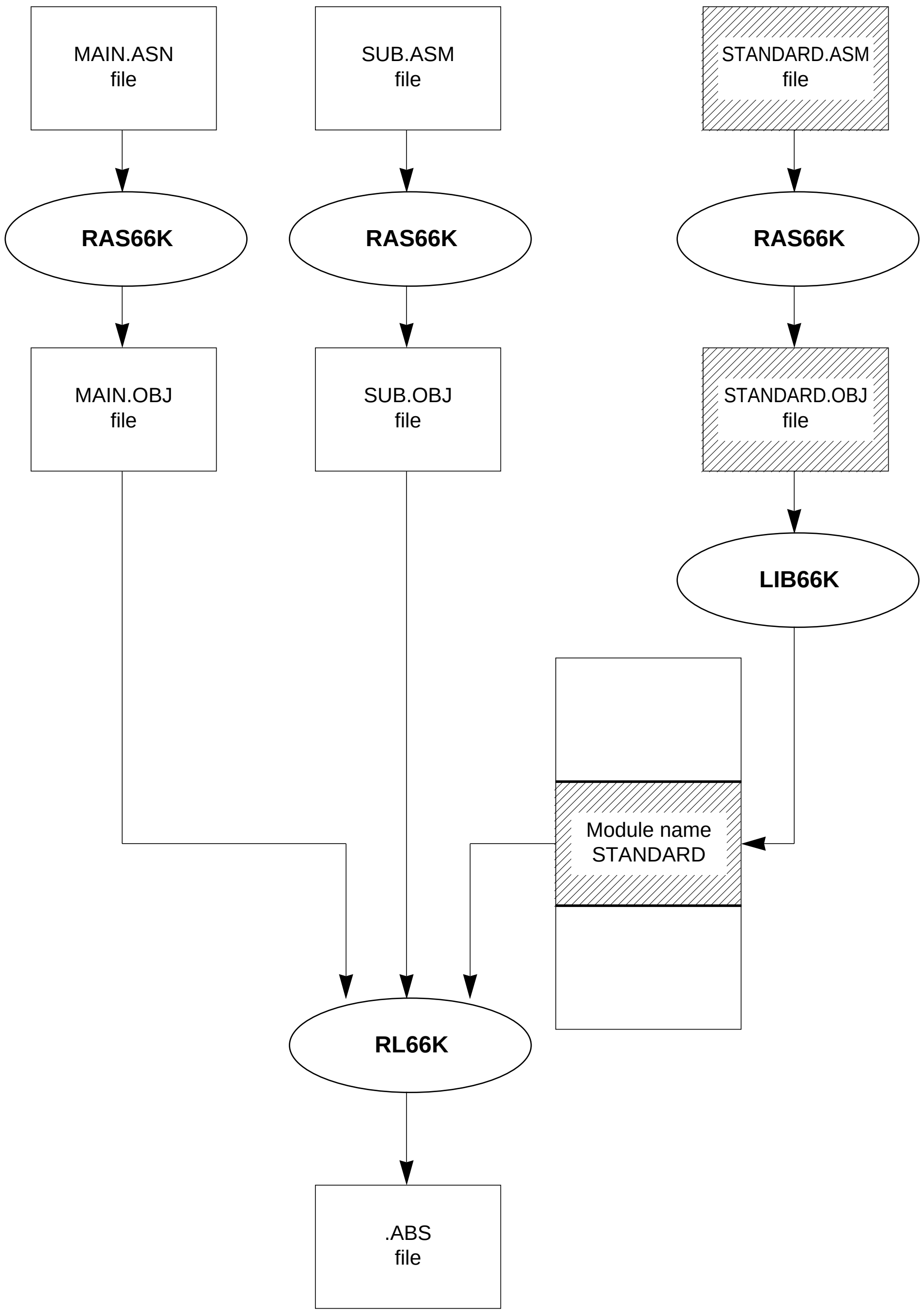


Figure 2-2. Module Programming

2.7 Using The MAC66K Assembler Package Software

This section provides a simple introduction to using the MAC66K Assembler Package software and shows concrete examples. Details of each software are explained in other chapters and manuals.

2.7.1 MP: Macro Expansion

For source files that include macros, all macros must be expanded using MP before assembling with RAS66K. The format of the command line to invoke MP is as follows.

```
MP source_file [options]
```

The *source_file* field specifies the source file for macro expansion. The *options* field can specify MP options.

For details on MP usage and options, refer to the Macroprocessor MP User’s Manual.

■ **Example** ■

```
MP MAIN GEN
MP DISPLAY GEN
```

In this example, first the macros of source file MAIN.ASM are expanded, generating the macro expansion file MAIN.Q. Then the macros of source file DISPLAY.ASM are expanded, generating the macro expansion file DISPLAY.Q. Both commands specify the GEN option, so the portions defined as macros will be output as comments in the expanded source file.

2.7.2 RAS66K: Assembler

Source files are assembled using RAS66K. The format of the command line to invoke RAS66K is as follows.

```
RAS66K source_file [options]
```

The *source_file* field specifies the source file for assembly. The *options* field can specify RAS66K options.

For details on RAS66K usage and options, refer to Chapter 4, “RAS66K.”

■ **Example** ■

```
RAS66K MAIN.Q
RAS66K DISPLAY.Q
```

In this example, the source files MAIN.Q and DISPLAY.Q are assembled, generating the object files MAIN.OBJ and DISPLAY.OBJ. Because the default source file name extension is “.ASM,” the extension “.Q” must be used explicitly.

2.7.3 LIB66K: Registering Object Modules In Library Files

Object files generated with RAS66K can be registered in library files using LIB66K. The format of the command line to invoke LIB66K is as follows.

```
LIB66K library_file [operations] [,[list_file]
[,[output_library]]] [;]
```

The *library_file* field specifies the input library file to work with. The *operations* field specifies the operation be performed with the library specified in the *library_file* field. The operations available are addition (+), deletion (-), replacement (%), copying (*), and extraction (&). The *list_file* field specifies the list file name. The *output_library* field specifies the output library file name.

For details on LIB66K usage and options, refer to Chapter 6, “LIB66K.”

■ Example ■

```
LIB66K MODULES +PROC1 +PROC2;
```

In this example, the two object files PROC1.OBJ and PROC2.OBJ generated by RAS66K will be added to the library file MODULES.LIB.

2.7.4 RL66K: Linker

RL66K links object files output by RAS66K, generating a single object file. The format of the command line to invoke RL66K is as follows.

```
RL66K object_files [,[absolute_file] [,[map_file] [,[libraries]]]] [;]
```

The *object_files* field specifies the names of object files and library files to be linked. The *absolute_file* field specifies the name of the object file to be generated. The *map_file* field specifies the name of the map file. The *libraries* field specifies the library files to use to resolve unresolved external references. Options can be specified anywhere on the command line.

For details on RL66K usage and options, refer to Chapter 5, “RL66K.”

■ Example ■

```
RL66K MAIN DISPLAY,SYSV,,MODULES
```

In this example, the object files MAIN.OBJ and DISPLAY.OBJ are linked, generating a single object file SYSV.ABS. MODULES.LIB is used to resolve unresolved references.

2.7.5 OH66K: Changing File Format

If an object file is to be written to ROM, perhaps by a PROM programmer, in binary format, then format conversion using OH66K is necessary. The format of the command line to invoke OH66K is as follows.

```
OH66K object_file [hex_file] [;]
```

The *object_file* field specifies the object file to be converted. The *hex_file* field specifies the HEX file in which to output the converted contents. Options can be specified anywhere on the command line. OH66K provides several conversion formats, which can be specified using options.

For details on OH66K usage and options, refer to Chapter 7, “OH66K.”

■ Example ■

```
OH66K SYSV;
```

In this example, the object file SYSV.ABS is converted to a HEX file, generating the HEX file SYSV.HEX.

2.7.6 Generating Assembly Level Debugging Information

Debugging information of assembly language programs is called assembly level debugging information. This debugging information is used when debugging programs with a symbolic debugger. The method for generating debugging information is as follows.

```
RAS66K HELLO /D
RAS66K WORLD /D
RL66K HELLO WORLD /D;
OH66K HELLO /D;
```

The first two commands assemble HELLO.ASM and WORLD.ASM, generating HELLO.OBJ and WORLD.OBJ. Because the /D option is specified, debugging information will be output to HELLO.OBJ and WORLD.OBJ. The third command links HELLO.OBJ and WORLD.OBJ, generating the object file HELLO.ABS. Because the /D option is specified, debugging information will be output to HELLO.ABS. The last command converts the format of HELLO.ABS, generating HELLO.HEX. Because the /D option is specified, debugging information will be output to HELLO.HEX.

In this example, debugging information was output to all modules comprising the program (the two modules HELLO and WORLD), but as shown in the following example, debugging information can be output to only specific modules.

```
RAS66K HELLO /D
RAS66K WORLD
RL66K HELLO WORLD /D;
OH66K HELLO /D;
```

In this example, when WORLD.ASM is assembled the /D option is not specified, so debugging information will not be output to WORLD.OBJ. Accordingly, RL66K and OH66K will only handle debugging information of the module HELLO.

Chapter 3

Basic Programming Knowledge

This chapter gives an overview of the OLMS-66K Series architecture, and explains how each piece of the MAC66K assembler package works with it. This chapter also provides other basic knowledge that may be needed when reading this manual.

3.1 Introduction

The MAC66K Assembler Package is software for OLMS-66K Series program development. In order to perform OLMS-66K Series program development using this software, you must understand the hardware of your target microcontroller.

This chapter gives an overview of the OLMS-66K Series architecture, and explains how each piece of the MAC66K assembler package works with it. This chapter also provides other basic knowledge that may be needed when reading this manual. For details on microcontroller functions, please refer to the related documents.

The contents of this chapter are as follows.

- Memory space

  This section discusses OLMS-66K Series memory configuration for each CPU core.

- Address space

  This section explains how the address spaces correspond to the memory spaces.

- Logical segments

  This section describes logical segments, which are necessary for creating programs.

- Series correspondence with DCL files

  This section explains the DCL files that contain fixed information for each microcontroller.

- File specifications

  This section explains file specifications used with this manual.

3.2 Memory Space

3.2.1 Overview Of Memory Space

OLMS-66K Series microcontrollers have two types of memory space.

- Program memory space
- Data memory space

Program memory space addresses are assigned by bytes. Data memory space addresses are assigned by bytes or bits. In other words, program memory space can only be accessed in byte units, while data memory space can be accessed in byte or bit units.

Each memory space is configured as several physical segments. The size of one physical segment is 64K bytes for both program memory space and data memory space. This manual calls the number of a physical segment the physical segment address, and the address within a physical segment the offset address.

Physical segments are distinguished through the use of physical segment addresses. Physical segment addresses are assigned to each physical segment, beginning with 0. To change physical segments, the user must manipulate the segment registers in his program. The segment registers select the current physical segment.

The number of physical segments logically allowed differs depending on the microcontroller's CPU core. Even some microcontrollers with CPU cores that support multiple physical segments may themselves have only one physical segment.

The explanations of this manual assume that program memory space and data memory space each have multiple physical segments. If the microcontroller you are using has only one physical segment, then you need not pay special attention to explanations regarding physical segments.

The physical segment address range of each memory space differs depending on the target microcontroller. Refer to Section 3.5, "Series Correspondence With DCL Files."

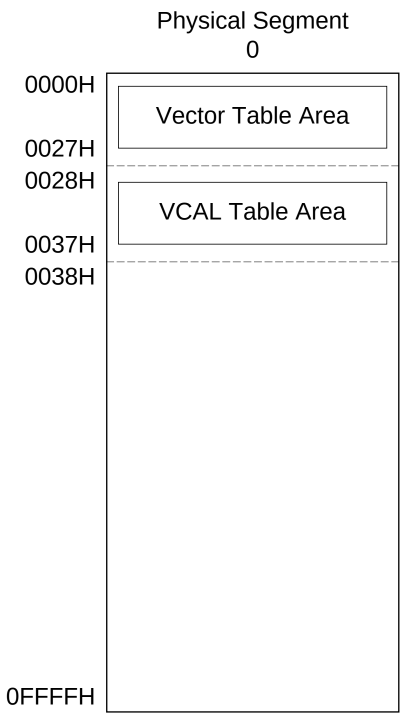
Program memory space and data memory space configurations are explained below for each CPU core type.

3.2.2 Memory Space Of nX-8/100, nX-8/200, nX-8/400

Microcontrollers with nX-8/100, nX-8/200, and nX-8/400 CPU cores can have one physical segment in program memory space and one physical segment in data memory space.

3.2.2.1 Program Memory Space

The figure below shows program memory space for nX-8/100, nX-8/200, and nX-8/400 CPU cores.



Program Memory Space Example

Each of the special areas is described next.

(1) Vector Table Area

The vector table area stores the addresses to be executed when the system is reset or an interrupt is generated. These addresses can be set using the DW directive.

■ Example ■

```
            TYPE    (M66201)

            CSEG    AT  0
            DW      PowerOnReset  ;Reset Pin
            DW      PowerOnReset  ;BRK
            DW      PowerOnReset  ;WDT
            .
            .
            .

MAIN        SEGMENT CODE
            RSEG    MAIN
PowerOnReset:
            .
            .
            .
```

In this example, addresses are set in the vector table area using the DW directive.

(2) VCAL Table Area

The VCAL table area stores addresses of subroutines to be executed by VCAL instructions. These addresses can be set using the DW directive.

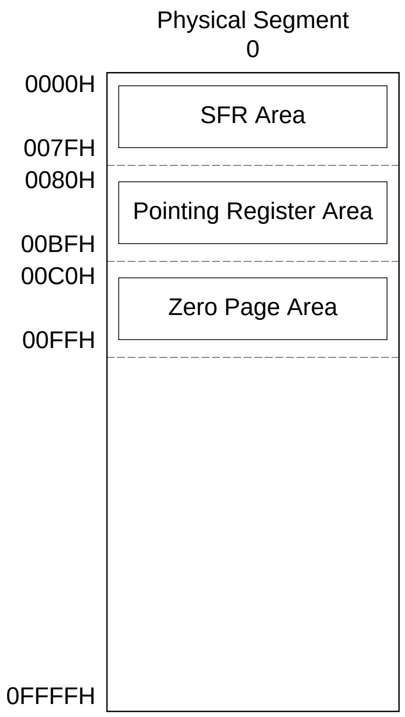
■ Example ■

```
            TYPE    (M66201)

            CSEG    AT 28H
VCAL_VCT00: DW      VCAL_FUNC00
VCAL_VCT01: DW      VCAL_FUNC01
VCAL_VCT02: DW      VCAL_FUNC02
            .
            .
            .

VCAL_ROUTINES       SEGMENT CODE
            RSEG    VCAL_ROUTINES
VCAL_FUNC00:
            .
            .
            .
            CSEG
            VCAL    VCAL_VCT00
            .
            .
            .
```

In this example, addresses are set in the VCAL table area using the DW directive.

3.2.2.2 Data Memory Space

The figure below shows data memory space for nX-8/100, nX-8/200, and nX-8/400 CPU cores. Addresses are shown as assigned by bytes.



Data Memory Space Example

Each of the special areas is described next.

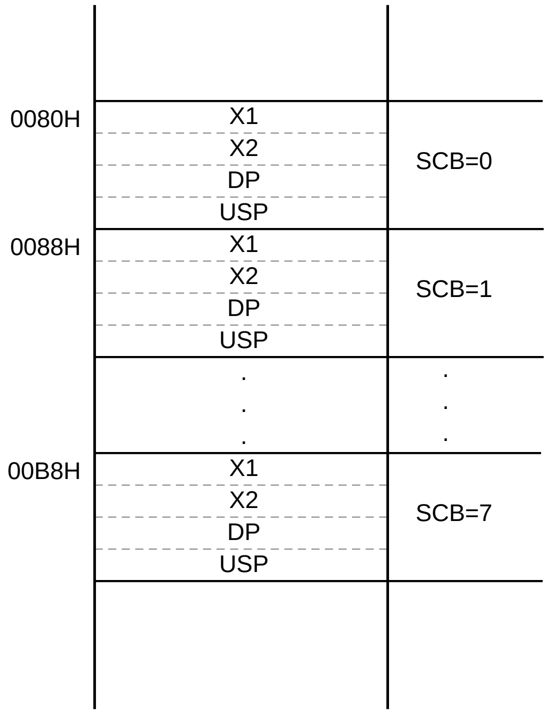
(1) Special Function Register (SFR) Area

The microcontroller's internal peripheral functions and the registers that control them are assigned to the special function register area. The addresses in this area are provided with reserved word names that correspond to the peripheral functions. These reserved words can be coded in programs instead of the address values. The reserved words differ depending on the target microcontroller. Refer to Section 3.5, "Series Correspondence With DCL Files."

(2) Pointing Register Area

The pointing register area is 64 bytes (8 bytes x 8 banks) to which the pointing register sets (X1, X2, DP, USP) are mapped. One of the 8 banks is selected by setting the SCB value in the PSW. To inform RAS66K of the SCB value, use the USING PREG directive.

The figure below shows a pointing register area and an example program that correspondingly sets the pointing register set.



Pointing Register Area

■ **Example** ■

```
TYPE    (M66201)

USING   PREG 3
ANDB    PSWL , #1111_1000B
ORB     PSWL , #0000_0011B
```

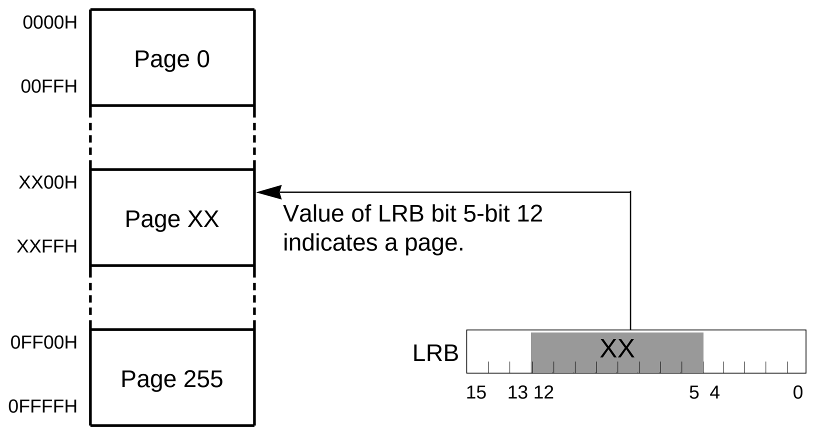
In this example, the program sets SCB of the PSW to 3 in order to select bank 3 of the pointing register set. It codes a USING PREG directive to inform RAS66K of the SCB value.

(3) Current Page Area

In data address space, the 256-byte spaces from base addresses on 256-byte boundaries are called pages. One physical segment in data address space is configured from 256 pages.

The current page is the page specified from bit 5 to bit 12 of the LRB. The current page area can be accessed using current page addressing. To inform RAS66K of the current page value, use the USING PAGE directive.

The figure below shows a current page area and an example program that correspondingly sets the current page.



Current Page Area

■ Example ■

```
          TYPE     (M66201)

          DSEG     AT   1000H
  BUF1_LR:DS       8
  BUF1:   DS       100H-8
          .
          .
          .
          CSEG
          USING    PAGE BUF1_LR
          MOV      LRB , #BUF1_LR>>3
```

In this example, in order to make the page containing the symbol BUF1_LR the current page, the program sets the page number in LRB. It codes a USING PAGE directive to inform RAS66K of the current page number.

(4) Zero Page Area

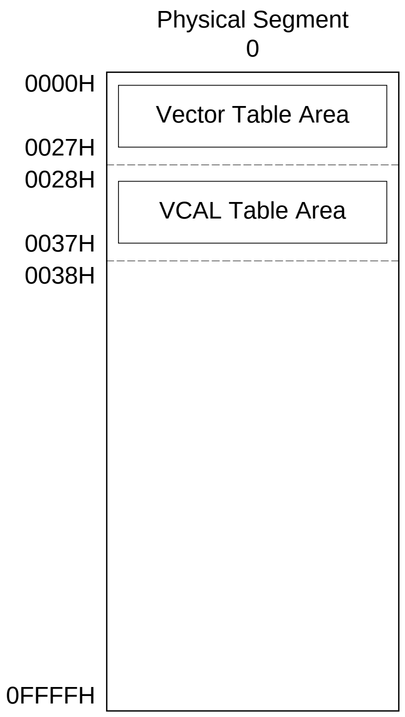
The zero page is the area of page 0, from 0000H to 00FFH. The zero page area can be accessed using zero page addressing.

3.2.3 Memory Space Of nX-8/300

Microcontrollers with nX-8/300 CPU cores can have one physical segment in program memory space and up to eight physical segments in data memory space.

3.2.3.1 Program Memory Space

The figure below shows program memory space for nX-8/300 CPU cores.



Program Memory Space Example

Each of the special areas is described next.

(1) Vector Table Area

The vector table area stores the addresses to be executed when the system is reset or an interrupt is generated. These addresses can be set using the DW directive.

■ Example ■

```
            TYPE    (M66301)

            CSEG    AT  0
            DW      PowerOnReset  ;Reset Pin
            DW      PowerOnReset  ;BRK
            DW      PowerOnReset  ;WDT
            .
            .
            .

MAIN        SEGMENT CODE
            RSEG    MAIN
PowerOnReset:
            .
            .
            .
```

In this example, addresses are set in the vector table area using the DW directive.

(2) VCAL Table Area

The VCAL table area stores addresses of subroutines to be executed by VCAL instructions. These addresses can be set using the DW directive.

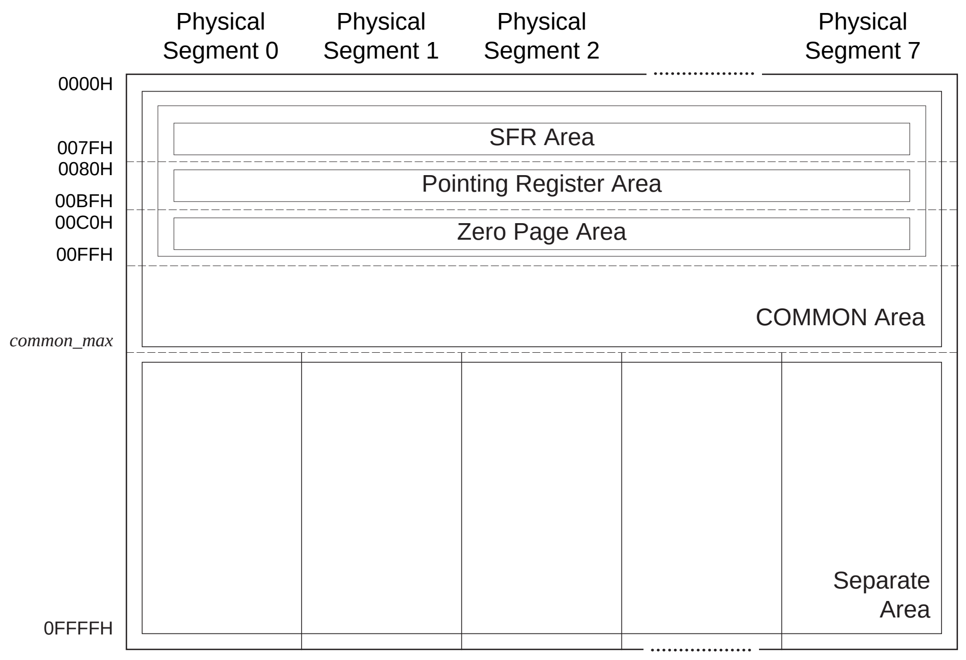
■ Example ■

```
            TYPE    (M66301)

            CSEG    AT 28H
VCAL_VCT00: DW      VCAL_FUNC00
VCAL_VCT01: DW      VCAL_FUNC01
VCAL_VCT02: DW      VCAL_FUNC02
            .
            .
            .

VCAL_ROUTINES       SEGMENT CODE
            RSEG    VCAL_ROUTINES
VCAL_FUNC00:
            .
            .
            .
            CSEG
            VCAL    VCAL_VCT00
            .
            .
            .
```

In this example, addresses are set in the VCAL table area using the DW directive.

3.2.3.2 Data Memory Space

The figure below shows data memory space for a physical segment address range 0-7. Addresses are allocated in byte units.



Data Memory Space Example

Data memory space can have up to eight physical segments. The physical segment is selected by setting its address in bit 13 to bit 15 of the local register base (LRB). To inform RAS66K of the physical segment address, use the USING DSREG directive.

Each of the special areas is described next.

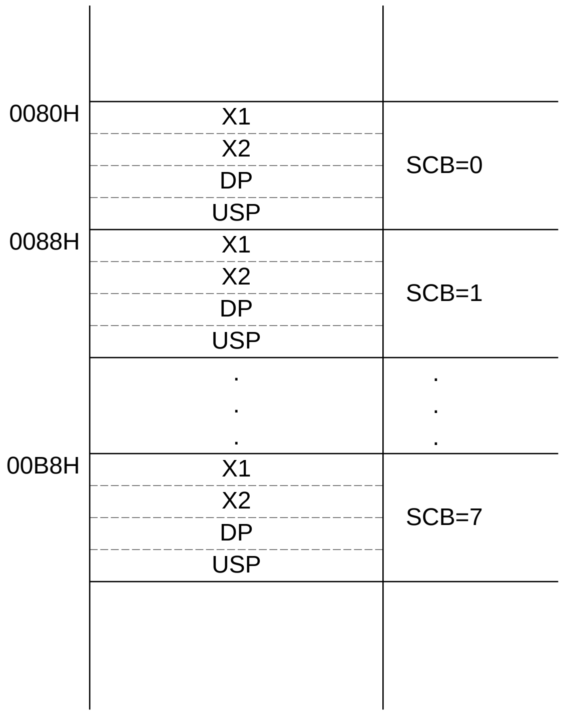
(1) Special Function Register (SFR) Area

The microcontroller’s internal peripheral functions and the registers that control them are assigned to the special function register area. The addresses in this area are provided with reserved word names that correspond to the peripheral functions. These reserved words can be coded in programs instead of the address values. The reserved words differ depending on the target microcontroller. Refer to Section 3.5, “Series Correspondence With DCL Files.”

(2) Pointing Register Area

The pointing register area is 64 bytes (8 bytes x 8 banks) to which the pointing register sets (X1, X2, DP, USP) are mapped. One of the 8 banks is selected by setting the SCB value in the PSW. To inform RAS66K of the SCB value, use the USING PREG directive.

The figure below shows a pointing register area and an example program that correspondingly sets the pointing register set.



Pointing Register Area

■ Example ■

```
TYPE     (M66301)

USING    PREG 3
ANDB     PSWL , #1111_1000B
ORB      PSWL , #0000_0011B
```

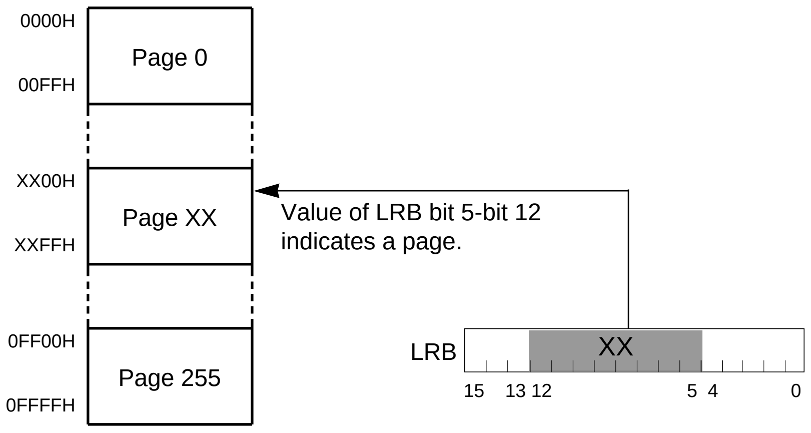
In this example, the program sets SCB of the PSW to 3 in order to select bank 3 of the pointing register set. It codes a USING PREG directive to inform RAS66K of the SCB value.

(3) Current Page Area

In data address space, the 256-byte spaces from base addresses on 256-byte boundaries are called pages. One physical segment in data address space is configured from 256 pages.

The current page is the page specified from bit 5 to bit 12 of the LRB. The current page area can be accessed using current page addressing. To inform RAS66K of the current page value, use the USING PAGE directive.

The figure below shows a current page area and an example program that correspondingly sets the current page.



Current Page Area

■ Example ■

```
        TYPE    (M66301)

        DSEG    AT   1000H
BUF1_LR:DS      8
BUF1:   DS      100H-8
        .
        .
        .
        CSEG
        USING   PAGE BUF1_LR
        MOV     LRB , #BUF1_LR>>3
```

In this example, in order to make the page containing the symbol BUF1_LR the current page, the program sets the page number in LRB. It codes a USING PAGE directive to inform RAS66K of the current page number.

(4) Zero Page Area

The zero page is the area of page 0, from 0000H to 00FFH. The zero page area can be accessed using zero page addressing.

(5) COMMON and Separate Areas

The COMMON area is an area common to all physical segments of data memory space. When the COMMON area is accessed, this common memory will be accessed regardless of the currently set physical segment. On the other hand, separate areas are independent memory spaces for each physical segment. When separate areas are accessed, even if the offset addresses accessed are the same, different memories will be accessed if the physical segment addresses are different.

The last address of the COMMON area, *common_max*, can be selected from one of four addresses. The *common_max* is selected by setting the value of BCB in the PSW to 0-3. To inform RAS66K of the value of BCB, use the COMMON directive.

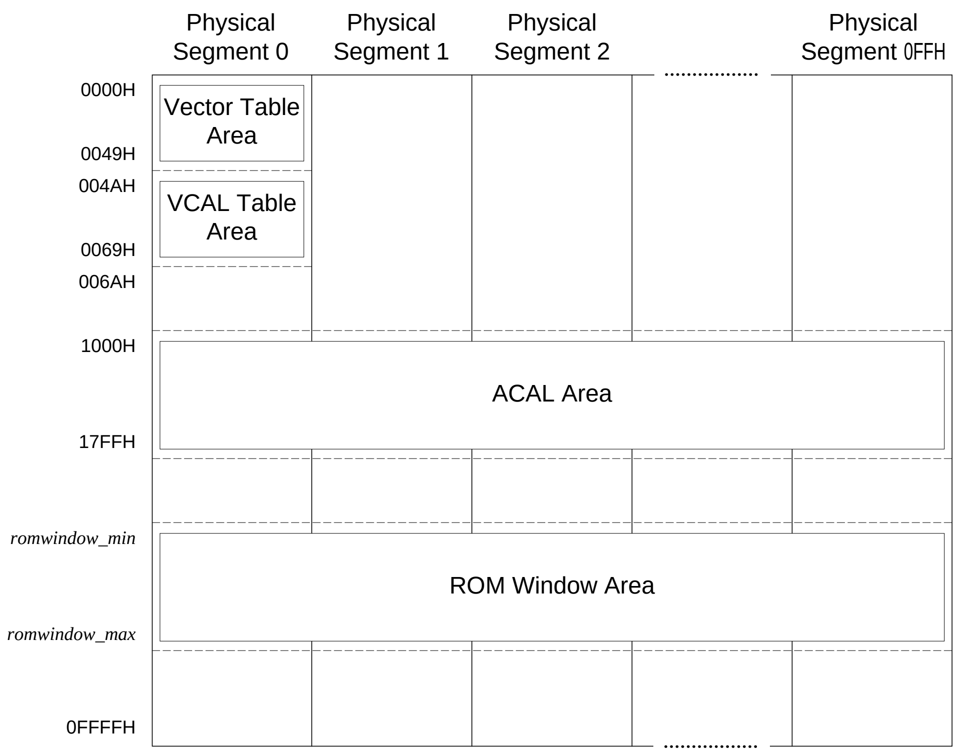
The four possible final addresses of common memory differ depending on the target microcontroller. Refer to Section 3.5, “Series Correspondence With DCL Files.”

3.2.4 Memory Space Of nX-8/500

Microcontrollers with nX-8/500 CPU cores can have up to 256 physical segments in program memory space and up to 256 physical segments in data memory space.

3.2.4.1 Program Memory Space

The figure below shows program memory space for nX-8/500 CPU cores.



Program Memory Space Example

Program memory space can have up to 256 physical segments. There are two segment registers that indicate physical segments in program memory space, the code segment register (CSR) and the table segment register (TSR).

When program memory space is accessed as code, CSR is the register that specifies the physical segment. If this register exists in the target microcontroller, then it will be allocated to the SFR area of data memory. The CSR is automatically rewritten by jumps between physical segments (FJ instruction) and calls between physical segments (FCAL instruction).

When program memory space is accessed as table data, TSR is the register that specifies the physical segment. If this register exists in the target microcontroller, then it will be allocated to the SFR area of data memory. To inform RAS66K of the value of TSR, use the USING TSREG directive.

Each of the special areas is described next.

(1) Vector Table Area

The vector table area stores the addresses to be executed when the system is reset or an interrupt is generated. These addresses can be set using the DW directive.

■ **Example** ■

```
            TYPE    (M66507)

            CSEG    AT  0
            DW      PowerOnReset  ;Reset Pin
            DW      PowerOnReset  ;BRK
            DW      PowerOnReset  ;WDT
            .
            .
            .

MAIN        SEGMENT CODE
            RSEG    MAIN
PowerOnReset:
            .
            .
            .
```

In this example, addresses are set in the vector table area using the DW directive.

(2) VCAL Table Area

The VCAL table area stores addresses of subroutines to be executed by VCAL instructions. These addresses can be set using the DW directive.

■ Example ■

```
            TYPE    (M66507)

            CSEG    AT 4AH
VCAL_VCT00: DW      VCAL_FUNC00
VCAL_VCT01: DW      VCAL_FUNC01
VCAL_VCT02: DW      VCAL_FUNC02
            .
            .
            .

VCAL_ROUTINES       SEGMENT CODE
            RSEG    VCAL_ROUTINES
VCAL_FUNC00:
            .
            .
            .
            CSEG
            VCAL    VCAL_VCT00
            .
            .
            .
```

In this example, addresses are set in the VCAL table area using the DW directive.

(3) ACAL Area

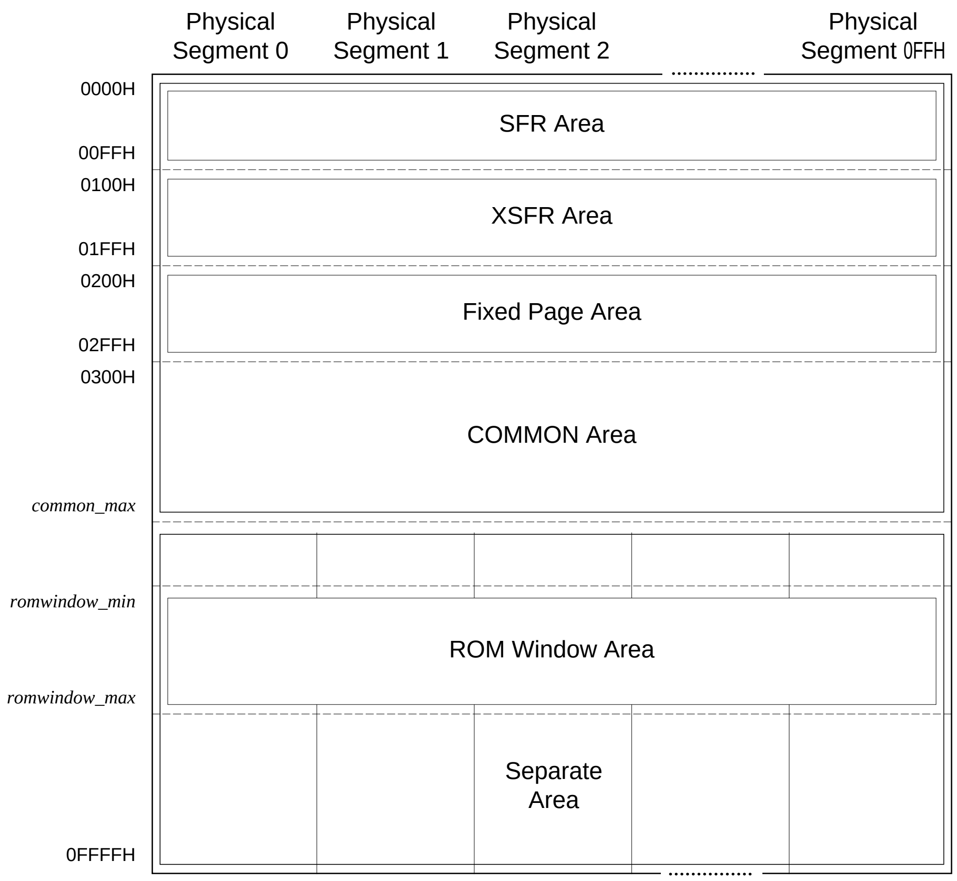
The ACAL area contains subroutines to be executed by ACAL instructions. Addresses of subroutines called by the ACAL instruction must be within the ACAL area.

(4) ROM Window Area

The ROM window area is an area of data memory space allocated to program memory space. Addressing data memory space in the ROM window will access program memory space. For details, refer to Section 3.2.4.2, “Data Memory Space.”

3.2.4.2 Data Memory Space

The figure below shows data memory space for a physical segment address range 0-0FFH. Addresses are allocated in byte units.



Data Memory Space Example

Data memory space can have up to 256 physical segments. The data segment register (DSR) indicates physical segment addresses in data memory space. To inform RAS66K of the physical segment address, use the USING DSREG directive.

Each of the special areas is described next.

(1) Special Function Register (SFR) Area and Extended Special Function Register (XSFR) Area

The microcontroller’s internal peripheral functions and the registers that control them are assigned to the SFR area and XSFR area. SFR page addressing can be used to access the SFR page, but it cannot be used to access the XSFR page.

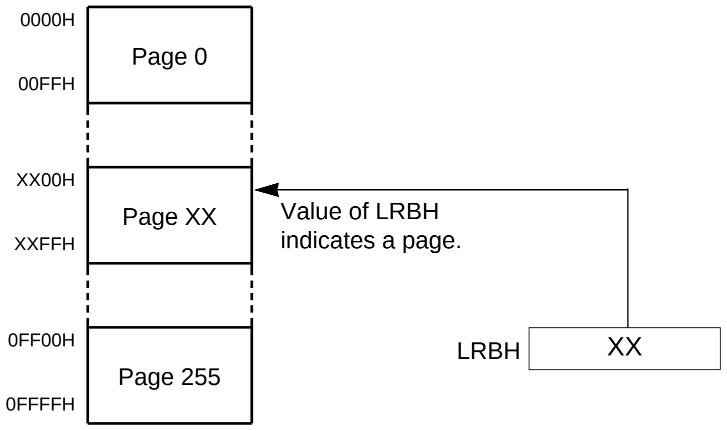
The addresses in these areas are provided with reserved word names that correspond to the peripheral functions. These reserved words can be coded in programs instead of the address values. The reserved words differ depending on the target microcontroller. Refer to Section 3.5, “Series Correspondence With DCL Files.”

(2) Current Page Area

In data address space, the 256-byte spaces from base addresses on 256-byte boundaries are called pages. One physical segment in data address space is configured from 256 pages.

The current page is the page specified by LRBH. The current page area can be accessed using current page addressing. To inform RAS66K of the current page value, use the USING PAGE directive.

The figure below shows a current page area and an example program that correspondingly sets the current page.



Current Page Area

■ Example ■

```
        TYPE    (M66507)

        DSEG    AT 1000H
BUF1:   DS      100H
        .
        .
        .
        CSEG
        USING   PAGE BUF1
        MOVB    ALRBH,#Page BUF1
```

In this example, in order to make the page containing the symbol BUF1 the current page, the program sets the page number in LRBH. It uses the “page” operator to determine the page number. It also codes a USING PAGE directive to inform RAS66K of the current page number.

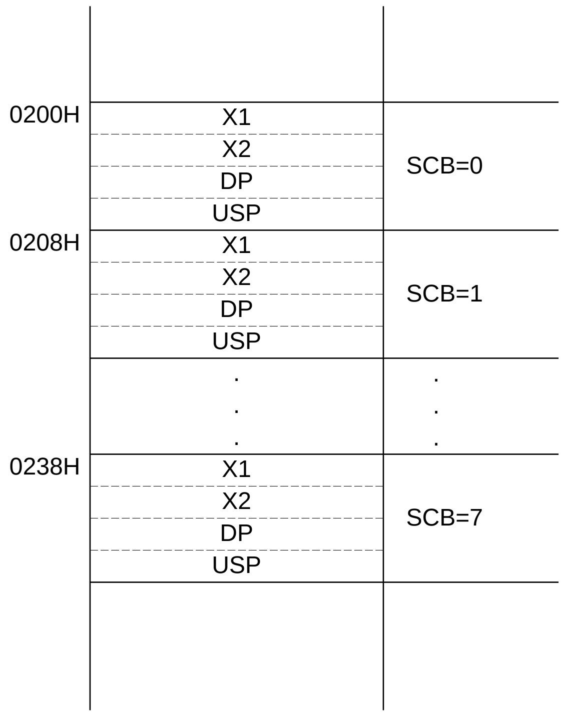
(3) Fixed Page Area

The fixed page is the area from 0200H to 02FFH. The fixed page area can be accessed using fixed page addressing.

(4) Pointing Register Area

The pointing register area is 64 bytes (8 bytes x 8 banks) after address 200H to which the pointing register sets (X1, X2, DP, USP) are mapped. One of the 8 banks is selected by setting the SCB value in the PSW. To inform RAS66K of the SCB value, use the USING PREG directive.

The figure below shows a pointing register area and an example program that correspondingly sets the pointing register set.



Pointing Register Area

■ **Example** ■

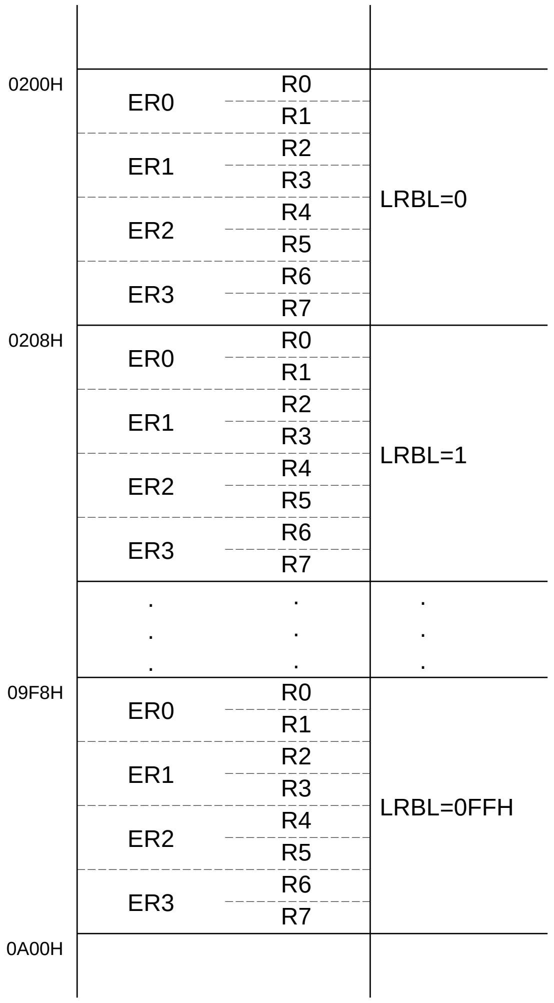
```
          TYPE    (M66507)

          USING   PREG 3
          ANDB    PSWL , #1111_1000B
          ORB     PSWL , #0000_0011B
```

In this example, the program sets SCB of the PSW to 3 in order to select bank 3 of the pointing register set. It codes a USING PREG directive to inform RAS66K of the SCB value.

(5) Local Register Area

The local register area is 2048 bytes (8 bytes x 256 banks) after address 200H to which the local register sets are mapped. One of the 256 banks is selected by setting the value of LRBL. To inform RAS66K of the LRBL value, use the USING LREG directive.

The figure below shows a local register area and an example program that correspondingly sets the local register set.



Local Register Area

■ **Example** ■

```
        TYPE    (M66507)

        USING   LREG 10H
        MOVB    ALRBL,#10H
```

In this example, the program sets LRBL to 10H in order to select bank 10H of the local register set. It codes a USING LREG directive to inform RAS66K of the LRBL value.

(6) EEPROM Area

The EEPROM area is the area in which the EEPROM is placed. It is the object of allocation by the EDATA segments and EBIT segments, which will be described later. EEPROM area memory can be initialized by DB or DW directives.

If the target microcontroller has an EEPROM, then the address range of the EEPROM area will be defined in the DCL file. Refer to Section 3.5, “Series Correspondence With DCL Files.”

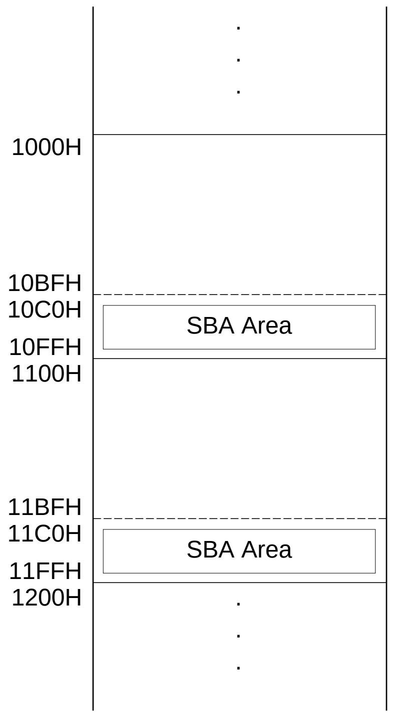
(7) Dual Port RAM Area

The dual port RAM area is the area in which the dual port RAM is placed.

If the target microcontroller has a dual port RAM, then the address range of the dual port RAM area will be defined in the DCL file. Refer to Section 3.5, “Series Correspondence With DCL Files.”

(8) SBA Area

The SBA area is area in each page of data memory space where the lower 8 bits of the address are 0C0H-0FFH. SBA areas can be accessed using sbaoff addressing. In addition, the SBA area of the fixed page area can be accessed using sbafix addressing.



SBA Area

(9) ROM Window Area

The ROM window area is an area of data memory space allocated to program memory space. Addressing data memory space in the ROM window will access program memory space. Therefore, even if external memory is placed in data memory space assigned to the ROM window area, that external memory cannot be accessed. However, the ROM window function is ineffective for internal RAM, the EEPROM area, and the dual port RAM area.

The lower 12 bits of the first address of the ROM window area are always 000H, and the lower 12 bits of the last address are always FFFH. The first address of the ROM window is always 1000H or above. To set the ROM window area, set the upper 4 bits of the first and last address of the ROM window area as the value of the ROMWIN register in the SFR area. To inform RAS66K of the first and last address of the ROM window area, use the WINDOW directive.

(10) COMMON and Separate Areas

The COMMON area is an area common to all physical segments of data memory space. When the COMMON area is accessed, this common memory will be accessed regardless of the currently set physical segment. On the other hand, separate areas are independent memory spaces for each physical segment. When separate areas are accessed, even if the offset addresses accessed are the same, different memories will be accessed if the physical segment addresses are different.

The last address of the COMMON area, *common_max*, can be selected from one of four addresses. The *common_max* is selected by setting the value of BCB in the PSW to 0-3. To inform RAS66K of the value of BCB, use the COMMON directive.

The four possible final addresses of common memory differ depending on the target microcontroller. Refer to Section 3.5, "Series Correspondence With DCL Files."

3.2.4.3 Memory Models

For microcontrollers with the nX-8/500 CPU core, the number of accessible physical segments in both program memory space and data memory space can be controlled. This means that four memory configurations exist for microcontrollers with the nX-8/500 CPU core. These are called memory models. The memory models and their meanings are given below.

Memory Models

Memory Model	Meaning
SMALL	Program memory space and data memory space are restricted to one physical segment each.
COMPACT	Program memory space is restricted to one physical segment.
MEDIUM	Data memory space is restricted to one physical segment.
LARGE	There are no restrictions on the number of physical segments of either program memory space or data memory space.

If the SMALL or COMPACT memory model is selected, then FJ, FCAL, and FRET instructions cannot be used.

To set the memory model, set the appropriate value in the register for memory model setting in the SFR area. To inform RAS66K of the type of memory model, use the MODEL directive. The memory model cannot be changed inside the program.

3.2.5 Memory Access

3.2.5.1 Wraparound

In the OLMS-66K Series, all calculations done with offset addresses are performed as 16 bits. For example, if 1 is added to offset address 0FFFFH, then the result will be offset address 0.

```
  0FFFFH
+     1H
--------
   0000H
```

As seen in this example, the carry from the most significant bits is not used. Thus, within a physical segment when the next byte is taken after the maximum address, it will come from the minimum address in that same physical segment. This is called wraparound. It is used in generating displacements for relative jump instructions.

■ **Example** ■

```
          CSEG
          ORG   10H
LABEL:
          .
          .
          .
          ORG   0FFEEH
          SJ    LABEL
```

The jump destination of the relative jump instruction placed at address 0FFEEH (SJ LABEL) is 10H, which is within the allowable range for relative jump. So, assembling this example will not cause an error.

3.2.5.2 Word Boundaries

When performing word-length (2-byte) accesses in OLMS-66K Series data memory, word boundaries exist. Word boundaries are boundaries between words. Word-length accesses are performed on these word boundaries. In other words, word-length accesses can only be to 2 bytes that start at an even address.

For example, if an attempt is made to specify a word access at address 1001H, then the least significant bit of the address will be dropped and the word from address 1000H will be accessed.

RAS66K and RL66K check for these word boundaries. Refer to Section 4.10.5, “Word Boundary Checks,” for information about these checks.

3.3 Address Space

For the MAC66K Assembler Package, consider that memory space is divided by its assigned addresses. These logical spaces are called address spaces.

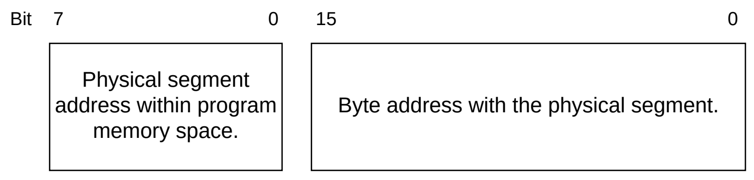
The table below shows the types of address spaces.

Address Spaces

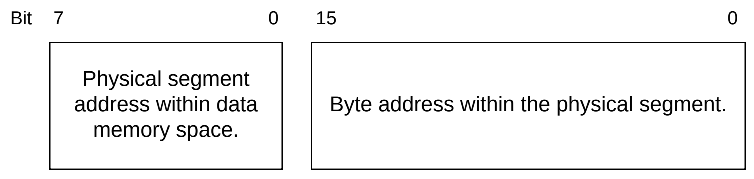
Address Space	Description
CODE address space	Program memory space where addresses are assigned by bytes.
DATA address space	Data memory space where addresses are assigned by bytes. (Excludes EEPROM area.)
BIT address space	Data memory space where addresses are assigned by bits. (Excludes EEPROM area.)
EDATA address space	EEPROM area where addresses are assigned by bytes.
EBIT address space	EEPROM area where addresses are assigned by bits.

The EDATA address space and EBIT address space exist only when EEPROM exists in the target device.

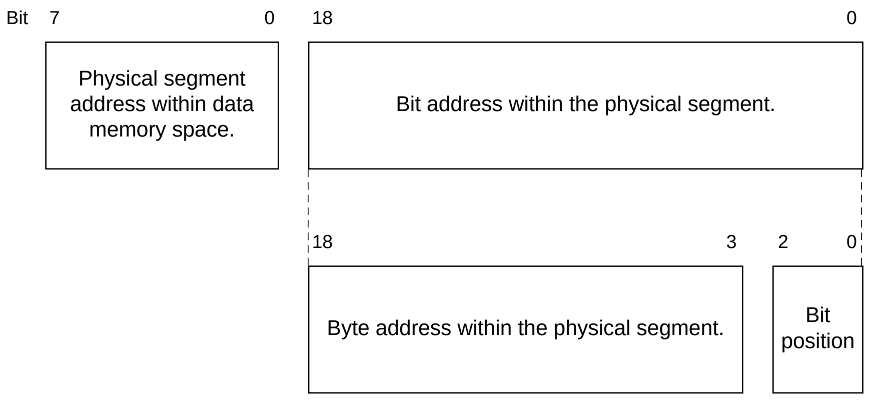
The physical segment addresses and offset addresses of the various address spaces are configured as follows.


Bit
7
0
15
0
Physical segment address within program memory space.
Byte address with the physical segment.

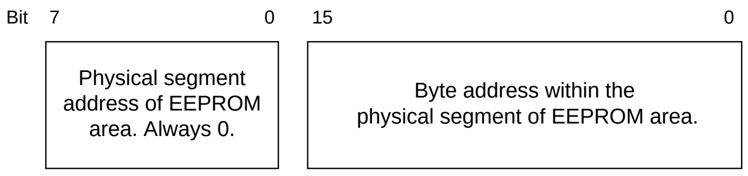
Addresses In CODE Address Space


Bit
7
0
15
0
Physical segment address within data memory space.
Byte address within the physical segment.

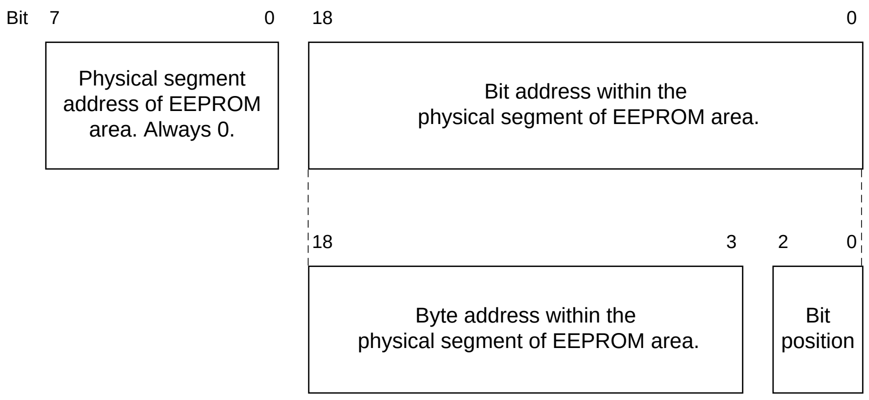
Addresses In DATA Address Space


Bit 7 0
Physical segment address within data memory space.
18 0
Bit address within the physical segment.
18 3 2 0
Byte address within the physical segment.
Bit position

Addresses In BIT Address Space


Bit 7 0
Physical segment address of EEPROM area. Always 0.
15 0
Byte address within the physical segment of EEPROM area.

Addresses In EDATA Address Space


Bit 7 0
Physical segment address of EEPROM area. Always 0.
18 0
Bit address within the physical segment of EEPROM area.
18 3 2 0
Byte address within the physical segment of EEPROM area.
Bit position

Addresses In EBIT Address Space

Physical segment addresses become part of object code of instructions only in the following cases.

■ Example ■

```
FCAL    CODELABEL1
MOVB    DSR,#SEG DATALABEL1
```

In this example, the physical segment addresses of CODELABEL1 and DATALABEL1 will become part of the instructions' object code.

The FCAL instruction is an instruction that can call any physical segment, so the physical segment address of CODELABEL1 will become the physical segment address of the jump location in object code.

The operand "#SEG DATALABEL1" of the MOVB instruction uses the SEG operator to extract the physical segment address. Accordingly, the physical segment address of DATALABEL1 will become the value written to DSR in object code.

However, look at the following example.

■ Example ■

```
CAL    CODELABEL2
MOVB   R0,DATALABEL2
```

In this example, the physical segment addresses of CODELABEL2 and DATALABEL2 will not become part of the instructions' object code.

The CAL instruction is an instruction that calls within the current physical segment, so the physical segment address of CODELABEL2 will not become the physical segment address of the jump location in object code.

The operand DATALABEL2 of the MOVB instruction specifies an address in the currently set physical segment address. Accordingly, the physical segment address of DATALABEL2 will not become part of the object code.

As demonstrated by the second example, the reason that each address includes the physical segment address even when the physical segment addresses will not become part of the instructions' object code is so RAS66K and RL66K can perform checks using the physical segment addresses. Refer to Section 4.10.4, "Physical Segment Address Checks."

3.4 Logical Segments

The OLMS-66K Series has five address spaces: CODE address space, DATA address space, BIT address space, EDATA address space, and EBIT address space. When writing a program in assembly language, the user needs to inform RAS66K and RL66K which parts of the program are in which address space. The concept of logical segments is used for this. A logical segment is an area of contiguous addresses.

For RAS66K, all code in a program belongs to these logical segments. RL66K fixes where the logical segments will be placed in memory space.

The table below shows the types of logical segments.

Logical Segments

Logical Segment	Description
CODE Segment	Logical segment corresponding to CODE address space.
DATA Segment	Logical segment corresponding to DATA address space.
BIT Segment	Logical segment corresponding to BIT address space.
EDATA Segment	Logical segment corresponding to EDATA address space.
EBIT Segment	Logical segment corresponding to EBIT address space.

RL66K fixes the allocation of logical segments in memory space. However, RL66K does not actually determine the allocation of all logical segments. Allocation of some logical segments can also be determined at RAS66K's level. These logical segments are defined by specifying absolute addresses for them. They are called absolute segments. Logical segments that do not have their allocation determined until RL66K's level are called relocatable segments.

RL66K checks that the addresses of absolute segments do not overlap addresses of other logical segments.

To write an absolute segment into a program, the address at which the segment is to be allocated must be specified in the program. Therefore, changing the address of an absolute segment requires the source program to be modified and reassembled.

Conversely, when developing a program using relocatable segments, one does not need to specify the allocation of the segments in the program. RL66K determines the addresses at which these logical segments will be allocated. Thus, the program does not need to specify the allocation in memory.

This feature of relocatable segments is very important. This is because the allocation of logical segments that make up a program can change if they are used multiple times in that same program.

Section 4.5, "Coding Logical Segments," explains the coding of logical segments.

■ Attention ■

When this manual uses the term “segment” it will be using the meaning of logical segment. However, when related documents use the term “segment” they may be using the meaning of physical segment, so be careful not to confuse them.

3.5 Series Correspondence With DCL Files

The MAC66K Assembler Package is software for the OLMS-66K Series. The software makes references to contents defined in a DCL file. Each DCL file contains particular information about a target microcontroller. By changing the DCL file, one can use the MAC66K Assembler Package with the various microcontrollers of the OLMS-66K Series.

DCL files are text format files. DCL file name extensions are always ".DCL." The name of the DCL file to be referred is specified within the source program. The software that refers the DCL file is RAS66K. RAS66K stores the DCL file information in the object file. RL66K and symbolic debuggers obtain the DCL file information through this object file.

3.5.1 Information In DCL Files

DCL files include the following information about microcontrollers.

(1) Core ID number

Based on this information, RAS66K, RL66K, and LIB66K distinguish between OLMS-66K Series CPU core architecture.

(2) Microcontroller ID number

Based on this information, RL66K determines the target microcontroller.

(3) Usable range of program memory space

Based on this information, RAS66K checks the values of operands accessed in program memory space. Based on this information, RL66K sets the last address of program memory space.

Information about program memory space is as follows.

- Offset address range
- Number of physical segments
- Range of special areas

(4) Usable range of data memory space

Based on this information, RAS66K checks the values of operands accessed in data address space and bit address space. It also checks whether accesses to the SFR area are in an accessible area. Based on this information, RL66K sets the last address of data memory space.

Information about data memory space is as follows.

- Offset address range
- Number of physical segments
- Range of COMMON area
- Range of special areas

(5) SFR area range and permitted range of access

Based on this information, RAS66K checks accesses to the SFR area. Refer to Section 4.10.6, “Special Function Register Access Checks.”

(6) Reserved words representing addresses

From this information, RAS66K obtains values and usage types of reserved words that represent addresses. RAS66K uses this to replace operands with addresses.

(7) Permitted instructions

RAS66K only assembles instructions that are permitted for use as defined in the DCL file. RAS66K will generate an error for other instructions.

3.5.2   About DCL66K.DOC

This manual is intended for all microcontrollers of the OLMS-66K Series. Particular functions about respective microcontrollers are described as follows.

“...is defined in the DCL file.”

The information included in DCL files is described in DCL66K.DOC.

3.6 File Specifications

Files are specified for input and output of the MAC66K Assembler Package software. This manual defines file specifications as follows.

drive: directory base_name.extension

The combination of drive and directory is called the path. The meanings for each part of file specifications are as shown below.

Term	Meaning
path	Specifies the drive and directory in which the target file exists.
drive	Specifies the drive on which the target file exists.
directory	Specifies the directory in which the target file exists.
base name	Specifies the target file name with a string up to 8 characters.
extension	Specifies the target file extension with a string up to 3 characters.

■ Example ■

```
A:\MAC66K\SRC\TEST.ASM
```

In this example, the elements of the file specification are as below.

Term	File Specification Element
path	A:\MAC66K\SRC\
drive	A
directory	\MAC66K\SRC\
base name	TEST
extension	ASM

Chapter 4

RAS66K

RAS66K is a relocatable assembler for the OLMS-66K Series of one-chip microcontrollers. This chapter describes information needed to use RAS66K and to create assembly language programs with RAS66K.

4.1 Introduction

RAS66K is a relocatable assembler for the OLMS-66K Series of one-chip microcontrollers. This chapter describes information needed to use RAS66K and to create assembly language programs with RAS66K.

RAS66K assembles a source file while referring to the contents of a DCL file. The DCL file contains information about the target microcomputer. By changing this file RAS66K can be used for any microcomputer of the OLMS-66K Series. A source file is a program written in OLMS-66K Series assembly language.

RAS66K generates four files as a result of assembly.

- Object file
- Print file
- Error file
- EXTRN declaration file

The object file includes relocatable object code and information needed for linking and debugging.

The print file includes the contents of the source file and the generated object code. Names and values of symbols used can also be output to the print file.

The error file includes error messages and the source statements that generated the errors. Unless otherwise specified, it will be output to the screen.

The EXTRN declaration file includes a list of EXTRN declarations corresponding to public symbols defined in the program.

Source files that follow the rules of OLMS-66K Series assembly language are basically structured from two types of statements.

- Microcontroller instruction statements
- Directive statements

Microcontroller instructions are discussed in related documents. By using directives, the programmer can control RAS66K operation or modify output. Directives are explained in this chapter.

4.2 File Specification Defaults

Input and output files must be specified to use RAS66K. Files are specified on the command line or as operands of directives. RAS66K file specifications are as follows.

- Source file specification
- Include file specification
- Object file specification
- Print file specification
- Error file specification
- EXTRN declaration file specification

Refer to Section 3.6, “File Specifications,” for the coding of each file specification.

The drive and directory can be omitted in the above file specifications. Except for source files and include files, the base name can also be omitted. Defaults when drives, directories, and base names are omitted are as shown below.

Table 4-1. File Specification Defaults

File Specification	Drive	Directory	Base Name	Extension
Source file	Current drive	Current directory of drive	Cannot be omitted	.ASM
Include file				Cannot be omitted
Object file	Current drive*	Current directory of drive*	Base name of source file	.OBJ
Print file				.PRN
Error file				.ERR
EXTRN declaration file				.EXT

NOTE: * If the drive, directory, base name and extension are all omitted, then the drive and directory will be that of the source file.

■ Example 1 ■

Below are some examples of source file specifications and the source files that RAS66K will read based on those specifications.

Source File Specification	Source File Read
TEXT.SRC	TEXT.SRC
TEXT.	TEXT
TEXT	TEXT.ASM

■ Example 2 ■

Below are some examples of how RAS66K handles the path and file name of the object file to be stored, assuming the source file specification is A:\SRC\TEXT.SRC.

Object File Specification	Path And File Name Of Generated Object File
TEXT.OUT	Generates the file TEXT.OUT in current directory of current drive.
A:TEXT.	Generates the file TEXT in the current directory of drive A.
B:\WORK\TEXT	Generates the file TEXT.OBJ in the \WORK\ directory of drive B.

4.3 Using RAS66K

4.3.1 Executing RAS66K

This section explains how to execute RAS66K.

At the DOS prompt, type RAS66K, specify the source file and options, and press the return key. Command line format is as follows.

```
RAS66K source_file [options]
```

The *source_file* specifies the source file to be assembled. The *options* can specify the various options. A slash (/) must be typed before the letters that indicate the option. Insert spaces between options.

If only RAS66K is typed without specifying *source_file*, and the return key is pressed, then RAS66K displays a guide to its use and a list of its options on the screen and returns to DOS.

■ Example ■

To assemble source file MAIN.ASM with option /S, type the following.

```
RAS66K MAIN.ASM /S
```

When the source file name extension is omitted, RAS66K adds the extension ".ASM" to process it. When the source file drive is omitted, RAS66K assumes the source file is on the current drive. When the source file directory is omitted, RAS66K assumes the source file is in the current directory.

If the command line is correctly input, then the RAS66K start message will be displayed on the screen. Next, the following messages will be displayed in order.

```
[dcl_file] loading...
 pass1...
 pass2...
```

RAS66K assembler processing first loads the DCL file. It displays "[*dcl_file*] loading" while the DCL file is being loaded. The *dcl_file* will actually be the name of DCL file loaded.

RAS66K divides its processing into two parts, called *pass 1* and *pass 2*. During pass 1, RAS66K determines symbol values and program addresses. During pass 2, RAS66K generates an object file using the results of pass 1. When pass 1 processing begins, "pass1..." will be displayed. When pass 2 processing begins, "pass2..." will be displayed.

If the generated program has errors, then error messages will be displayed. Refer to Section 4.15, "Error Messages."

When assembly ends, RAS66K displays the following messages and returns to DOS.

```
Print  File : foo.prn
Object File : foo.obj
Error  File : Console

Errors   : 0
Warnings : 0 (/Wrpeaus)
Lines    : 100
Assembly End.
```

The first three lines are names of generated files. The print file name follows “Print File,” the object file name follows “Object File,” and the error file name follows “Error File” (normally this will be “Console,” indicating screen display).

Following the file names, information about assembly results is displayed. The number of errors is shown after “Errors,” and the number of warnings is shown after “Warnings.” The warning types checked are shown following the “/W.” The number of lines in the source file is shown after “Lines.”

■ **Information** ■

RAS66K outputs all messages displayed on the screen to the standard output device. Messages can be output to a file using the DOS redirection function. To output only error messages and warning messages from a source program to a file, use the /E option or ERR directive.

4.3.2 Option Specifications

RAS66K operation and output file format can be controlled by specifying options. All options begin with a slash (/). The option name is specified following the slash. Spaces cannot be inserted between the slash and the option name. Both upper case and lower case letters can be used for option names. For many options, directives exist with identical functions.

4.3.2.1 List Of Options

Table 4-2 shows the options provided by RAS66K.

When neither an option or its corresponding directive is specified, RAS66K operation is shown in the “Default” column of Table 4-2. An asterisk (*) indicates that the function of that option is specified by default. Numbers indicate the default value set for that option’s function.

Table 4-2. Options With Functions Of Directives

Option	Default	Corresponding Directive	Function
/MS	*	MODEL SMALL	Set memory model to SMALL memory model.
/MC		MODEL COMPACT	Set memory model to COMPACT memory model.
/MM		MODEL MEDIUM	Set memory model to MEDIUM memory model.
/ML		MODEL LARGE	Set memory model to LARGE memory model.
/CF		CHK	Check flag attributes of branch instructions.
/NCF	*		Do not check flag attributes of branch instructions.
/CD			Recognize upper-case and lower-case distinctions.
/NCD	*		Ignore upper-case and lower-case distinctions.
/W **[***warning_type***]**	*		Specify type of warnings to check.
/NW **[***warning_type***]**			Specify type of warnings not to check.
/I*include_path*			Specify include file path.
/CC			Read C source level debugging information file.
/V **[***buffer_size***]**			Reserves a buffer for source file reading.
/PR **[***print_file*]	*	PRN	Generate print file.
/NPR		NOPRN	Do not generate print file.
/A **[***abl_file***]**			Generate absolute print file.
/L	*	LIST	Generate assembly list.
/NL		NOLIST	Do not generate assembly list.
/S		SYM	Generate symbol list.
/NS	*	NOSYM	Do not generate symbol list.
/R		REF	Generate cross-reference list.
/NR	*	NOREF	Do not generate cross-reference list.
/PW*page_width*	79	PAGE	Specify characters per line in print file.
/PL*page_length*	60	PAGE	Specify lines per page in print file.
/T **[***tab_code***]**	8	TAB	Replace tab codes.

Table 4-2. Options With Functions Of Directives (continued)

Option	Default	Corresponding Directive	Function
/O **[***objectfile***]**	*	OBJ	Generate object file.
/NO		NOOBJ	Do not generate object file.
/D		DEBUG	Output debugging information.
/ND	*	NODEBUG	Do not output debugging information.
/E **[***errorfile***]**	*	ERR	Set output destination of error messages.
/NE		NOERR	Output error messages to the screen.
/X **[***extern_file***]**			Generate EXTRN declaration fie.

4.3.2.2 Option Functions

(1) Memory Model Specification (/MS, /MC, /MM, /ML)

■ Syntax ■

```
/MS
/MC
/MM
/ML
```

■ Description ■

These options set the memory model used by the application program.

The /MS option sets the SMALL memory model; the /MC option sets the COMPACT memory model; the /MM option sets the MEDIUM memory model; and the /ML option sets the LARGE memory model. If no memory model is set, the SMALL memory model will be selected.

Refer to Section 3.2.4.3, “Memory Models.”

■ Corresponding Directives ■

The MODEL directive can be used to set the memory model instead of specifying these options. If both an option and the MODEL directive set the memory model, then the option setting will take precedence.

Refer to Section 4.12.2, “Memory Model Specification (MODEL).”

■ Example ■

To assemble the source file FOO.ASM with the COMPACT memory model, type the following.

```
RAS66K FOO.ASM /MC
```

■ Attention ■

Memory models are particular to microcontrollers with the nX-8/500 CPU core. When using microcontrollers with other CPU cores, RAS66K will ignore any specifications of memory model options.

(2) Control of Branch Instruction Flag Attribute Checks (/CF, /NCF)

■ Syntax ■

```
/CF
/NCF
```

■ Description ■

When the /CF option is specified, RAS66K will check whether or not the flag attributes of branch instruction sources and destinations match.

When the /NCF option is specified, RAS66K will not perform branch instruction flag attribute checks.

If neither option is specified, then RAS66K will not perform branch instruction flag attribute checks.

Refer to Section 4.10.9, "Flag Attribute Checks," for details on flag attributes and flag attribute checks.

■ Corresponding Directives ■

The CHK directive can be used in a program instead of specifying the /CF option. There is no directive with functions equivalent to the /NCF option.

Refer to Section 4.12.14.6, "Branch Instruction Flag Attribute Checks (CHK)."

■ Example ■

To assemble the source file FOO.ASM with branch instruction flag attribute checks, type the following.

```
RAS66K FOO.ASM /CF
```

(3) Suppression Of Upper And Lower Case Distinction (/CD, /NCD)

■ Syntax ■

```
/CD
/NCD
```

■ Description ■

When the /CD option is specified, RAS66K recognizes distinctions between upper case and lower case letters used in symbols in the source file. Even when symbols have the same spelling, if just one character differs in case, then the symbols will be different.

When the /NCD option is specified, RAS66K ignores distinctions between upper and lower case letters used in symbols. When symbols have the same spelling, they will be the same symbol even if the characters differ in case. If the /NCD option is specified, then RAS66K will convert all characters used in symbols to upper case. These converted names will be stored as symbol information in the print file and object file.

RAS66K will ignore case distinctions by default.

Case distinction can be controlled only for user symbols defined in the program, such as labels and segment names, and SFR symbols defined in the DCL file. Case distinction will be ignored in reserved words, such as instructions and directives, regardless of the /CD option specification.

■ Example ■

To assemble the source file FOO.ASM with upper case and lower case distinction, type the following.

```
RAS66K FOO.ASM /CD
```

If the source file FOO.ASM includes the following code, then no double-definition error will occur.

```
    UCSYM   EQU     200H
    ucsym   DATA    200H

    DSEG AT 200H
uCsYm:
    DS      10H
```

The spelling of symbols UCSYM, ucsym, and uCsYm have the same spelling, but they have different combinations of upper and lower case letters. Therefore, RAS66K will handle them as different symbols. If the code is assembled without the /CD option, then a double-definition error will occur.

(4) Warning Check Control (/W, /NW)

■ Syntax ■

```
/W [ warning_type ]
/NW [ warning_type ]
```

■ Description ■

The warnings checked by RAS66K during assembly, are classified into several groups. The /W and /NW options are used to enable and disable particular warnings.

The /W option enables warnings of the types specified by *warning_type*.

The /NW option disables warnings of the types specified by *warning_type*.

The *warning_type* is a combination of characters that indicate types of warnings. Table 4-3 shows the relationship of the characters and the warning types they indicate.

Table 4-3. Characters That Indicate Warning Types

Character	Type Of Warnings
R	Relocatable segment definition checks
P	Directive coding checks
E	Expression coding checks
A	Addressing coding checks
U	USING directive checks
S	SFR access attribute checks

If *warning_type* is omitted, then all warning types will be assumed specified. In other words, if only /W is specified, then all warnings will be checked. If only /NW is specified, then no warning checks will be performed.

The default is to check all warnings.

Refer to Section 4.15.2.3, “Warning Messages,” regarding actual warning messages and their respective warning types.

■ Example ■

To assemble the source file FOO.ASM without performing USING directive warning checks, type the following.

```
RAS66K FOO.ASM /NWEU
```

(5) Include File Path Specification (/I)

■ Syntax ■

```
/Iinclude_path
```

■ Description ■

The /I option specifies the path of files read by the INCLUDE directive. RAS66K searches for include files in the following order.

(1) First, an include file is searched for in the current directory. If the target file exists in the current directory, then that file will be read.

(2) If the target file does not exist in the current directory and an include file path has been specified with the /I option, then the target file will be searched for on that path.

Refer to Section 4.12.15, “Using Include Files (INCLUDE),” regarding INCLUDE directives.

■ Example ■

To assemble the source file FOO.ASM and read include files on the path A:\USR\PROG\INC, type the following.

```
RAS66K FOO.ASM /IA:\USR\PROG\INC
```

(6) Output of C Source Level Debugging Information (/CC)

■ Syntax ■

```
/CC
```

■ Description ■

The /CC option is specified when the source file is an output file of CC66K. When the /CC option is specified, the source file will be assembled and an object file that includes C source level debugging information will be generated. This option enables C source level debugging. When the /CC option is not specified, C source level debugging will not be possible.

The name of the file read will be that of the source file with the extension “.DBG.” RAS66K also assumes that this file will be located in the same directory as the source file. If RAS66K cannot find the file, then an error will occur and assembly will stop.

RAS66K will not distinguish between upper-case and lower-case letters by default. However, if the /CC option is specified, then RAS 66K will distinguish between upper-case and lower-case letters by default.

■ Example ■

```
RAS66K CCFOO /CC
```

In this example, RAS66K will assemble CCFOO.ASM, a source file output by CC66K, and generate an object file that includes C source level debugging information.

(7) Saving File Read Buffer (/V)

■ Syntax ■

```
/V [ buffer_size ]
```

■ Description ■

The /V option is used to reserve a buffer for reading the source file. A 512-byte buffer is provided to read normal files, but the /V option can reserve a buffer up to 32,767 (7FFFH) bytes.

The *buffer_size* specifies an integer constants that indicates the size of the buffer to be reserved. If a value greater than 32,767 is specified or if the /V option is specified without a buffer size, then a 32,767-byte buffer will be reserved.

The purpose of using the /V option is to speed up assembly. In particular, pass 2 processing accesses multiple files simultaneously, so the file buffer size will greatly influence assembly speed.

The effectiveness of the /V option is especially conspicuous when using comparatively slow disks, such as floppy disks. Conversely, its effectiveness is not so noticeable when using fast disks, such as hard disks or RAM disks.

■ Example ■

```
RAS66K FOO.ASM /V4000H
```

In this example, a 4000H-byte buffer is reserved for reading the source file.

(8) Print File Generation Control (/PR, /NPR)

■ Syntax ■

```
/PR [ print_file ]
/NPR
```

■ Description ■

When the /PR option is used, RAS66K will generate a print file. The *print_file* specifies the name of the print file. Refer to Section 4.2, “File Specification Defaults,” regarding defaults when the operand or part of the file specification is omitted.

When the /NPR option is used, RAS66K will not generate a print file. However, if the /A option is specified at the same time, then RAS66K will generate a print file, even though /NPR is specified.

If both the /PR and /NPR options are omitted, then RAS66K will generate a print file with the source file name and an extension changed to “.PRN.”

■ Corresponding Directive ■

The PRN directive can be used in a program instead of specifying the /PR option. The NOPRN directive can be used in a program instead of specifying the /NPR option.

Refer to Section 4.12.23.1, "Print File Output Control (PRN, NOPRN)."

■ Example ■

```
RAS66K FOO.ASM /PROUTPUT.LST
```

This example specifies the generation of a print file OUTPUT.LST.

```
RAS66K FOO.ASM /NPR
```

This example specifies no generation of a print file.

(9) Absolute Print File Generation (/A)

■ Syntax ■

```
/A [ abl_file ]
```

■ Description ■

The /A option is specified to generate an absolute print file. An absolute print file includes no unresolved machine code or unresolved address information, but does include all resolved information.

The *abl_file* is the name of an ABL file. ABL files are generated by RL66K in a binary format with the information needed to generate absolute print files.

Even if the /A option is used, the file name specification rules for the /PR option do not change. However, the default extension of ordinary print files is ".PRN," but that of absolute print files is ".APR."

For details on how absolute print files are created, refer to Chapter 8, "Absolute Print File Generation."

■ Example ■

To generate the absolute print file of source file FOO.ASM, type the following. In this example, the ABL file APRINFO.ABL will be read.

```
RAS66K FOO.ASM /AAPRINFO
```

(10) Assembly List Output Control (/L, /NL)

■ Syntax ■

```
/L
/NL
```

■ Description ■

When the /L option is specified, RAS66K will output each statement to the assembly list until it encounters a NOLIST directive in the program.

When the /NL option is specified, RAS66K will not output statements to the assembly list until it encounters a LIST directive in the program. However, statements that include errors will always be output to the assembly list even if the /NL option is specified.

The default is the /L specification.

Assembly lists are explained in Section 4.13, “Print File.” Refer to Section 4.12.23.6, “Assembly List Output Control (LIST, NOLIST),” regarding the LIST and NOLIST directives.

■ Corresponding Directive ■

The LIST and NOLIST directives have nearly the same functions as these options. However, the options take effect from the start of the program, while the LIST and NOLIST directives take effect after the source statement in which they are coded.

■ Example ■

To assemble the source file FOO.ASM and output its contents to the assembly list, type the following.

```
RAS66K FOO.ASM /L
```

To assemble the source file FOO.ASM without sending its contents to the assembly list, type the following.

```
RAS66K FOO.ASM /NL
```

(11) Symbol List Output Control (/S, /NS)

■ Syntax ■

```
/S
/NS
```

■ Description ■

When the /S option is specified, RAS66K will output all user symbol information to a symbol list. When the /NS option is specified, RAS66K will not generate a symbol list.

The default is to not generate a symbol list.

Symbol lists are explained in Section 4.13, “Print File.” Refer to Section 4.12.23.7, “Symbol List Output Control (SYM, NOSYM),” regarding the SYM and NOSYM directives.

■ Corresponding Directive ■

The SYM directive can be used in a program instead of specifying the /S option. The NOSYM directive can be used in a program instead of specifying the /NS option. If both an option and directive are specified, then the option setting will take precedence.

■ Example ■

To output all symbols used in the source file FOO.ASM to the assembly list, type the following.

```
RAS66K FOO.ASM /S
```

To assemble the source file FOO.ASM without generating a symbol list, type the following.

```
RAS66K FOO.ASM /NS
```

(12) Cross-Reference List Output Control (/R, /NR)

■ Syntax ■

```
/R
/NR
```

■ Description ■

When the /R option is specified, RAS66K will output the line numbers in which each user symbol is used to a cross-reference list. When the /NR option is specified, RAS66K will not generate a cross-reference list.

To be more precise, REF and NOREF directives coded in the program will affect the generation of the cross-reference list. Even when the /R option is specified, if a NOREF directive is coded in the program, then no line numbers from its line until the line of a REF directive is encountered will be output to the cross-reference list. Conversely, even when the /NR option is specified, if a REF directive is coded in the program, then the line numbers from its line until the line of a NOREF directive is encountered will be output to the cross-reference list. In other words, the REF and NOREF directives have the role of cross-reference list output switches.

However, the use described above is not often needed. Accordingly, you can use the /R option to generate a cross-reference list and the /NR option to not generate one.

The default is to not generate a cross-reference list.

Cross-reference lists are explained in Section 4.13, “Print File.” Refer to Section 4.12.23.8, “Cross-Reference List Output Control (REF, NOREF),” regarding the REF and NOREF directives.

■ Corresponding Directive ■

The REF and NOREF directives have nearly the same functions as these options. However, the options take effect from the start of the program, while the REF and NOREF directives take effect after the source statement in which they are coded.

■ Example ■

To generate a cross-reference list of all symbols used in the source file FOO.ASM, type the following.

```
RAS66K FOO.ASM /R
```

To assemble the source file FOO.ASM without generating a cross-reference list, type the following.

```
RAS66K FOO.ASM /NR
```

(13) Print File Characters Per Line Specification (/PW)

■ Syntax ■

```
/PW page_width
```

■ Description ■

The /PW option specifies the number of characters per line in the print file.

The *page_width* specifies an integer constant for the number of characters. This is the number of 1-byte characters. The *page_width* can be specified in the following range.

79 to 132

If a value less than 79 is specified, then the characters per line in the print file will become 79. If a value greater than 132 is specified, then the characters per line in the print file will become 132.

The default characters per line in the print file is set to 79.

■ Corresponding Directive ■

The PAGE directive can be used in a program instead of specifying the /PW option. The number of characters set as the second operand of the PAGE directive will perform the same setting as with the /PW option. If both the /PW option and PAGE directive are specified, then the /PW option setting will take precedence.

Refer to Section 4.12.22.3, “Lines Per Page and Characters Per Line Specification (PAGE with operands),” regarding the PAGE directive.

■ Example ■

```
RAS66K FOO.ASM /PW132
```

In this example, 132 characters per print file line is specified.

(14) Print File Lines Per Page Specification (/PL)

■ Syntax ■

```
/PL page_length
```

■ Description ■

The /PL option specifies the number of lines per page in the print file.

The *page_length* specifies an integer constant for the number of lines. This includes the print file header and surrounding blank lines. The *page_length* can be specified in the following range.

10 to 65535

If a value less than 10 is specified, then the lines per page in the print file will become 10. If a value greater than 65535 is specified, then the lines per page in the print file will become 65535.

The default lines per page in the print file is set to 60.

■ Corresponding Directive ■

The PAGE directive can be used in a program instead of specifying the /PL option. The number of lines set as the first operand of the PAGE directive will perform the same setting as with the /PL option. If both the /PL option and PAGE directive are specified, then the /PL option setting will take precedence. Refer to Section 4.12.22.3, "Lines Per Page and Characters Per Line Specification (PAGE with operands)," regarding the PAGE directive.

■ Example ■

```
RAS66K FOO.ASM /PL100
```

In this example, 100 lines per print file page is specified.

■ Information ■

In some circumstances, you may not want to split the print file into pages. For example, you may output to a printer using software that supports page feed functions or you may want to read the print file on your PC screen. RAS66K does not provide options or directives that disable page breaks, but you can use the /PL option to prevent visible page breaks. Do this by specifying a sufficiently large value for /PL. For example, page breaks can be effectively disabled by setting the lines per page to 65535, as shown below.

```
RAS66K FOO.ASM /PL65535
```

(15) Tab Code Replacement (/T)

■ Syntax ■

```
/T [ tab_width ]
```

■ Description ■

When the /T option is used, tab codes used in the program will be replaced by the appropriate number of spaces. Use of the /T option enables properly aligned list output of the print file even when the printer does not recognize tab codes.

The *tab_width* is the number of spaces corresponding to one tab code. It can be specified as an integer constant from 1 to 16. If the *tab_width* is omitted, then the specification will be 8.

■ Corresponding Directive ■

The TAB directive can be used in a program instead of specifying the /T option. If both the /T option and TAB directive are specified, then the /T option setting will take precedence.

If neither the /T option or the TAB directive is specified, then tab codes will be output as is to the print file.

■ Example ■

```
RAS66K FOO.ASM /T4
```

In this example, tab codes used in the program will be replaced by up to 4 spaces to align the print file.

(16) Object File Output Control (/O, /NO)

■ Syntax ■

```
/O [ object_file ]
/NO
```

■ Description ■

When the /O option is specified, RAS66K will generate an object file. The *object_file* specifies the object file name. Refer to Section 4.2, “File Specification Defaults,” when all or part of the file specification is omitted.

When the /NO option is specified, RAS66K will not generate an object file.

If both the /O and /NO option are omitted, then an object file will be generated. The object file name will be the source file name with the extension “.OBJ.”

■ Corresponding Directive ■

The OBJ directive can be used in a program instead of specifying the /O option. The NOOBJ directive can be used in a program instead of specifying the /NO option. If both an option and directive are specified, then the option setting will take precedence.

Refer to Section 4.12.24.1, “Object File Output Control (OBJ, NOOBJ),” regarding the OBJ and NOOBJ directives.

■ Example ■

```
RAS66K FOO.ASM /OOUTPUT.OBJ
```

In this example, the generation of object file OUTPUT.OBJ is specified.

```
RAS66K FOO.ASM /NO
```

In this example, no generation of an object file is specified.

(17) Output of Assembly Level Debugging Information (/D, /ND)

■ Syntax ■

```
/D
/ND
```

■ Description ■

When the /D option is specified, RAS66K will output assembly level debugging information to the object file. Symbolic debugging of the program is made possible by including debugging information in the object file

When the /ND option is specified, RAS66K will not output assembly level debugging information to the object file.

The default is to not output debugging information to the object file

■ Corresponding Directive ■

The DEBUG directive can be used in a program instead of specifying the /D option. The NODEBUG directive can be used in a program instead of specifying the /ND option. If both an option and directive are specified, then the option setting will take precedence.

Refer to Section 4.12.24.2, “Assembly Level Debugging Information Output Control (DEBUG, NODEBUG),” regarding the DEBUG and NODEBUG directives.

■ **Example** ■

```
RAS66K FOO.ASM /D
```

In this example, assembly level debugging information will be output to the object file.

(18) Error Message Output Control (/E, /NE)

■ **Syntax** ■

```
/E [ error_file ]
/NE
```

■ **Description** ■

The /E option tells RAS66K where to output error messages. When an error file name is specified for *error_file*, error messages will be output to that file. Refer to Section 4.2, "File Specification Defaults," when all or part of the file specification is omitted.

The /NE option tells RAS66K to display error messages on the screen (standard error).

The default is to display error messages on the screen.

By using the /E option, the output destination of error messages can be controlled only for assembler error messages and warnings. To output to a file error messages including fatal error messages and internal processing error messages, use the DOS redirection function.

■ **Corresponding Directive** ■

The ERR directive can be used in a program instead of specifying the /E option. The NOERR directive can be used in a program instead of specifying the /NE option. If both an option and directive are specified, then the option setting will take precedence.

Refer to Section 4.12.25, "Error Message Output Control (ERR, NOERR)," regarding the ERR and NOERR directives.

■ **Example** ■

```
RAS66K FOO.ASM /EERROR.LST
```

This example specifies generation of an error file called ERROR.LST.

(19) Generation of EXTRN Declaration Files (/X)

■ Syntax ■

```
/X [ extrn_file ]
```

■ Description ■

When the /X option is specified, RAS66K will generate an EXTRN declaration file. The *extrn_file* specifies the EXTRN declaration file. Refer to Section 4.2, “File Specification Defaults,” when all or part of the file specification is omitted.

The default is to not generate an EXTRN declaration file.

Refer to Section 4.14, “EXTRN Declaration Files.”

■ Example ■

```
RAS66K FOO.ASM /XEXTRN.INC
```

This example specifies generation of an EXTRN declaration file called EXTRN.INC.

4.3.3 Termination Code

RAS66K returns a value corresponding to its termination state when assembly ends. This value is called a termination code. The termination code can be detected using a batch file. Termination codes are as follows.

Table 4-4. Termination Codes

Code	Description
0	No errors.
1	Warnings occurred.
2	Assembler errors occurred.
3	Fatal error, internal processing error, or DCL error occurred. Assembly forcibly terminated.

4.3.4 Symbol Table

RAS66K maintains a data table to manage symbols. This is generally called the symbol table. Symbols that appear in the program and their related information are stored in the symbol table. Information needed to generate a cross-reference list will also be stored in this table.

The size of the symbol table depends on the amount of free memory available. If the memory capacity of the symbol table becomes insufficient, then at that point RAS66K will display an error message and stop assembling. In such cases the following counter-measures are necessary.

- Reduce the number of user symbols.
- Shorten the length of user symbols.
- Split up the source file.

4.4 Creating Programs

A program consists of a sequence of source statements written in OLMS-66K Series assembly language. Source statements are constructed from microcontroller instructions, directives, operands, and comments. This section explains the fundamentals of creating programs.

4.4.1 Initial Program Code

To create an OLMS-66K Series program, you need to specify several items at the start of the program using directives.

- Target microcontroller specification
  Specify the name of the target microcontroller using the TYPE directive.

- COMMON area specification
  Specify the value set in SCB (which is the last address of the COMMON area) using the COMMON directive.

- Memory model specification
  Specify the memory model used with the MODEL directive.

- ROM window area specification
  Specify the first and last address of the ROM window area using the WINDOW directive.

Among these items, the target microcontroller name absolutely must be specified with the TYPE directive. Specify the other items as necessary. For example, if the CPU core is nX-8/100~400, then there is no need to specify the memory model or ROM window. If there is only one physical segment in data memory space, then there is no need to specify the COMMON area.

If the COMMON area, memory model function, or ROM window function are valid for a target microcontroller but the program is assembled without making their initial settings, then assembler processing may appear to terminate correctly. However, the actual program settings and the settings selected by RAS66K might be contradictory. Therefore, you must make the corresponding initial settings when using a target microcontroller for which the COMMON area, memory model function, or ROM window function are valid.

These initial settings are required before RAS66K begins interpreting program contents, so they need to be specified at the start of the program. Refer to Section 4.4.1.5, “Code Position Restrictions,” regarding the position of each directive.

4.4.1.1 Target Microcontroller Specification

To specify the target microcontroller in a program, specify the DCL file name for the microcontroller as the operand of the TYPE directive. The syntax of the TYPE directive is as follows.

■ Syntax ■

```
    TYPE (dcl_name)
```

RAS66K will read the information of a DCL file with base name *dcl_name* and extension “.DCL.”

Microcontroller names and DCL file base names are very similar, but they are different in that microcontroller names start with “MSM” while DCL file base names start with “M.” For example, if your microcontroller is “MSM66507,” then specify “M66507” in the TYPE directive.

RAS66K searches for the DCL file in the following order.

(1) Current directory
(2) Directory that contains RAS66K.EXE
(3) Directory specified in environment variable DCL

Refer to the DCL66K.DOC file for correct *dcl_name* specifications. If no TYPE directive is specified or if no DCL file can be found, then RAS66K will generate a DCL error and forcibly terminate.

RAS66K reads the contents of the DCL file at the start of its assembly process. It will read to the end of the DCL file even if there is an error in its contents. In such a case, it will display all DCL errors generated and forcibly terminated. If the DCL file is read without problems, then RAS66K will then assemble the source file.

■ Example ■

If the target microcontroller is MSM66507, then specify the following.

```
      TYPE (M66507)
```

4.4.1.2 COMMON Area Specification

Microcontrollers based on the nX-8/300 or nX-8/500 CPU core have multiple physical segments in data memory space. They contain an area COMMON to all physical segments called the *COMMON area*. There are four possible addresses for the end of the COMMON area. One of these can be selected by setting the value of BCB in the PSW from 0 to 3.

To inform RAS66K of the value of BCB, specify the value set in BCB as the operand of the COMMON directive. The syntax of the COMMON directive is as follows.

■ Syntax ■

```
   COMMON bcb_value
```

The *bcb_value* specifies the value 0-3 actually set in BCB. The four possible addresses for the end of the COMMON area differ depending on the target microcontroller.

■ Example ■

When BCB is set to 3, specify the following COMMON directive.

```
      COMMON 3
```

4.4.1.3 Memory Model Specification

When using a microcontroller with a nX-8/500 CPU core and multiple physical segments, a memory model must be specified. To specify the memory model within the program, specify it as the operand of the MODEL directive. The syntax of the MODEL directive is as follows.

■ Syntax ■

```
MODEL memory_model
```

Specify one of the following for *memory_model*.

memory_model	**Memory Model**
SMALL	SMALL memory model
COMPACT	COMPACT memory model
MEDIUM	MEDIUM memory model
LARGE	LARGE memory model

Refer to Section 3.2.4.3, “Memory Models.”

■ Example ■

When using the LARGE memory model, specify the following.

```
    MODEL LARGE
```

4.4.1.4 ROM Window Area Specification

When using a microcontroller with the nX-8/500 CPU core, the ROM window function is available for use. The ROM window function allocates a particular area of data memory space to the same address range of program memory space. This area is called the ROM window function.

To use the ROM window function, write the values indicating the address range of the ROM window area to the SFR ROMWIN register. To inform RAS66K that the ROM window function is being used, specify the start and end address of the ROM window area as operands of the WINDOW directive. The syntax of the WINDOW directive is as follows.

■ Syntax ■

```
WINDOW romwindow_start, romwindow_end
```

The *romwindow_start* is a constant expression the indicates the start address of the ROM window area. The *romwindow_start* must be at least 1000H, and its lower 12 bits must be 000H.

The *romwindow_end* is a constant expression the indicates the end address of the ROM window area. The *romwindow_end* must be at least 1000H, and its lower 12 bits must be FFFH.

Refer to Section 3.2.4.2 (9), “ROM Window Area.”

■ Example ■

When the ROM window start address is set to 5000H and end address is set to 6FFFH, specify the following WINDOW directive.

```
      WINDOW 5000H, 6FFFH
```

4.4.1.5 Code Position Restrictions

The TYPE, COMMON, MODEL, and WINDOW directives described above inform RAS66K of initial program settings. RAS66K manages memory space and performs SFR access checks based on this information. Accordingly, these settings must be coded at the beginning of the program.

This leads to the following restrictions on the code position of each directive.

- Directives that may be coded before a TYPE or MODEL directive are as follows.

```
INCLUDE DEFINE   IF       IFDEF    IFNDEF   ELSE
PRN     NOPRN    OBJ      NOOBJ    ERR      NOERR    DEBUG
NODEBUG LIST     NOLIST   SYM      NOSYM    REF      NOREF
PAGE    DATE     TITLE    TAB
```

- Directives that may be coded before a COMMON or WINDOW directive are as follows.

```
INCLUDE DEFINE   IF       IFDEF    IFNDEF   ELSE
PRN     NOPRN    OBJ      NOOBJ    ERR      NOERR    DEBUG
NODEBUG LIST     NOLIST   SYM      NOSYM    REF      NOREF
PAGE    DATE     TITLE    TAB      TYPE     EQU      SET
CODE    CBIT     DATA     BIT      EDATA    EBIT     MODEL
```

■ Example ■

The example below follows the rules above.

```
     SYM
     REF
     TAB    4

     TYPE    (M66507)
     MODEL   LARGE

ROMWBASE DATA    5000H
ROMWTOP  DATA    7FFFH
BCB      EQU     3

     WINDOW  ROMXBASE,ROMWTOP
     COMMON  BCB
```

4.4.2 Program End Specification

The END directive specifies the end of a program. The syntax of the END directive is as follows.

■ **Syntax** ■

```
END
```

RAS66K will not assemble statements following the END directive. If a program has no END directive, then RAS66K will assemble until the end of the file.

■ **Example** ■

Here is a simple example of program format.

```
    TYPE(M66507)
    NOP
    END
```

This program consists of three source statements. The first source statement “TYPE(M66507)” is a directive that specifies the microcontroller. The second source statement “NOP” is a microcontroller instruction. The third source statement “END” is the directive that indicates the end of the program.

4.4.3 Writing Source Statements

There are three types of source statements.

- Instruction statements
- Directive statements
- Special statements

Source statements always reside in some logical segment. That logical segment is determined by the position in the program where the source statement is located.

4.4.3.1 Writing Instruction Statements

Instruction statements are source statements that code OLMS-66K Series instructions. Instruction statements have four fields. The order in which the fields appear cannot be changed. Instruction statements end with a carriage return code.

Instruction statements have the following syntax.

■ **Syntax** ■

```
[label_field] operation_field [operand_field] [comment_field]
```

■ Description ■

Each field is described below.

- *label_field*
  A label is specified in the labelfield. A label is a symbol that takes the address of the instruction statement as its value. A colon (:) must be coded after the label. The label takes an address within the logical segment in which its instruction statement resides. This is sometimes expressed by saying that the label resides in the logical segment.

- *operation_field*
  A microcontroller instruction is specified in the operationfield. Refer to related documentation for details on instructions.

- *operand_field*
  Operands required by the microcontroller instruction are specified in the operandfield. Refer to related documentation regarding operands required by instructions. Refer to Section 4.11, “Addressing Modes,” for operand syntax.

- *comment_field*
  A string that begins with a semicolon (;) and ends with a carriage return code is coded in the commentfield. Strings coded in this field have no effect on assembly.

■ Example ■

```
        MOV ER0,#0FFFFH   ;comment field
```

This source statement is an instruction statement. MOV is the microcontroller instruction. “ER0,#0FFFFH” are the operands. Following the semicolon (;) is a comment.

4.4.3.2 Writing Directive Statements

Directive statements are source statements that code RAS66K directives. The syntax of directive statements differs for each directive. However, comments that begin with a semicolon (;) and end with a carriage return code can be coded at the end of all directive statements. The end of a directive statement is a carriage return code.

Refer to Section 4.12, “Directives,” to see how to write each directive statement.

4.4.3.3 Writing Special Statements

Special statements are source statements that are neither instructions nor directives. A special statement is a source statement that consists of a carriage return only or a label or comment only followed by a carriage return. Special statements have the following syntax.

■ Syntax ■

```
    [label_field] [comment_field]
```

■ **Example** ■

```
LABEL1:
LABEL1X:
    MOV ER0,#0FFFFH              ;comment field

LABEL2:         ;source statement with label and comment only
                ;source statement with comment only
```

This example includes several special statements. The first and second lines are source statements with labels only. The fourth line is a source statement with only a carriage return code. The fifth line is a source statement with a label and comment only. The sixth line is a source statement with a comment only.

4.4.4 Block Comments

■ **Syntax** ■

/* *characters* */

■ **Description** ■

Multi-line comments are possible by using block comments.

A block comment starts with /* and ends with */. RAS66K assembly ignores these block comments. The characters are coded with a string. The string can be built from characters including carriage return codes, except for the block comment termination code (*/). Thus, block comments can cross multiple lines. Block comments can be nested.

■ **Example** ■

Below is an example that makes use of several block comments.

```
SYM1    EQU 100H          /* This is
                a block comment. */
MOV     ER0,#SYM1         ;comment field
/*
                          This line is a block comment
*/
/* This is /* a block comment /*
nested */ three levels */
deep.
*/
```

4.5 Coding Logical Segments

Logical segments are contiguous areas in address space. When you create a program using OLMS-66K assembly language, all source statements reside in these logical segments.

Logical segment types correspond to address space as below.

Table 4-5. Logical Segments

Logical Segment	Description
CODE segment	Logical segment that resides in CODE address space.
DATA segment	Logical segment that resides in DATA address space.
BIT segment	Logical segment that resides in BIT address space.
EDATA segment	Logical segment that resides in EDATA address space.
EBIT segment	Logical segment that resides in EBIT address space.

Which parts of the source file become which logical segments is defined using directives. Below are the directives that define each logical segment.

Table 4-6. Directives For Defining Logical Segments

Logical Segment	Directive
CODE segment	CSEG, RSEG
DATA segment	DSEG, RSEG
BIT segment	BSEG, RSEG
EDATA segment	ESEG, RSEG
EBIT segment	EBSEG, RSEG

There are two types of directives for defining logical segments. The RSEG directive is explained later. An example using CSEG, DSEG, and BSEG directives is described below.

■ Example ■

```
DSEG     ─┐
.         │
.         │  DATA segment range
.        ─┘
BSEG     ─┐
.         │
.         │  BIT segment range
.        ─┘
CSEG     ─┐
.         │
.         │  CODE segment range
.        ─┘
```

In this example, the DATA segment begins where the DSEG directive is coded. After that, the BIT segment begins where the BSEG directive is coded. After that, the CODE segment begins where the CSEG directive is coded.

Thus, a logical segment definition is valid until the next logical segment is defined. A logical segment in a program is a group of contiguous source statements. A source statement will always reside in some logical segment.

A logical segment is allocated within a single physical segment. Logical segments that extend across multiple physical segments cannot be defined. In addition, logical segments cannot be placed in the SFR area.

4.5.1 Source Statements Coded In Logical Segments

To create a program in assembly language, one must be aware of logical segments. In other words, source statements for CODE address space must coded after the CODE segment is defined. Similarly, source statements for DATA address space, BIT address space, EDATA address space, and EBIT address space must be coded after the corresponding segment is defined. The following source statements are coded mainly in their respective logical segments.

(1) Source statements coded mainly in the CODE segment

- Instruction statements
- DB and DW directive statements for initializing specified values
- GJMP and GCAL directives for conversion to optimal branch instructions

(2) Source statements coded mainly in the DATA segment

- DS directive statements for reserving byte-wide data

(3) Source statements coded mainly in the BIT segment

- DBIT directive statements for reserving bit-wide data

(4) Source statements coded mainly in the EDATA segment

- DB and DW directive statements for initializing specified values
- DS directive statements for reserving byte-wide data

(5) Source statements coded mainly in the EBIT segment

- DBIT directive statements for reserving bit-wide data

Source statements other than those above can be coded in all logical segments.

4.5.2 Absolute Segments And Relocatable Segments

The various logical segments can be divided into two types.

- Absolute segments
- Relocatable segments

Absolute segments are logical segments for which RAS66K can determine addresses during assembly. Relocatable segments are logical segments for which RAS66K cannot determine addresses during assembly. RL66K determines relocatable segment addresses.

Each type of logical segment is defined using directives.

4.5.2.1 Absolute Segments

Absolute segments are logical segments for which RAS66K can determine addresses during assembly. Absolute segments are managed in each physical segment. The following items can be specified when defining an absolute segment.

- Starting address of the absolute segment.
- Physical segment address where the absolute segment resides.

If these items are not specified, then they will inherit their specifications form the preceding specified absolute segment. For details on how they are inherited, refer to Section 4.12.7, "Absolute Segment Definitions."

If multiple absolute segments have the same physical segment address within a single program, then unless the starting addresses of the absolute segments are specified, these absolute segments will be placed contiguously in memory. In other words, these absolute segments will become a single absolute segment.

The source statements from the start of the program until the first logical segment is defined will be an absolute segment residing in code address space. Accordingly, if no logical segments are defined in a program, then the entire program will be this segment. This segment's physical segment address will be 0, and it will start at address 0 within the physical segment.

Terminology for absolute segments depends on the address space in which they reside.

Table 4-7. Absolute Segments

Absolute Segment	Description
Absolute CODE segment	Absolute segment residing in CODE address space.
Absolute DATA segment	Absolute segment residing in DATA address space.
Absolute BIT segment	Absolute segment residing in BIT address space.
Absolute EDATA segment	Absolute segment residing in EDATA address space.
Absolute EBIT segment	Absolute segment residing in EBIT address space.

The absolute segments that reside in each address space are defined using the following directives.

Table 4-8. Directives For Defining Absolute Segments

Directive	Description
CSEG	Defines an absolute CODE segment.
DSEG	Defines an absolute DATA segment.
BSEG	Defines an absolute BIT segment.
ESEG	Defines an absolute EDATA segment.
EBSEG	Defines an absolute EBIT segment.

■ **Example** ■

```
     CSEG    #1 AT 100H     ;Absolute CODE segment definition (1)
     NOP

     DSEG    #0             ;Absolute DATA segment definition (1)
LABEL7:
     DS     2

     CSEG    #1             ;Absolute CODE segment definition (2)
     MOV     ER0,#0

     DSEG                   ;Absolute DATA segment definition (2)
LABEL8:
     DS     2
```

In this example, four absolute segments are defined. Absolute CODE segment definition (1) specifies the physical segment address as 1 and the starting address in that physical segment as 100H. Absolute CODE segment definition (2) does not specify a starting address, so it will inherit the address of absolute CODE segment (1) in the same physical segment. In other words, the "MOV R0,#0" instruction will be placed at the address following the NOP instruction.

Absolute DATA segment definition (1) defines the physical segment address as 0, but does not specify a starting address in that physical segment. Absolute DATA segment definition (2) does not specify either a physical segment address or a starting address. This segment will therefore inherit the address of absolute DATA segment (1).

4.5.2.2 Relocatable Segments

Relocatable segments are logical segments for which RAS66K cannot determine addresses during assembly. RL66K determines relocatable segment addresses. RAS66K manages relocatable segments with segment symbols. If there are multiple relocatable segments in a single source file defined using the same segment symbol, then these relocatable segments will be allocated contiguously in memory. They will be allocated in their order of definition.

If relocatable segments are defined in different source files using the same segment symbol, then RL66K will link them before allocating them in memory. The value of the segment symbol will be the starting address of the linked segment. The relocatable segments linked by RL66K are called partial segments. Refer to Section 5.5, “Link Processing,” regarding address resolution and linking order of relocatable segments.

Terminology for relocatable segments depends on the address space in which they reside.

Table 4-9. Relocatable Segments

Relocatable Segment	Description
Relocatable CODE segment	Relocatable segment residing in CODE address space.
Relocatable DATA segment	Relocatable segment residing in DATA address space.
Relocatable BIT segment	Relocatable segment residing in BIT address space.
Relocatable EDATA segment	Relocatable segment residing in EDATA address space.
Relocatable EBIT segment	Relocatable segment residing in EBIT address space.

The relocatable segments that reside in each address space are defined using the following directive.

Table 4-10. Directive For Defining Relocatable Segments

Directive	Description
RSEG	Defines a relocatable segment.

Segment symbols are specified as operands of RSEG directives. They are defined using SEGMENT directives. When one defines a segment symbol, one also specifies the address space in which their relocatable segments will reside is also specified.

■ **Example** ■

```
CODESEG2    SEGMENT CODE    #1    ;Segment symbol (CODESEG2) definition
DATASEG2    SEGMENT DATA          ;Segment symbol (DATASEG2) definition

        RSEG    CODESEG2          ;Relocatable CODE segment definition
        NOP

        RSEG    DATASEG2          ;Relocatable DATA segment definition
LABEL9:
        DS    2

        RSEG    CODESEG2          ;Relocatable CODE segment definition
        MOV     ER0,#0
```

In this example, two segment symbols (CODESEG2 and DATASEG2) are defined. These are used to define relocatable segments. Segment symbols are defined using the SEGMENT directive. Relocatable segments are defined using the RSEG directive. When one defines a segment symbol, one also specifies the address space in which relocatable segments corresponding to that segment symbol will be allocated. The physical segment in which they will be allocated can also be specified.

In this example, when the segment symbol CODESEG2 is defined, its allocation in code address space (CODE) and allocation in a physical segment (#1) are specified. When the segment symbol DATASEG2 is defined, its allocation in data address space (DATA) is specified, but its allocation in which physical segment is not specified. In this case RL66K will determine in which physical segment the relocatable segment will be allocated.

Also in this example, two relocatable CODE segments are defined using the one segment symbol CODESEG2. These two relocatable CODE segments will be allocated contiguously in memory. In other words, the “MOV ER0,#0” instruction will be placed following the NOP instruction.

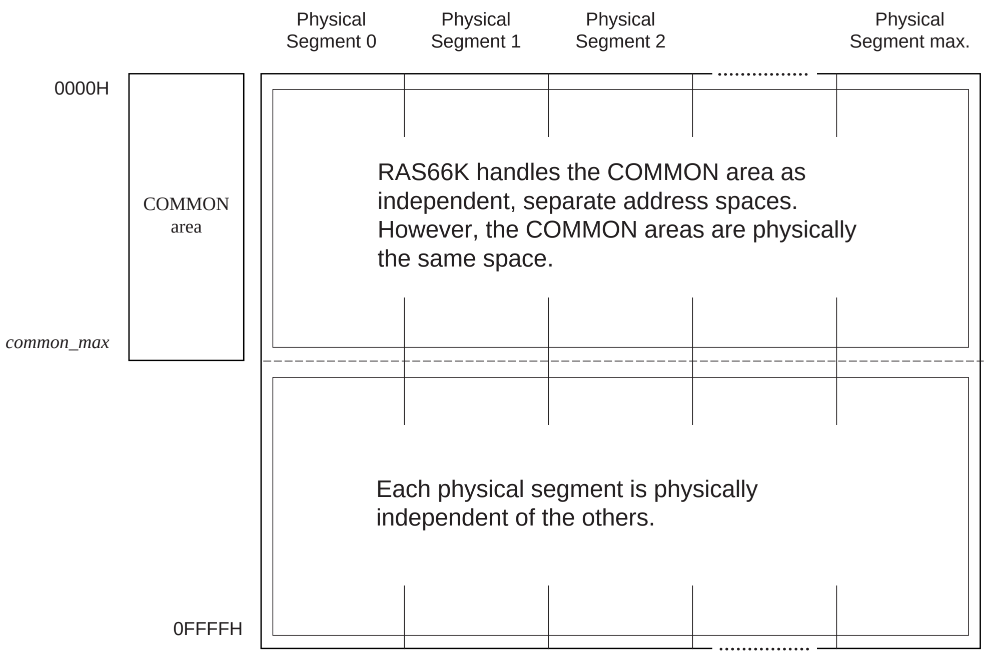
As shown by this example, an absolute address for allocated a relocatable segment cannot be specified. The purpose of relocatable segments is to allow programs to be written independent of the absolute addresses at which those logical segments will be placed. To specify the absolute address at which a relocatable segment is to be allocated, specify it as an option when RL66K is invoked.

4.5.3 COMMON Area

4.5.3.1 Data Memory Space Seen By RAS66K

The COMMON area of data memory space is an area common to all physical segments. This area does not reside in any particular physical segment. In other words, the concept of physical segment addresses does not exist within the COMMON area.

In order to realize this feature, RAS66K handles the COMMON area and the individual physical segments as independent logical spaces. Data memory space can therefore be considered to have the following memory map.


Physical Segment 0
Physical Segment 1
Physical Segment 2
Physical Segment max.
0000H
COMMON area
common_max
0FFFFH
RAS66K handles the COMMON area as independent, separate address spaces. However, the COMMON areas are physically the same space.
Each physical segment is physically independent of the others.

As shown in this figure, the COMMON area is independent of other physical segments. Its address range is 0000H to *common_max*. The address range of each physical segment is 0000H to 0FFFFH.

The programmer should be aware of the following point. This figure appears to show that the COMMON area and the address spaces from 0000H to *common_max* are mutually independent. However, this is how RAS66K views the address space. Actually the range from 0000H to *common_max* exists in only one physical space.

4.5.3.2 Segment Allocation To COMMON Area

The programming technique for allocating a DATA segment or BIT segment to the COMMON area is slightly different from the techniques introduced until now. The difference is that the keyword "COMMON" must be added.

Below is an example showing an absolute DATA segment allocated to the COMMON area.

■ Example ■

```
DSEG AT 200H COMMON
DS   10H
```

In this example, a 10H-byte area from COMMON area address 200H will be reserved. In examples until now physical segment addresses like "#1" were specified, but here "COMMON" is specified.

Next is an example showing a relocatable DATA segment allocated to the COMMON area.

■ Example ■

```
COM_TBL  SEGMENT DATA COMMON
RSEG     COM_TBL
DS       10H
```

As in the first example this reserves a 10H-byte area, because it is a relocatable segment there is no start address. However, "COMMON" is specified when segment symbol COM_TBL is defined, so RL66K will allocate this segment to the COMMON area.

Thus, by specifying "COMMON" when a segment is defined (whether absolute or relocatable), it is guaranteed to be allocated to the COMMON area. Segments that are clearly allocated in the COMMON area with the "COMMON" specification are called common segments.

Ordinary segment addresses are expressed as a physical segment address and an offset address, but common segment addresses have no physical segment address. Addresses greater than the end address of the COMMON area cannot be reserved in the common segment. DSR checks are not performed when addresses in a common segment are accessed. These are the differences in how RAS66K handles common segments and ordinary segments.

■ Information ■

Specifications that allocate relocatable segments with special conditions, such as “COMMON,” are called *special area attributes*. Other special area attributes are “WINDOW,” which specifies allocation to the ROM window area, and “ACAL,” which specifies allocation to the ACAL area.

Specifications that place conditions on the boundary values of start addresses of relocatable segments are called *boundary value attributes*. These are “WORD,” which specifies allocation on word boundaries, and “PAGE,” which specifies allocation on page boundaries.

Refer to Section 4.12.8.1, “Segment Symbol Definition (SEGMENT),” regarding special area attributes and boundary value attributes.

4.5.4 Stack Segment

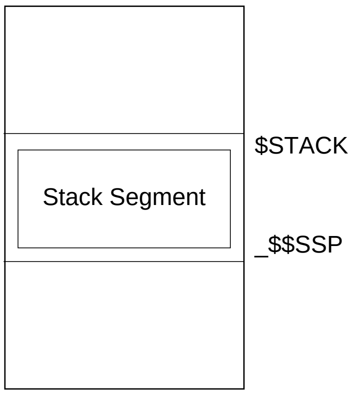
The stack segment is a relocatable segment that represents the stack area. The stack segment is allocated to physical segment 0 of data memory space.

Use the STACKSEG directive to define the stack segment. Specify the stack size as the operand of the STACKSEG directive.

The stack segment is a relocatable segment, so its start address and end address cannot be obtained directly. However, they can be obtained by referring symbols.

The stack segment start address can be obtained by referring the symbol $STACK. $STACK is the name of the stack segment. It is automatically generated by RAS66K when the stack segment is defined.

The initial value of SSP is also one less than the stack segment end address. It can be obtained by referring the symbol _$$SSP. _$$SSP is not automatically generated. To refer _$$SSP, it must be declared using the EXTRN directive before it is used in a program.



■ **Example** ■

```
STACKSEG    200H
EXTRN      DATA:_$$SSP
MOV        SSP,#_$$SSP
```

This example first defines a stack segment with stack size of 200H bytes. Then it declares _$$SSP with EXTRN. Finally it sets the stack area end address _$$SSP to the stack pointer SSP.

4.5.5 Overlapping Logical Segments

RAS66K does not check for overlapping addresses of different logical segments, but RL66K does. However, neither RAS66K nor RL66K checks for overlapping addresses within the same logical segment. Refer to the following examples.

■ Example 1 ■

```
CSEG  #0  AT  100H
DW   0
ORG   200H
DW   1

CSEG  #0  AT  200H
DW   2
```

In this example, two absolute segments overlap at address 200H. However, this will not cause an error during assembly. RL66K will output a message during linking.

■ Example 2 ■

```
CSEG  #0  AT  300H
DW   3
ORG   300H
DW   4
```

In this example, address 300H of the CODE segment is initialized twice. This is fundamentally incorrect code, but no error will occur during assembly or linking.

4.6 Location Counter

During assembly, RAS66K normally stores the address of the logical segment that it is currently assembling. The counters that store these addresses are called location counters.

Relocatable segments each have their own location counters. For absolute segments, a location counter is provided for each physical segment of address space.

4.6.1 Location Counter Initialization

(1) Initialization of location counters of relocatable segments

Relocatable segments each have their own location counters. When a segment symbol is defined, the corresponding location counter is initialized to zero.

(2) Initialization of location counters of absolute segments

For absolute segments, a location counter is provided for each physical segment of address space. Each location counter is initialized when RAS66K is invoked.

Absolute segment location counters are initialized in the following ways.

- CODE address space location counters

  All physical segment location counters are initialized to the minimum address of physical segments in CODE address space, as defined in the DCL file.

- DATA address space location counters

  The location counters of the COMMON area and physical segment 0 are initialized to the minimum address outside the SFR but within the offset addresses of DATA address space, as defined in the DCL file. Location counters of physical segments above 0 are initialized to the starting address of the separate area.

- BIT address space location counters

  The location counters of the COMMON area and physical segment 0 are initialized to the minimum address outside the SFR but within the offset addresses of BIT address space, as defined in the DCL file. Location counters of physical segments above 0 are initialized to the starting address (bit address) of the separate area.

- EDATA address space location counter

  The location counter of EDATA address space is initialized to the starting address of the EEPROM area, as defined in the DCL file.

- EBIT address space location counter

  The location counter of EBIT address space is initialized to the starting address (bit address) of the separate area, as defined in the DCL file.

4.6.2 Changing Location Counter Values

The various location counters are modified by using the microcontroller instructions or directives explained below.

- Starting address specification when absolute segment is defined.

  If a starting address is specified when an absolute segment is defined, then the location counter will be changed to that address value.

- Microcontroller instructions

  The value of CODE segment location counters will increase by the number of words in each instruction.

- GJMP, GCAL directive

  The value of CODE segment location counters will increase by the number of bytes of the converted branch instruction.

- DS directive

  The value of DATA, EDATA, and CODE segment location counters will increase by the value of the operand.

- DBIT directive

  The value of BIT and EBIT segment location counters will increase by the value indicated by the operand.

- DB, DW directive

  The value of CODE and EDATA segment location counters will increase by the total number of words in the operands.

- ORG directive

  The value of segment location counters will become the value of the operand.

4.6.3 Referring Location Counter Values

By using the dollar sign ($), a source statement can refer the value of the current location counter of the logical segment in which it resides. The dollar sign ($) is called the location counter symbol. For details, refer to Section 4.8.4, “Location Counter Symbol,” and Section 4.8.5.2, “Usage Types and Physical Segment Attributes.”

4.7 Conditional Assembly and Macros

Conditional assembly and macros can raise program development efficiency and make programs easier to read and maintain. This section explains the use of conditional assembly and macros.

4.7.1 Using Conditional Assembly

By using conditional assembly, you can control assembly such that blocks in a program are assembled only when particular conditions are met.

Below is a simple example.

■ **Example** ■

```
IF  SW==1
  BUFSIZE    EQU    200H
ELSE
  BUFSIZE    EQU    400H
ENDIF
```

In this example, if the value of SW is 1, then BUFSIZE will be set to 200H. Otherwise, BUFSIZE will be set to 400H.

As shown here, conditional assembly is realized by coding conditional assembly directives. Conditional assembly directives have the following syntax.

```
IFxxx conditional_operand
   true_conditional_body
ENDIF
```

or

```
IFxxx conditional_operand
   true_conditional_body
ELSE
   false_conditional_body
ENDIF
```

Here IF*xxx* represents one of the following conditional assembly directives.

IF   IFDEF   IFNDEF

The *conditional_operand* provides the true or false condition for conditional assembly. The specification for *conditional_operand* differs depending on the conditional assembly directive.

If the condition is true, then the statement block of *true_conditional_body* will be assembled. If the condition is false, then the statement block of *true_conditional_body* will be skipped. If there is an ELSE directive in this case, then the *false_conditional_body* will be assembled. Conditional assembly directives can be nested up to 15 levels.

The next sections explain the syntax of conditional assembly directives and how they determine if conditions are true or false.

4.7.1.1 IF Directive

■ Syntax ■

```
    IF constant_expression
```

■ Description ■

The *constant_expression* is a constant expression that does not include forward references.

The condition will be true if *constant_expression* evaluates to a value other than 0. The condition will be false if *constant_expression* evaluates to 0.

■ Example ■

```
SW  EQU   1
IF  SW==1
INCLUDE  (SYMDEF1.INC)
ELSE
INCLUDE  (SYMDEF2.INC)
ENDIF
```

In this example, the value of SW is 1, so the condition will be true. Therefore SYMDEF1.INC will be included.

4.7.1.2 IFDEF Directive

■ Syntax ■

```
    IFDEF symbol
```

■ Description ■

The *symbol* is any symbol other than a reserved word.

The condition will be true if *symbol* is a symbol that has already been defined. The condition will be false if *symbol* was not defined before this IFDEF directive was encountered.

■ Example ■

```
DEFSYM EQU   1

IFDEF  DEFSYM
 INCLUDE  (SYMDEF1.INC)
ELSE
 INCLUDE  (SYMDEF2.INC)
ENDIF
```

In this example, DEFSYM was previously defined, so the condition will be true. Therefore, SYMDEF1.INC will be included.

4.7.1.3 IFNDEF Directive

■ Syntax ■

```
   IFNDEF symbol
```

■ Description ■

The *symbol* is any symbol other than a reserved word.

The condition will be true if *symbol* was not defined before this IFNDEF directive was encountered. The condition will be false if *symbol* is a symbol that has already been defined.

■ Example ■

```
DEFSYM EQU   1

IFNDEF  DEFSYM
 INCLUDE  (SYMDEF1.INC)
ELSE
 INCLUDE  (SYMDEF2.INC)
ENDIF
```

In this example, DEFSYM was previously defined, so the condition will be false. Therefore, SYMDEF2.INC will be included.

4.7.2 Using Macros

Macros assign text strings to symbol names. Frequently used repetitive code can be programmed more easily through the use of macros.

■ Syntax ■

```
DEFINE symbol  “macro_body”
```

The *symbol* specifies the macro symbol being defined. The *macro_body* specifies the string to be assigned to *symbol*.

A macro can be used after the DEFINE directive statement that defines it. When RAS66K encounters the macro *symbol*, it replaces the macro with the original *macro_body* string before assembling.

■ Example ■

```
DEFINE LA     "L A,"
DEFINE RWSEG  "SEGMENT CODE WINDOW"
LA    ER0
LA    [X1]
SEG1   RWSEG
```

In this example, “L A” is assigned to LA, and “SEGMENT CODE ROMWINDOW” is assigned to RWSEG.

Separate macros can be coded in *macro_body*. This is called *macro nesting*. Macros can be nested up to 8 levels.

■ Information ■

The RAS66K macro function is very simple. RAS66K cannot define macros of multiple statements or macros with parameters. If you want to use high-level macro functions, you should use the macroprocessor MP.

Refer to the Macroprocessor MP User’s Manual for more information about the macroprocessor MP.

4.8 Program Elements

Program elements are character set, constants, symbols, and location counter symbol that RAS66K uses in programs.

4.8.1 Character Set

The following types of characters can be used in programs.

- Letters, digits, underscore, question mark, dollar sign
- White space
- Line feed code, carriage return code
- Special characters
- Operators
- Escape sequences

However, character constants, string constants, and comments can use all characters expressible with 1-byte codes (00H-0FFH).

4.8.1.1 Letters, Digits, Underscore, Question Mark, Dollar Sign

Upper-case and lower-case letters, decimal Arabic digits, the underscore (_), and question mark (?) , and dollar sign ($) can be used in programs. These are listed below.

	Usable characters
Upper-case letters	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Lower-case letters	abcdefghijklmnopqrstuvwxyz
Decimal digits	0123456789
Underscore	_
Question mark	?
Dollar sign	$

4.8.1.2 White Space

Spaces (20H) and tabs (09H) have the role of delimiting adjacent elements in source statements. These are called white space. A string of white space characters has the same meaning as a single white space character.

4.8.1.3 Line Feed Code, Carriage Return Code

A line feed code (0AH) indicates the end of a source statement. Carriage return codes (0DH) have no syntactical meaning. RAS66K skips over carriage return code.

4.8.1.4 Special Characters

Special characters are characters that give special meaning to the elements that precede or follow them. Special characters are listed below.

Table 4-11. Special Characters

Character	Description
#	Specifies immediate addressing.
	Specifies physical segment address.
$	Specifies location counter symbol.
,	Used for operator delimiters.
:	Specifies a label.
	Used to delimit usage type and symbol in EXTRN and COMM directives.
	Used to delimit physical segment addresses and offset addresses in address expressions.
;	Starts a comment.
[ ]	Specifies indirect addressing.
'	Specifies a character constant.
"	Specifies a string constant.
\	Specifies current page addressing.

4.8.1.5 Operators

Operators are specified by either single characters or groups of characters. Refer to Section 4.9.2, “Operators,” for the function of each operator. Operators are listed below.

```
(       )       .       !       ~
+       -       *       /       %
<<      >>
<       <=      >       >=      ==      !=
&       ^       |       &&      ||
HIGH    MID     LOW     SEG     OFFSET
PAGE    LREG    BPOS    SIZE
```

4.8.1.6 Escape Sequences

Escape sequences can be used in character constants and string constants. An escape sequence is a backslash (\) followed by a character or digits. The types of escape sequences are listed below.

Table 4-12. Escape Sequences

Syntax	Description
nnn	The *nnn* indicates an octal number of one to three digits. The escape sequence will be replaced by the value of this octal number. The value of the octal number must be within the range 0-255.
\x*nn*, \X*nn*	The *nn* indicates a hexadecimal number of one or two digits. The escape sequence will be replaced by the value of this hexadecimal number. The value of the hexadecimal number must be within the range 0-255.
\a	This is converted to 07H.
\b	This is converted to 08H.
\f	This is converted to 0CH.
\n	This is converted to 0AH.
\r	This is converted to 0DH.
\t	This is converted to 09H.
\v	This is converted to 0BH.
\char	The *char* indicates an ASCII character. other than a, b, f, n, r, t, and v. The escape code will be replaced by the 1-byte code corresponding to this character.

■ Examples ■

Examples of escape sequences are shown below. The value that each escape sequence will become is shown as a hexadecimal number.

Escape sequence	Value
`\0`	`00H`
`\47`	`27H`
`\377`	`0FFH`
`\8`	`38H`
`\047`	`27H`
`\x0`	`00H`
`\xA`	`0AH`
`\xFF`	`0FFH`
`\x0F`	`0FH`
`\F`	`46H`
`\a`	`07H`
`\n`	`0AH`

4.8.2 Constants

Constants are numbers, characters, and strings that are used as a fixed value in the program. RAS66K recognizes the following constants.

- Integer constants
- Address constants
- Character constants
- String constants

4.8.2.1 Integer Constants

■ Syntax ■

```
ddigits
hdigitsH
ddigitsD
odigitsO
odigitsQ
bdigitsB
```

■ Description ■

Integer constants are integers expressible by 32 bits. Binary, octal, decimal, and hexadecimal numbers can be used as integer constants. The radix is specified by appending a radix specifier to the number. If the radix specifier is omitted, then the number will be taken as decimal.

The *hdigits*, *ddigits*, *odigits*, and *bdigits* code hexadecimal, decimal, octal, and binary, respectively. In order to distinguish them from symbols, integer constants must have a digit 0-9 as their first character. Thus, if the first character of a hexadecimal number would be a letter, then it must be prefixed with the digit 0.

To make programs easier to read, underscores (_) can be used within strings that express numbers. However, an underscore (_) cannot be the first character of an integer constant.

The characters that can be used in constants of each radix and the radix specifiers are listed below.

Table 4-13. Radix Specifiers

Radix Specifier	Description	Usable Characters
H, h	Hexadecimal	0123456789ABCDEFabcdef_
D, d	Decimal	0123456789_
O, o, Q, q	Octal	01234567_
B, b	Binary	01_

Either upper-case or lower-case letters can be used as radix specifiers. Also, both upper-case and lower-case letters can be used in hexadecimal numbers.

■ Examples ■

The decimal number 256 specified in hexadecimal, decimal, octal, and binary are coded as below.

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

Also, the meaning of the integer constant will not change even if several digits 0 are prefixed to it. The following code also show decimal 256.

	Description
Hexadecimal	00100H
Decimal	0256 00256D
Octal	000400O 000400Q
Binary	000100000000B

Shown below are some examples of decimal 256 using underscores.

	Description
Hexadecimal	1_00H 1_00_H
Binary	1_0000_0000B 1_0000_0000_B

4.8.2.2 Address Constants

■ Syntax ■

```
    integer_constant1:integer_constant2
```

■ Description ■

Address constants directly express addresses in address space.

Address constants are expressed with two fields: a physical segment address and an offset address. The *integer_constant1* is an integer expression that indicates the physical segment address. Its value must be 0-0FFH. The *integer_constant2* is an integer expression that indicates the offset address. Its value must be 0-7FFFFH.

The *integer_constant1* and *integer_constant2* are delimited by a colon (:). Spaces and tabs cannot be inserted before or after the colon (:).

An address expression itself does not express the type of address space. RAS66K determines the address space for the address expression based on the instruction or directive of the source statement in which it is coded.

■ Example ■

The example below expresses an address in DATA address space at physical segment address 2 and offset address 1000H.

```
    MOV     ER0,2:1000H
D_ADR DATA    2:1000H
```

The example below expresses an address in CODE address space at physical segment address 2 and offset address 1000H.

```
    FJ      2:1000H
C_ADR CODE    2:1000H
```

4.8.2.3 Character Constants

■ Syntax ■

```
'char'
```

■ Description ■

Character constants are converted to the 1-byte codes of their specified characters. The *char* specifies a character expressed by a 1-byte code. It can also be an escape sequence that expresses a 1-byte code. If the character or escape sequence is omitted, then the character constant’s value will be 0H.

■ Examples ■

Examples of character constants are shown below. The value represented by each character constant is shown by a hexadecimal integer constant.

Character Constant	Value
'	00H
'A'	41H
'\0'	00H
'\47'	27H
'\377'	0FFH
'\8'	38H
'\047'	27H
'\x0'	00H
'\xA'	0AH
'\xFF'	0FFH
'\x0F'	0FH
'\F'	46H
'\''	27H

4.8.2.4 String Constants

■ Syntax ■

```
"characters"
```

■ Description ■

String constants are strings enclosed in double quotation marks (“) that are used to initialize code memory. The *characters* specify a string. All escape sequences and all characters expressed by 1-byte codes can be coded within a string. Strings must be expressible with 256 bytes of code or fewer.

■ Examples ■

Some examples of string constants used as operands of DB and DW directives are shown below. The code values are shown as hexadecimal integer constants in the comments.

```
DB "STRING"     ;53H, 54H, 52H, 49H, 4EH, 47H
DB "\111\222"   ;49H, 92H
DB "\x10\XFF"   ;10H, 0FFH
```

4.8.3 Symbols

Symbols are names that represent the following items.

- Numbers
- Addresses
- Relocatable segments
- Instructions
- Directives
- Registers
- Register addresses
- Operators
- Addressing types
- Special operands of instructions
- Special operands of directives
- Macros

Symbols are either symbols defined in a program or symbols already provided by RAS66K. Symbols defined in a program are called *user symbols*, and symbols provided by RAS66K are called *reserved words*. Symbols representing numbers and addresses include both user symbols and reserved words. Symbols that represent relocatable segments and macros are all user symbols. Other symbols are all reserved words.

Each symbol is a string of 1 to 32 characters consisting of letters, digits, underscores, question marks, and dollar signs. The first character must be a letter, underscore, question mark, or dollar sign. If a symbol exceeding 32 characters is coded, then all characters beyond 32 will be ignored. Unless it is defined using a SET directive, a symbol can be defined only once within a source file. If a symbol is defined twice or more within a source file, then an error will occur.

4.8.3.1 User Symbols

User symbols are symbols defined by the user within the program. Reserved words cannot be defined as user symbols.

Whether or not RAS66K will distinguish between upper-case and lower-case letters that make up user symbols can be controlled with the /CD and /NCD option. If the /CD option is specified in a program, then RAS66K will distinguish between upper-case and lower-case letters. If the /NCD option is specified, then RAS66K will not distinguish between upper-case and lower-case letters. RAS66K will not distinguish between upper-case and lower-case letters by default. However, if the /CC option is specified, then RAS66K will distinguish between upper-case and lower-case letters by default.

User symbols can be defined as labels or defined using directives. The methods for defining each type of user symbol are shown below.

Table 4-14. Defining User Symbols

User Symbol	Symbol Type or Directive To Define
User symbols representing numbers	EQU directive, SET directive, EXTRN directive
User symbols representing addresses	Label, EQU directive, SET directive, CODE directive, CBIT directive, DATA directive, BIT directive, EDATA directive, EBIT directive, COMM directive, EXTRN directive
User symbols representing relocatable segments	SEGMENT directive
User symbols representing macros	DEFINE directive

Symbols that represent macros correspond to text strings. Any user symbol other than a macro has a value. This value is assigned when the symbol is defined. Symbols that represent numbers take those numbers as values. Symbols that represent addresses take those addresses as values. Symbols that represent relocatable segments take the first address of the areas in which the relocatable segments are allocated as values.

■ Example 1 ■

Examples of definitions of user symbols that represent numbers are shown below. In this example, SYMEQU1, SYMSET1, and SYM_EXT_NUM1 will become user symbols that represent numbers.

```
SYMEQU1 EQU 0FFH
SYMSET1 SET 100H
EXTRN   NUMBER:SYM_EXT_NUM1
```

■ Example 2 ■

Examples of definitions of user symbols that represent addresses are shown below.

```
EXTINT0 CODE    3H
EXTINT1 EQU     EXTINT0+1
SYMDAT1 DATA    80H
SYMBIT1 BIT     80H.0
SYMSET2 SET     10H+SYMDAT1
SYM_COMM_DAT1   COMM  DATA 2
EXTRN   BIT:SYM_EXT_BIT1
```

In this example, EXTINT0, EXTINT1, SYMDAT1, SYMBIT1, SYMSET2, SYM_COMM_DAT1, and SYM_EXT_BIT1 will become user symbols that represent numbers.

■ Example 3 ■

Examples of definitions of user symbols that represent relocatable segments are shown below. In this example, MAINCOD, TABLE1, and COMBUF1 will become user symbols that represent relocatable segments.

```
MAINCOD SEGMENT CODE #0
     RSEG    MAINCOD
     .
     .
     .

TABLE1  SEGMENT CODE #1
     RSEG    TABLE1
     DW      0000H
     DW      0001H
     .
     .
     .

COMBUF1 SEGMENT DATA 2 COMMON
     RSEG    COMBUF1
     DS    2
```

■ Example 4 ■

An example of a user symbol that represents a macro is shown below. In this example, MCRSYM will become a user symbol that represents a macro.

```
DEFINE MCRSYM “MACROBODY”
```

The following explanation classifies user symbols into different types. The reason for this is that the symbol type determines where in a program a symbol can be used. The following classifications do not include symbols representing macros.

The types of user symbols are given below.

- Absolute symbols
- Relocatable symbols

Absolute symbols are symbols for which RAS66K can determine a value. Relocatable symbols are symbols for which RAS66K cannot determine a value. RL66K determines the values of relocatable symbols.

(1) Absolute Symbols

Absolute symbols include absolute symbols that represent numbers and absolute symbols that represent addresses. An absolute symbol is defined in one of the following ways.

- Define the symbol by coding a constant expression as the operand of a local symbol definition directive (EQU, SET, CODE, CBIT, DATA, BIT, EDATA, EBIT).

- Define a label residing in an absolute segment.

■ Example ■

An example of absolute symbol definitions is shown below. In this example, SYMCOD, SYMDAT, SYMBIT, SYMEQU, SYMSET, DATALABEL, and CODELABEL will be absolute symbols.

```
SYMCOD     CODE    100H
SYMDAT     DATA    80H
SYMBIT     BIT     80H.0
SYMEQU     EQU     (-1)
SYMSET     SET     10H+SYMDAT

      DSEG #0 AT 280H
DATALABEL:
      DS   2

      CSEG #0 AT 100H
CODELABEL:
      NOP
```

(2) Relocatable Symbols

Relocatable symbols include the following types.

- Simple relocatable symbols
- Segment symbols
- External symbols
- Communal symbols

Each of these types of relocatable symbols is explained below.

- **Simple relocatable symbols**

Simple relocatable symbols are symbols that represent addresses of relocatable segments in the same source program. Symbols that represent relocatable segments (segment symbols) are not simple relocatable symbols. A simple relocatable symbol is defined in one of the following ways.

  - Define a label residing in a relocatable segment.

- Define the symbol by coding an expression that uses a simple relocatable symbol as the operand of a local symbol definition directive (EQU, SET, CODE, CBIT, DATA, BIT, EDATA, EBIT).

Simple relocatable symbols reside in relocatable segments. The relocatable segment in which a simple relocatable symbol resides is determined in one of the following ways.

- For a label defined in a relocatable segment, the simple relocatable symbol will reside in that relocatable segment.

- For a simple relocatable symbol defined using a local symbol definition directive (EQU, SET, CODE, CBIT, DATA, BIT, EDATA, EBIT), the simple relocatable symbol will reside in the same segment as the simple relocatable segment specified in the operand.

■ Example ■

An example of simple relocatable symbol definitions is shown below.

```
DATSEG  SEGMENT DATA
     RSEG    DATSEG
LBUF:
     DS     1
HBUF:
     DS     1

CODSEG  SEGMENT CODE
     RSEG    CODSEG
START:
     NOP

SIMCOD  CODE    START+1
SIMDAT  DATA    LBUF+2
SIMBIT  BIT     (LBUF+2).0
SIMEQU  EQU     HBUF+2
SIMSET  SET     LBUF+4
```

In this example, LBUF, HBUF, START, SIMCOD, SIMDAT, SIMBIT, SIMEQU, and SIMSET will become simple relocatable symbols. DATASEG and CODESEG are symbols that represent relocatable segments; they are not simple relocatable symbols.

- **Local Symbols And Public Symbols**

Unless some declaration is made for an absolute symbol or simple relocatable symbol, then that symbol can be referred only within the source file that defines it. Such absolute symbols and simple relocatable symbols are called local symbols.

To refer a local symbol from another source file, the local symbol must be declared public using a PUBLIC directive. Local symbols declared public are called public symbols.

The fact that a local symbol can be referred only from within the source file that defines it is very important. In general, each source file will have some independent functions, so each source file will be created separately. If a local symbol with the same name is used in another source file but is not handled as a separate symbol, then there may be problems with symbol name management.

■ **Example** ■

The following example shows an example of declaring a public symbol.

```
PUBLIC  SYMEQU  DATABUF1        ;A public symbol declaration

SYMEQU  EQU     1

DATSEG2 SEGMENT DATA
RSEG    DATSEG2
DATABUF1:
    DS    2
```

In this example, SYMEQU is an absolute symbol and DATABUF1 is a simple relocatable symbol. Accordingly, these two symbols are local symbols. This example declares local symbols SYMEQU and DATABUF1 to be public using the PUBLIC directive.

- **Segment Symbols**

Segment symbols are symbols that represent relocatable segments. Segment symbols are defined using the SEGMENT directive. If a segment symbol is coded as the operand of an RSEG directive, then a relocatable segment with the name of that segment symbol will be defined. This means that a segment symbol may be a segment name.

Segment symbols take as values the first address of the area in which the relocatable segment is placed. To refer a segment symbol of another source file, one must define that segment symbol using the SEGMENT directive.

■ **Example** ■

```
WORKDAT SEGMENT DATA COMMON
CHARBUF SEGMENT DATA #2
SUB1    SEGMENT CODE
```

In this example, the segment symbols WORKDAT, CHARBUF, and SUB1 are defined.

- **External Symbols**

Public symbols and communal symbols of other source files can be referred by using external symbols. External symbols are declared using the EXTRN directive.

■ Example ■

The following example shows external symbol declaration

```
EXTRN DATA:EXTSYM1 EXTSYM2
EXTRN NUMBER:EXTSYM3 CODE:EXTSYM4
```

In this example, EXTSYM1 and EXTSYM2 are external symbols that represent data addresses. EXTSYM3 is an external symbol that represents a number. EXTSYM4 is an external symbol that represents a code address.

- **Communal Symbols**

A communal symbol represents the first address of a data area common to multiple source files.

If a communal symbol with the same name has been declared in other source files, then RL66K will allocate memory based on the maximum size of all declarations of communal symbols with the same name. The communal symbol will represent the first address of that area.

If no communal symbol with the same name has been declared in other source files, then RL66K will allocate memory based on the specified area size at the time of the communal symbol's declaration. The communal symbol will represent the first address of that area.

Communal symbols are declared using the COMM directive.

■ Example ■

A communal symbol declaration example is shown below.

```
COMMSYM  COMM  DATA 10H
```

In this example, COMMSYM is a communal symbol allocated to DATA address space. The communal symbol's size specification is 10H bytes.

(3) Referring User Symbols

The value of a defined user symbol can be referred from operands of instructions and directives in the same source file.

There are two types of user symbol references.

- Backward references

  The referred symbol is defined in the source file code before the point of reference.

- Forward references

  The referred symbol is defined in the source file code after the point of reference.

When a user symbol is referred in a microcontroller instruction operand, both backward and forward references are permitted.

When a user symbol is referred in a directive operand, backward references are always permitted, but depending on the directive, forward references might not be permitted. Refer to Section 4.12, "Directives," for directive operands that do not permit forward references.

■ Example ■

In this example, both backward and forward referred symbols are coded as operands of two micro-controller instructions (J, MOV) and two directives (DW, ORG).

```
BACKWARD_VALUE  EQU 10H
BACKWARD_SYM    EQU 100H
BACKWARD_LABEL:
        J   FORWARD_LABEL
        MOV ER0,#FORWARD_SYM
        DW  FORWARD_VALUE
        ORG FORWARD_VALUE        ;error


        ORG BACKWARD_VALUE
        DW  BACKWARD_VALUE
        MOV ER0,BACKWARD_SYM
        J   BACKWARD_LABEL
FORWARD_LABEL:
FORWARD_SYM     EQU 200H
FORWARD_VALUE   EQU 20H
```

The backward referenced symbols are BACKWARD_VALUE, BACKWARD_SYM, and BACK-WARD_LABEL. The forward referenced symbols are FORWARD_VALUE, FORWARD_SYM, and FORWARD_LABEL. Both backward and forward references are permitted in the operands of microcontroller instructions and DW directives. However, forward references are not permitted in the operands of ORG directives. Accordingly, the source statement "ORG FORWARD_VALUE" will cause an error.

(4) Referring User Symbols From Multiple Source Files

To refer the same user symbol from multiple source files, use the following types of symbols.

- Public symbols
- External symbols
- Communal symbols
- Segment symbols

Refer to Section 4.12.13, “Creating Programs From Multiple Source Files,” for methods of using these symbols.

(5) Macro Symbols

Macro symbols are unusual compared to other user symbols. Ordinary user symbols are assigned values, but macro symbols are assigned text strings. Macro symbols cannot be declared public to refer them in source files other than whichever source file defined them.

The greatest feature of macro symbols is not the macro symbol itself, but that its assigned text string has meaning in assembly language. That is why this manual clearly distinguishes between macro symbols and other user symbols. When this manual mentions “user symbols,” it is nearly always referring to symbols other than macro symbols.

4.8.3.2 Reserved Words

Reserved words are symbols already provided by RAS66K. Reversed words include the following types.

- Instructions
- Directives
- Registers
- Operators
- Fixed microcontroller addresses
- Local register addresses
- Pointing register addresses
- Addressing specifiers
- Special operands of instructions
- Special operands of directives

Case sensitivity of reserved words that represent fixed microcontroller addresses can be controlled using the /CD and /NCD options, the same as for user symbols. There is no case sensitivity for all other reserved words.

Some symbols among the reserved words have multiple functions. Also, some reserved words are specific only to microcontrollers with a particular CPU core. These reserved words are freed when using other CPU cores, and can be used as user symbols.

Appendix B lists all reserved words (except those representing fixed microcontroller addresses), their functions, and their applicable CPU cores.

Each type of reserved word is explained below.

(1) Instructions

These are microcontroller instructions that RAS66K can assemble. Refer to related documentation for instruction functions.

(2) Directives

These are directives provided by RAS66K. Refer to Section 4.12, "Directives," regarding directive functions.

(3) Registers

These are symbols that represent registers. They are used as operands of microcontroller instructions. Reserved words that represent register addresses differ from those that express the registers themselves. Refer to Section 4.11, "Addressing Modes," for how to write operands using registers. Here is a list of reserved words that represent registers.

```
R0      R1      R2      R3
R4      R5      R6      R7
ER0     ER1     ER2     ER3
X1      X2      DP      USP
X1L     X2L     DPL     USPL
LRB     PSW     PSWH    PSWL    SSP
A       C       PR      ER      CR
```

(4) Operators

These are symbols that represent operators. They are used to code expressions. Refer to Section 4.9.2, "Operators," regarding the use and functions of operators. Here is a list of reserved words representing operators.

```
HIGH    LOW     MID     SEG     SIZE
OFFSET  PAGE    LREG    BPOS
```

(5) Fixed microcontroller addresses

These are symbols that have values of fixed addresses of the target microcontroller. They apply to register names and register bit names of the SFR areas. RAS66K assembles these reserved words as absolute symbols. These reserved words are defined in the DCL file. Refer to the DCL66K.DOC file for reserved words that represent addresses.

Reserved words that represent fixed microcontroller addresses are different from other reserved words in that they are affected by the /CD and /NCD options. By default, or if /NCD is specified, symbols with the same spelling will be assembled as the same symbol even if they have different combinations of upper-case and lower-case letters. If the /CD option is specified, then symbols with the same spelling will be assembled as the same symbol only when they have the same combination of upper-case and lower-case letters.

(6) Local register addresses

These are symbols that represent addresses in DATA memory space allocated to local registers. Here is a list of symbols that represent local register addresses.

```
AR0     AR1     AR2     AR3
AR4     AR5     AR6     AR7
AER0    AER1    AER2    AER3
```

These symbols can be used only when the CPU core is nX-8/500. Local register addresses change according to the value set in LRBL. To change the value of LRBL, you need to code a USING LREG directive. RAS66K calculates local register addresses from the bank number of the local register set specified with the USING LREG directive, and then resets the values of these symbols.

(7) Pointing register addresses

These are symbols that represent addresses in DATA memory space allocated to pointing registers. Here is a list of symbols that represent pointing register addresses.

```
AX1     AX2     ADP     AUSP
```

Pointing register addresses change according to the value set in SCB of the PSW. To change the value of SCB, you need to code a USING PREG directive. RAS66K calculates pointing register addresses from the bank number of the pointing register set specified with the USING PREG directive, and then resets the values of these symbols.

(8) Addressing specifiers

These are symbols that represent types of addressing. Refer to Section 4.11, “Addressing Modes,” for how to write operands using addressing specifiers. Here is a list of reserved words that represent addressing specifiers.

```
DIR     OFF     SFR     FIX
SBAOFF  SBAFIX
```

(9) Special operands of instructions

These are symbols with special meanings as instruction operands, such as flag names and branch conditions of conditional jump instructions. Refer to related documentation for their actual meaning and uses. Here is a list of reserved words that represent special operands of instructions.

```
GT      GE      NC      EQ      NE
NZ      LE      LT      PS      NS
LTS     LES     GTS     GES     NV
ZF      CY      OV      NULL
```

(10) Special operands of directives

These are symbols with special meanings as instruction operands. Refer to Section 4.12, “Directives,” for their actual meaning and uses.

4.8.4 Location Counter Symbol

■ Syntax ■

```
    $
```

■ Description ■

RAS66K always stores the current address of the logical segment being assembled. The counter that stores this address is called the location counter. The value of the location counter in the current logical segment can be referred by using the dollar sign ($). The dollar sign ($) is called the location counter symbol.

When the location counter symbol is used in source statements in absolute segments, RAS66K will handle the location counter symbol as an absolute symbol.

When the location counter symbol is used in source statements in relocatable segments, RAS66K will handle the location counter symbol as a simple relocatable symbol.

■ Example ■

The example below shows use of the location counter symbol.

```
CODSEG       SEGMENT CODE
       RSEG    CODSEG
MES:       DB      "help message"
MES_LENGTH  EQU     $-MES
```

In this example, the symbol MES_LENGTH will be equal to the length of the string defined from the address MES.

4.8.5 Value Attributes

RAS66K program elements that express values have several attributes. Program elements that express values include constants and symbols themselves, as well as expressions that combine constants and symbols.

The programmer usually does not have to be concerned with the value attributes explained here while programming. However, in some cases these attributes fulfill a critical role in programming. For example, RAS66K makes use of value attributes in its addressing checks and addressing optimizations.

4.8.5.1 Numeric Values and Address Values

Values used in programs are either simple values or addresses in memory space. Simple values are called *numeric values*, and addresses in memory space are called *address values*.

A numeric values is expressed in 32 bits. An address value consists of a physical segment address and an offset address. A physical segment address is expressed in 8 bits, and an offset address is expressed in 32 bits.

■ **Example** ■

The example below defines numeric value symbols and address value symbols using EQU, CODE, and DATA directives.

```
NUMB_SYM  EQU  1000H
CODE_SYM  CODE 1000H
DATA_SYM  DATA 1000H
```

In this example, NUMB_SYM is assigned 1000H as its value. Accordingly, the value of NUMB_SYM can be called a number. CODE_SYM is assigned address 1000H in CODE address space, and DATA_SYM is assigned address 1000H in DATA address space. Accordingly the values of CODE_SYM and DATA_SYM can be called addresses.

4.8.5.2 Usage Types and Physical Segment Attributes

(1) Usage Types

The usage type is an attribute that indicates the purpose for using a value. Usage types are listed below.

Table 4-15. Usage Types

Usage Type	Meaning
NUMBER	Element holds a value.
CODE	Element holds an address in CODE address space.
CBIT	Element holds an address in CBIT address space.
DATA	Element holds an address in DATA address space.
BIT	Element holds an address in BIT address space.
EDATA	Element holds an address in EDATA address space.
EBIT	Element holds an address in EBIT address space.
NONE	Element holds an address, but in which address space is not determined.

The usage type NUMBER indicates a value, so saying a value's usage type is NUMBER is completely equivalent to saying a value is a numeric value.

The CODE, DATA, BIT, EDATA, and EBIT usage types are called *segment types*. In this manual, the types of address spaces in which addresses reside are called segment types. The purposes for using values and addresses are called *usage types*.

Usage types have the critical role of protecting addressing. Values can be coded in the operands of many instructions and directives, but the type of value that should be coded is predetermined. RAS66K performs checks comparing the usage types that should be used in operands and the usage types actually used. For details refer to Section 4.10.3, “Usage Type Checks.”

(2) Physical Segment Attributes

The physical segment attribute indicates the state of the physical segment of an address value.

This manual first described address values as having physical segment addresses, but there are several exceptions. For one, values that represent COMMON area addresses are certainly addresses, but they do not have a physical segment address. For another, the physical segment address of many relocatable symbols is not known during assembly. The attribute that expresses these states of physical segments of address values is the physical segment attribute.

Physical segment attribute types are listed below.

Table 4-16. Physical Segment Attribute Types

Physical Segment Attribute	Meaning
Fixed	The physical segment address is fixed.
COMMON	Address is in COMMON area.
ANY	The physical segment address is not fixed.
None	There is no information about the physical segment.

RAS66K uses physical segment attributes for addressing checks on physical segment registers and code generated from jump and call instructions between physical segments.

The following explanation describes the usage type and physical segment address of each program element that represents a value.

• Attributes of Integer Constants and Character Constants

Integer constants and character constants represent numeric values, so their usage type is NUMBER, and they have no physical segment.

■ Example ■

```
100H
'A'
'\0'
```

These integer constants and character constants have usage type NUMBER.

• Attributes of Address Constants

Address constants express addresses but their address space is not fixed, so their usage type is NONE. Address constants include physical segment addresses, so their physical segment address is fixed.

■ Example ■

```
2:1000H
4:200H
```

These address constants have usage type NONE. The first is fixed in physical segment address 2, and the second is fixed in physical segment address 4.

• Attributes of User Symbols

The usage types and physical segment attributes of user symbols differ depending on how the symbols are defined. The relation between user symbol definition and usage type is shown below.

Table 4-17. User Symbol Definitions and Usage Types

Symbol Definition Method	Usage Type
Label	Will be segment type of logical segment in which label resides.
EQU directive	Inherits usage type of expression of operand.
SET directive	Inherits usage type of expression of operand.
CODE directive	Will be usage type CODE.
CBIT directive	Will be usage type CBIT.
DATA directive	Will be usage type DATA.
BIT directive	Will be usage type BIT.
EDATA directive	Will be usage type EDATA.
EBIT directive	Will be usage type EBIT.
SEGMENT directive	Will be usage type specified in operand of directive.
COMM directive	Will be usage type specified in operand of directive.
EXTRN directive	Will be usage type specified in operand of directive.

■ Example ■

```
        DSEG
SYM1:
SYM2    EQU     1
SYM3    CODE    1000H
SYM4    EQU     SYM3+10H
SYM5    SEGMENT DATA
SYM6    COMM    BIT  4
    EXTRN   NUMBER:SYM7
```

The usage types of symbols defined in this example are explained next.

The label SYM1 is coded immediately after the DATA segment is selected with a DSEG directive, so its usage type will be DATA, which is the segment type of the segment in which it resides. SYM2 is defined by an EQU directive with a numeric value as the operand, so its usage type will be NUMBER. SYM3 is defined by a CODE directive, so its usage type will be CODE. SYM4 is defined by an EQU directive with SYM3 (usage type: CODE) as the operand, so it will inherit the usage type CODE. SYM5 is specified by a SEGMENT directive with DATA as the operand, so its usage type will be DATA. SYM6 is specified by a COMM directive with BIT as the operand, so its usage type will be BIT. SYM7 is specified after an EXTRN directive with NUMBER as the operand, so its usage type will be NUMBER.

The relation between user symbol definition and physical segment attribute is shown below.

Table 4-18. User Symbol Definitions and Physical Segment Attributes

Symbol Definition Method	Physical Segment Attribute
Label	Inherits physical segment attribute of physical segment in which the label resides.
EQU directive SET directive	No physical segment attribute if operand is numeric value. Inherits physical segment attribute of operand if operand is address value.
CODE directive CBIT directive DATA directive BIT directive	Fixed to 0 if operand is numeric value. Inherits physical segment attribute of operand if operand is address value.
EDATA directive EBIT directive	Will be COMMON if EEPROM area is included in the COMMON area. Fixed to 0 if EEPROM area is included in a separate area.
SEGMENT directive COMM directive	Will be physical segment attribute specified in operand of directive.
EXTRN directive	No physical segment attribute if usage type NUMBER is specified as the operand. Will be COMMON if usage type EDATA or EBIT is specified as the operand and the EEPROM area is included in the COMMON area. Fixed to 0 if usage type EDATA or EBIT is specified as the operand and the EEPROM area is included in a separate area. Will be ANY for all other cases.

■ Example ■

The program below defines segment symbols as an example of physical segment attributes.

```
SYM1    SEGMENT DATA  #1
SYM2    SEGMENT DATA  COMMON
SYM3    SEGMENT DATA
```

In this example, SYM1 is specified with #1 in the operand, so its physical segment address will be fixed to 1. SYM2 is specified with COMMON in the operand, so its physical segment attribute will be COMMON. SYM3 is has no physical segment information in its operand, so its physical segment attribute will be ANY.

• Attributes of Reserved Words Representing Fixed Microcontroller Addresses

These are explained in the DCL66K.DOC file.

• Attributes of Reserved Words Representing Local Register Addresses

Local registers are mapped to physical segment 0 in data memory space. Therefore, reserved words that represent local register addresses will have usage type DATA and will be fixed at physical segment address 0.

■ **Example** ■

```
TYPE   (M66507)
USING  LREG  4
L     A,AER0
```

In this example, the AER0 coded as the second operand of the L instruction will have usage type data and physical segment address 0. The bank is specified as 4 by the USING LREG directive, so the offset address of AER0 will be 220H (200H+8*4+00H).

- Attributes of Reserved Words Representing Pointing Register Addresses

Pointing registers are mapped to physical segment 0 in data memory space. Therefore, reserved words that represent pointing register addresses will have usage type DATA and will be fixed at physical segment address 0.

■ **Example** ■

```
TYPE   (M66507)
USING  PREG  4
L     A,AX1
```

In this example, the AX1 coded as the second operand of the L instruction will have usage type data and physical segment address 0. The bank is specified as 2 by the USING PREG directive, so the offset address of AX1 will be 210H (200H+8*2+00H).

- Attributes of Location Counter Symbol

The location counter symbol is used as an operand in source statements. It will inherit its usage type and physical segment attribute from the physical segment of each source statement in which it resides.

■ **Example** ■

```
CSEG  #0
SJ    $          ;Infinite loop
```

In this example, the location counter symbol ($) has usage type CODE, which is the segment type of the physical segment of the source statement in which it resides. Its physical segment address is fixed to 0.

4.8.5.3 Flag Attributes

This section describes user symbol flag attributes after first explaining their application.

Many OLMS-66K Series microcontroller instructions are affected by the data descriptor (DD) or the stack flag (SF). Other instructions automatically change the state of these flags. It is very important to manage the states of these flags when creating an OLMS-66K Series application program.

Flag attributes indicate the states of DD and SF managed by RAS66K. Only microcontrollers with the nX-8/300 CPU core have the SF flag.

The critical application of flag attributes is in checking flag attributes of instructions. RAS66K saves flag states within a program and checks whether instructions are used with the appropriate flag states. To inform RAS66K of the current flag attributes, you must code USING directives. When the program specifies the state of DD, use the USING DATA directive. When the program specifies the state of SF, use the OPRT directive.

The meaning of each flag state and how to specify the USING directives are shown below.

Table 4-19. Data Descriptor (DD) States

State Value	Meaning	USING Directive Specification
WORD	DD value is 1.	USING DATA WORD
BYTE	DD value is 0.	USING DATA BYTE
ANY	Do not assume DD value.	USING DATA ANY

Table 4-20. Stack Flag (SF) States

State Value	Meaning	USING Directive Specification
STACK	SF value is 1.	USING OPRT STACK
A	SF value is 0.	USING OPRT A
ANY	Do not assume SF value.	USING OPRT ANY

The flag state of ANY can be thought of as meaning the RAS66K will not use its flag attribute check function. RAS66K sets the ANY state by default.

■ Example ■

```
    USING  DATA  WORD
    STB    A,R0              ;Warning
```

In this example, the USING DATA directive in the first line assumes that DD is in the word state. The next STB instruction cannot be used unless DD is in the byte state, so this statement will generate a warning.

Flag attributes have one more important application. RAS66K checks that the flag attributes of branch sources and branch destinations of branch instructions match. To do this, RAS66K assigns the flag attribute currently set by the USING directive to each user symbol when defined. In other words, for each branch instruction RAS66K checks whether or not the flag attribute of the source statement coding that instruction matches that of the branch destination (which is the symbol specified as the instruction operand).

■ **Example** ■

```
        USING  DATA  WORD
        J     NEXT
        .
        .
        .
        USING  DATA  BYTE
NEXT:
        .
        .
        .
```

In this example, the data descriptor is coded in the word state for the J instructions, but the data descriptor of branch destination NEXT is in the byte state. The flag states of the branch source and branch destination do not match, so a warning will be generated.

4.8.5.4 Addressing Attributes

When the target microcontroller has a nX-8/500 CPU core, RAS66K performs optimization processing for RAM addressing.

There are several types of addressing that directly refer the contents of addresses in data memory. Addressing specifiers are provided to specify the type addressing. If an addressing specifier is placed before an address when it is coded as an operand, then it will clarify the addressing type. However, if only a simple address is coded without an address specifier, then RAS66K will check the address value and select the addressing that makes the most efficient code.

Below is an example of RAM addressing optimization.

■ **Example** ■

```
        L      A,80H         ;SFR address, so handled as SFR 80H.
        L      A,200H        ;Fixed page area address, so handled as FIX 200H.
        L      A,1000H       ;Not a special area address, so handled as DIR 1000H.
```

Operand addresses are fixed in this example, so RAS66K will select the best addressing for each address value.

However, when an operand is coded in a relocatable segment, the address is not fixed, so RAS66K cannot select the addressing type. RAS66K solves this problem by giving relocatable symbols an attribute call the addressing attribute.

Addressing attribute types are shown below.

Table 4-21. Addressing Attributes

Attribute	Meaning
FIX	Address is in fixed page area.
SBA	Address is in SBA area.

Symbols with the FIX addressing attribute are optimized to fixed page addressing. Symbols with the SBA addressing attribute are optimized to SBA area addressing. Symbols with both the FIX and SBA addressing attributes are optimized to fixed page SBA area addressing.

The method for assigning addressing attributes to each type of relocatable symbol is shown below.

Table 4-22. Addressing Attribute Assignment To Relocatable Symbols

Symbol Type	Addressing Attribute Assignment
Simple relocatable symbol	If the segment in which the symbol resides has the special area attribute FIX, SBA, or SBAFIX, then that will automatically be assigned as the addressing attribute.
Segment symbol	Special area attribute FIX, SBA, or SBAFIX is specified in the SEG MENT directive.
Communal symbol	Special area attribute FIX, SBA, or SBAFIX is specified in the COMM directive.
External symbol	Special area attribute FIX, SBA, or SBAFIX is specified in the EXTRN directive.

The example below shows how relocatable symbols with the FIX addressing attribute are defined.

■ Example ■

```
FIX_SEG   SEGMENT DATA WORD FIX
FIX_COM   COMM    DATA 2 FIX
EXTRN     DATA    FIX : FIX_EXT
          RSEG    FIX_SEG
FIX_REL   DS      2

          CSEG
          L    A,FIX_SEG
          L    A,FIX_COM
          L    A,FIX_EXT
          L    A,FIX_REL
```

In this example, when the symbols FIX_SEG, FIX_COM, and FIX_EXT are defined, FIX is specified in the operand of each directive, so they will be given the FIX addressing attribute. FIX_REL resides in relocatable segment FIX_SEG, so it will also be given the FIX addressing attribute. The second operands of the four L instructions coded after CSEG will all have fixed page addressing.

4.9 Operators and Expressions

Expressions can be used in operands of instructions and directives. An expression consists of multiple constants and symbols connected by operators. RAS66K evaluates expressions coded in source statements and converts them to single values. Instead of coding values directly in a program, the programmer can code expressions that show the meaning of the values.

This section first explains the basic concepts of expressions in assembly language. It then describes the types and functions of the various operators, the types of value attributes of expressions, and the procedure that RAS66K uses to evaluate expressions.

4.9.1 Basic Concepts Of Expressions

4.9.1.1 Meaning Of Attributes Of Expressions

As mentioned in Section 4.8.5, “Value Attributes,” program elements that represent values, such as constants and symbols, have various attributes.

Just as for constants and symbols, RAS66K classifies expressions as numeric values and address values, and assigns usage type attributes that indicate the purpose of each value. The attributes of expression are determined by its types of operators and the types of constants and symbols that are operated on. In other words, RAS66K manages not only expression values, but expression meanings as well.

This explanation may seem too vague, but you can see that these concepts are simple and natural if you look as some actual examples.

■ Example 1 ■

```
MOV   ER0,TBL+2
```

In this example, assume that TBL is an address in DATA address space. The expression “TBL+2” is understood as the address two bytes from TBL. Therefore, “TBL+2” is an expression that represents an address in DATA address space.

■ Example 2 ■

```
MOV   ER0,#END_ADR-START_ADR
```

In this example, assume that START_ADR and END_ADR are start and end addresses of a table provided in DATA address space. START_ADR and END_ADR represent addresses, but the expression “END_ADR-START_ADR” is understood to be a value that indicates the size of the table.

■ Example 3 ■

```
        SB     D_ADR.4
```

In this example, assume D_ADR is an address in DATA address space. The expression "D_ADR.4" is understood to be the fourth bit of D_ADR. Therefore, "D_ADR.4" is an expression that represents an address in BIT address space.

From these examples, you can see two reasons for expressions having attributes.

(1) RAS66K observes whether or not expressions coded as operands of instructions and directives are used correctly.

(2) RAS66K observes whether or not the meanings of the expressions themselves are contradictory.

In particular (1) is very important. When an expression is specified as an operand of an instruction or directive that it should not be, RAS66K will display a warning to inform the programmer that the operand specification could be in error.

■ Example 4 ■

```
        MOV    ER0,CODE_ADR
```

In this example, assume that CODE_ADR is an address of code memory space. However, the second operand of a MOV instruction should be specified as an address in data memory space. Accordingly, the use of CODE_ADR is clearly incorrect. In such a case, RAS66K will issue a warning for this source statement.

■ Example 5 ■

```
        MOV    ER0,#D_ADR1+D_ADR2
```

In this example, there is no meaning in the expression results. Assume that D_ADR1 and D_ADR2 are both addresses in data memory space. The expression "D_ADR1+D_ADR2" adds two addresses, so the calculation itself has no particular meaning. The programmer might have consciously intended to code this, but there is a large probability that the usage is mistaken. In such a case, RAS66K will issue a warning.

The attributes of expressions and restrictions on how they are coded are important for assuring correct programs.

4.9.1.2 Using Physical Segment Addresses

Address expressions have physical segment addresses and offset addresses. When address expressions are calculated, most operators act only on the offset addresses, and not on the physical segment addresses.

■ Example 1 ■

```
DATA_SYM DATA    2:3000H+100H
```

In this example, the numeric value 100H is added to address 2:3000H. The result of this calculation is 2:3100H. In other words, for the left address term, the offset address 3000H is the object of the addition, but the physical segment address 2 is not.

Similarly, most operators do not act on physical segment addresses. However, relational operators that compare whether values are larger or smaller do act on physical segment addresses. The SEG operator, which gives physical segment addresses, also acts on them. Below are some actual examples.

■ Example 2 ■

```
   IF  2:1000H>1:1000H
```

This example uses a relational operator. An address comparison is performed in the operand of an IF directive. When 2:1000H and 1:1000H are compared, both offset addresses are 1000H, but the physical segment address of 2:1000H is greater. Since “2:1000H>1:1000H” checks “is 2:1000H greater than 1:1000H?” the expression evaluates to true, or 1.

■ Example 3 ■

```
DATA_SEG SEGMENT DATA #3
     CSEG
     MOV  DSR,#SEG DATA_SEG
```

In this example, the “SEG DATA_SEG” specified as the second operand of the MOV instruction gives the physical segment address of the segment symbol DATA_SEG. This was specified as physical segment #3 by the SEGMENT directive, so the expression’s value will be 3.

4.9.1.3 Unresolved Expressions During Assembly

Expressions that consist of elements with fixed values, such as integer constants and absolute symbols, can be resolved during assembly. These are called *constant expressions*. All operators can be used in constant expressions.

On the other hand, expressions that include relocatable symbols might not be resolved during assembly. These expressions are called *relocatable expressions*. RAS66K outputs information about relocatable expressions to the object file, and RL66K resolves them.

Examples of relocatable expressions are shown next.

■ Example 1 ■

```
EXTRN  DATA:GL_TBL
MOV    ER0,GL_TBL+2
```

This example adds 2 to the external symbol GL_TBL. The value of GL_TBL is not fixed during assembly, so the value of the expression “GL_TBL+2” is not fixed during assembly either.

■ Example 2 ■

```
EXTRN  DATA:GL_TBL
SB     GL_TBL.4
```

This example refers bit 4 of the address in data memory space indicated by the external symbol GL_TBL. As in example 1 above, the value of GL_TBL is not fixed during assembly, so the value of the expression “GL_TBL.4” is not fixed during assembly either.

A few operators cannot be used in calculations on relocatable symbols, and there some restrictions on coding format. Refer to Section 4.9.3, “Expression Types,” for restrictions on coding relocatable expressions.

4.9.2 Operators

This section describes the functions of the operators provided by RAS66K. RAS66K provides the following types of operators.

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

Operators include unary and binary operators. Expressions can be used on the right side of unary operators and on both sides of binary operators.

In the following explanations, “true” means a non-zero value, while “false” means a value of zero. Also, the *expression*, *expression1*, and *expression2* used in descriptions of syntax indicate expressions.

4.9.2.1 Arithmetic Operators

Arithmetic operators are operators for general arithmetic operations.

Table 4-23. Arithmetic Operators

Operator	Syntax	Meaning
+	*expression1* + *expression2*	Addition
	+ *expression1*	Positive (unary operator)
-	*expression1* - *expression2*	Subtraction
	- *expression1*	Negative (unary operator)
*	*expression1* * *expression2*	Multiplication
/	*expression1* / *expression2*	Division
%	*expression1* % *expression2*	Modulo calculation (remainder from dividing *expression1* by *expression2*)

■ Example 1 ■

```
VALUE   EQU 100H
    ADD ER1,#VALUE+1
```

In this example, the + operator is used in the operand of an ADD instruction.

■ Example 2 ■

```
BUFSIZE EQU 1024H
    DS  BUFSIZE*4
```

In this example, the * operator is used in the operand of a DS directive.

4.9.2.2 Logical Operators

Logical operators evaluate the truth or falseness of expression to the left and right (or just the right) and return a true/false value according to the condition.

Table 4-24. Logical Operators

Operator	Syntax	Meaning
&&	*expression1* && *expression2*	1 if both expressions are true; 0 otherwise.
||	*expression1* || *expression2*	1 if either expression is true; 0 otherwise.
!	! *expression*	0 if the expression is true; 1 if false.

■ Example ■

```
SW1 EQU  0
SW2 EQU  2

IF   SW1&&SW2
 BUSIZE   EQU  1024
ELSE
 BUSIZE   EQU  2048
ENDIF
```

In this example, the && operator is used in the operand of an IF directive.

4.9.2.3 Bitwise Logical Operators

Bitwise logical operators execute logical operations on each bit of an expression.

Table 4-25. Bitwise Logical Operators

Operator	Syntax	Meaning
&	*expression1* & *expression2*	Logical AND.
|	*expression1* | *expression2*	Logical OR.
^	*expression1* ^ *expression2*	Exclusive OR.
<<	*expression1* << *expression2*	Shifts *expression1* to the left by the number of bits given by *expression2*. Zeroes are shifted in from the least significant bit.
>>	*expression1* >> *expression2*	Shifts *expression1* to the right by the number of bits given by *expression2*. Zeroes are shifted in from the most significant bit.
~	*~expression*	Bit inversion.

■ Example ■

```
SCB_MSK  EQU   1111_1000B
BCB_MSK  EQU   1100_1111B
    ANDB  PSWL,#SCB_MSK&BCB_MSM
```

In this example, the & operator is used in the operand of an ANDB instruction.

4.9.2.4 Relational Operators

Relational operators compare the values of two expressions. If the condition is satisfied, then the result will be 1. If the condition is not satisfied, then the result will be 0. Relational operators can only be used in constant expressions. When two expressions that represent addresses are compared, the their physical segment addresses will be included in the comparison.

Table 4-26. Relational Operators

Operator	Syntax	Meaning
>	*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.

■ Example ■

```
IF  VALUE1>=VALUE2
  .
  .
  .
ENDIF
```

In this example, the >= operator is used in an IF directive to perform conditional assembly.

4.9.2.5 Dot Operator

The dot operator calculates bit addresses from bit offsets and data addresses.

■ **Syntax** ■

```
expression1.expression2
```

■ **Description** ■

The *expression1* specifies a data address value. The *expression2* specifies a bit position within the data address. The above syntax has the same meaning as the following expression.

((*expression1* << 3)+*expression2*)

RAS66K handles the dot operator the same as an arithmetic operator. In other words, it does not check the ranges of the values of *expression1* and *expression2*.

■ **Example** ■

Below is an example of dot operator usage. In this example, the user symbols DATSYM1 and DATSYM2 have usage type DATA. The user symbol EXTNUM1 has usage type NUMBER. Bit 0 of data memory at each of these symbol's addresses is specified as the operand of an SB instruction.

```
DATSYM1 DATA    200H
EXTRN   NUMBER:EXTNUM1

    DSEG    AT 220H
DATSYM2:
    DS    1

    CSEG
    SB    DATSYM1.0
    SB    DATSYM2.0
    SB    EXTNUM1.0
```

4.9.2.6 Special Operators

Special operators extract strings of bits from expression values. There are several types of special operators.

Table 4-27. Special Operators

Operator	Syntax	Meaning
HIGH	HIGH *expression*	Identical to ((*expression*>>16)&0FFH).
MID	MID *expression*	Identical to ((*expression*>>8)&0FFH).
LOW	LOW *expression*	Identical to (*expression*&0FFH).
SEG	SEG *expression*	Obtains physical segment address of address expression.
OFFSET	OFFSET *expression*	Obtains offset address of address expression.
PAGE	PAGE *expression*	Obtains page number of address expression.
LREG	LREG *expression*	Obtains bank number of local register set of address expression in local register area.
BPOS	BPOS *expression*	Obtains bit offset value (lower 3 bits) of expression representing a bit address.
SIZE	SIZE *segment_symbol*	Obtains segment size of segment symbol.

As opposed to other general operators, the special operators are provided for effective use of OLMS-66K Series architecture, so their applications are limited. Accordingly, there are several restrictions on expressions that use these operators.

The HIGH, MID, and LOW operators obtain specific bytes of values. These operators can be used in address expressions, but this will cause a warning.

■ Example 1 ■

The following example uses LOW, MID, and HIGH operators.

```
VALUE   EQU  123456H
     DB   LOW VALUE  ;56H
     DB   MID VALUE  ;34H
     DB   HIGH VALUE ;12H
```

The SEG operator extracts the physical segment address from an address expression, so it cannot be used on numeric value expressions. Also, if used on an address expression with the COMMON physical segment attribute, then a warning will occur and the calculated result will be 0.

The OFFSET operator extracts the offset address from an address expression. It can be used to convert address expressions to numeric values. The OFFSET operator can be used on numeric value expressions, but this will cause a warning.

■ **Example 2** ■

```
DATA_SEG SEGMENT  DATA
    MOVB      DSR,#SEG DATA_SEG           ;physical segment address
    MOV       ER0,#OFFSET DATA_SEG        ;offset address
```

In this example, the SEG and OFFSET operators are used to get the physical segment address and starting address of the relocatable segment DATA_SEG.

The PAGE operator extracts the page number from an address expression. When the usage type of an address expression is CODE, DATA, EDATA, or NONE, the PAGE operator will work the same as the calculation below.

(*expression*>>8)&0FFH

When the usage type of an address expression is CBIT, BIT, or EBIT, the PAGE operator will work the same as the calculation below.

(*expression*>>11)&0FFH

The PAGE operator can be used on numeric value expressions, but this will cause a warning and return a value that is the same as the calculation below.

(*expression*>>8)&0FFH

■ **Example 3** ■

```
     TYPE (M66507)
     .
     .
     .
DATA_SEG  SEGMENT DATA WORD INPAGE
     MOVB     ALRBH,#PAGE DATA_SEG      ;Obtains page number.
```

In this example, the PAGE operator is used to obtain the page number of relocatable segment DATA_SEG. The INPAGE specified in the operand of the SEGMENT directive is a special area attribute that allocates the segment within a page.

The LREG operator is particular to microcontrollers with the nX-8/500 CPU core. It returns the bank number of local registers. The LREG operator will work the same as the calculation below.

((OFFSET(*expression*)-200H)>>3)&0FFH

If the LREG operator is used on an expression without usage type DATA, then a warning will occur.

■ **Example 4** ■

```
     TYPE (M66507)
     .
     .
     .
DATA_SEG  SEGMENT DATA OCT LREG
     MOVB    ALRBL,#LREG DATA_SEG      ;Obtains local register bank number.
```

In this example, the LREG operator is used to obtain the local register bank number of relocatable segment DATA_SEG. The LREG specified in the operand of the SEGMENT directive is a special area attribute that allocates the segment in a local register area.

The BPOS operator obtains the bit offset of a bit address. It is provided only to access particular bit positions of a contiguous data area. The BPOS operator will work the same as the calculation below.

expression & 7

If the BPOS operator is used on an expression without usage type CBIT, BIT, or EBIT, then a warning will occur.

■ **Example 5** ■

```
    BSEG
FLG1:   DBIT  1

    DSEG
D_TBL:  DS    10H

    CSEG
    MOV  X1,#D_TBL
LOOP:
    SB   [X1].BPOS FLG1            ;Obtains bit position
```

In this example, the BPOS operator is used to obtain the bit position that is the same as FLG1 in D_TBL.

The SIZE operator obtains the size of a relocatable segment. It can only be used with segment symbols.

■ **Example 6** ■

```
DATA_SEG SEGMENT DATA WORD
     MOV     ER0,#SIZE DATA_SEG          ;Obtains segment size.
```

In this example, the SIZE operator is used to obtain the segment size of relocatable segment DATA_SEG.

4.9.3 Expression Types

From the viewpoint of how much RAS66K can resolve expression values, expression can be broadly split into three types.

- Constant expressions
- Relocatable expressions
- Simple relocatable expressions

RAS66K can resolve the values of constant expressions.

RAS66K cannot resolve the values of relocatable expressions. Relocatable expressions are resolved by RL66K. All relocatable symbols within the range allowed by syntax can be used in relocatable expressions.

Simple relocatable expressions are a type of relocatable expression for which the offset address from the segment base of the program can be determined. Use of relocatable symbols in simple relocatable expressions is limited to simple relocatable symbols.

■ Example 1 ■

```
ABS_DATA  DATA    1000H

EXTRN     DATA:EXT_DATA

DATA_SEG  SEGMENT  DATA WORD
      RSEG     DATA_SEG
D_TBL:   DS      10H

      CSEG
      MOV      ER0,#ABS_DATA+10H
      MOV      ER1,#EXT_DATA+10H
      MOV      ER2,#D_TBL+10H
```

Look at the second operand of the MOV instructions in order. First, the value of ABS_DATA is fixed to 1000H, so the value of the expression “ABS_DATA+10H” is fixed to 1010H. Thus, it is a constant expression.

Second, the expression “EXT_DATA+10H” adds to an address that represents an external symbol, so RAS66K cannot determine the expression’s value. Thus, it is a relocatable expression.

Third, the address of D_TBL itself is not fixed, but its offset value from the segment base of this program is 0. Therefore the value of the expression “D_TBL+10H” is at least fixed to 10H within this program. Thus, it is a simple relocatable expression.

The reason for classifying instructions like this is that there are restrictions on the types of expressions that can be specified in operands depending on the type of instruction or directive. Expressions coded in instruction and directive operands are classified by how freely they can be used.

- Constant expressions
- Simple expressions
- General expressions

Constant expressions are expressions for which RAS66K can determine values, as previously described. Constant expressions can be coded least freely.

Simple expressions are either constant expressions or simple relocatable expressions. They are expressions for which RAS66K can at least determine the offset address from the segment base of the program.

General expressions are either constant expressions or relocatable expressions. They include all permitted syntaxes, so they can be coded most freely.

General expressions can be coded from most operands of basic instructions, but only constant expressions or simple expressions are recognized as operands of many directives. The reason for these restrictions is that depending on the type of directive, RAS66K processing might not proceed correctly unless the operand is a constant expression or simple expression. Section 4.9.3.4, “Restrictions On Coding Expressions,” shows some actual examples.

Refer to Section 4.11, “Addressing Modes,” regarding expressions that can be used as instruction operands. Refer to Section 4.12, “Using Directives,” regarding expressions that can be used as directive operands.

The formats of constant expressions, simple expressions, and general expressions are described below.

4.9.3.1 Constant Expressions

Constant expressions are expressions for which RAS66K can determine values. More specifically, constant expressions consist of integer constants, character constants, absolute symbols, and location counters of absolute segments, connected by operators. A constant expression cannot use relocatable symbols because RAS66K must determine the constant expression's value. However, in the following cases even expressions that include relocatable symbols will be constant expressions.

- Expressions that evaluate as differences between simple relocatable symbols that reside in the same relocatable segment will be constant expressions.
- Expressions that perform the dot operation (.) on relocatable symbols that reside in the same relocatable segment will be constant expressions.
- Expressions that perform the BPOS operation on a dot operation on relocatable symbols will be constant expressions.

■ Example ■

Below is an example using constant expressions.

```
ABSSYM1 EQU     100H

DATSEG4 SEGMENT DATA
    RSEG    DATSEG4
LABEL1:
    DS    2
LABEL2:
```

Constant expressions that use symbols as defined above and their values are shown below.

Constant Expression	Value
100H	100H
'A'	41H
ABSSYM1	100H
1+2	3H
(1+2)	3H
+(1+2*ABSSYM1)	201H
100H*'A'	4100H
(LABEL2-LABEL1)	2H
(LABEL2.0-LABEL1.0)	10H
BPOS (LABEL1.3)	3H

4.9.3.2 Simple Expressions

Simple expressions include no relocatable symbols other than simple relocatable symbols. Location counter symbols residing in relocatable segments can be used as simple relocatable symbols. Simple expressions include constant expressions.

Simple expressions have the following syntax.

■ Syntax ■

Expression	Definition
simple expression	constant expression
	| simple relocatable symbol
	| (simple expression)
	| +simple expression
	| simple expression + constant expression
	| constant expression - simple expression
	| simple expression - constant expression
	| simple expression.constant expression
	| OFFSET simple expression

The vertical bars (|) in this syntax definition indicate that just one of the several items is to be specified.

This syntax has some restrictions. If a dot operator is used in a simple expression that is not a constant expression, then that expression cannot use a further dot operator or OFFSET operator. If a simple expression includes simple relocatable symbols, then RAS66K cannot determine the value of the relocatable expression. In these cases, RL66K will determine the final value of the simple expression.

■ Example ■

Below are examples of simple expressions that are not constant expressions.

```
DATSEG5 SEGMENT DATA
        RSEG    DATSEG5
        DS      2
LABEL3:
        ORG     LABEL3
```

Simple expressions that use symbols as defined above are shown below.

Simple Expression
(LABEL3)
+LABEL3
LABEL3-1
LABEL3.4
OFFSET LABEL3

4.9.3.3 General Expressions

General expressions add expressions that include segment symbols, external symbols, and communal symbols to simple expressions. General expressions have the following syntax.

■ Syntax ■

Expression	Definition
general expression	constant expression
	| relocatable expression
	| relocatable operation expression
relocatable expression	relocatable symbol
	| (relocatable expression)
	| + relocatable expression
	| relocatable expression + constant expression
	| constant expression + relocatable expression
	| relocatable expression - constant expression
	| relocatable expression . constant expression
	| OFFSET relocatable expression
relocatable operation expression	HIGH relocatable expression
	| MID relocatable expression
	| LOW relocatable expression
	| SEG relocatable expression
	| PAGE relocatable expression
	| LREG relocatable expression
	| BPOS relocatable expression
	| SIZE segment symbol
	| (relocatable operation expression)

The vertical bars (|) in this syntax definition indicate that just one of the several items is to specified.

When a relocatable expression includes a dot operator, it cannot use special operators or additional dot operators. RAS66K cannot determine values of relocatable expressions. RL66K determines the final values of relocatable expressions.

■ **Example** ■

Below are examples of general expressions that are not simple expressions or constant expressions.

```
SEGSYM  SEGMENT  DATA
COMMSYM COMM     DATA 2
EXTRN   DATA:EXTSYM BIT:BITSYM
```

General expressions that use symbols as defined above are shown below.

General Expression
EXTSYM
(COMMSYM)
+EXTSYM
COMMSYM-1
EXTSYM.1
HIGH EXTSYM
LOW (COMMSYM)
SEG (+EXTSYM)
OFFSET (COMMSYM-1)
PAGE SEGSYM
LREG (COMMSYM)
BPOS BITSYM
SIZE SEGSYM

4.9.3.4 Restrictions On Coding Expressions

There are restrictions on where each type of expression can be coded. Also, forward references to user symbols within expressions are not always permitted. These restrictions are explained below.

(1) Restrictions On ORG Directive Operands

For source statements that reside in an absolute segment, only constant expressions can be specified as operands of ORG directives. This is because the address of the operand must be determined during assembly.

For source statements that reside in a relocatable segment, simple expressions can be specified as operands of ORG directives. However, simple relocatable symbols included in the simple expressions must reside in the current relocatable segment. This is because even though RAS66K cannot determine the address of the relocatable segment, it must be able to determine the relationship between relative addresses. If the relationship of relative addresses is determined, then size of a relocatable segment can be determined. RL66K uses these calculated sizes to allocate logical segments to memory.

■ **Example** ■

```
    TYPE(M66507)
XCODSEG SEGMENT CODE
    RSEG    XCODSEG
XLABEL:
    .
    .
    .
CODESEG SEGMENT CODE
    RSEG    CODESEG
    .
    .
    .
    ORG     10H
    .
    .
    .
LABEL:
    .
    .
    .
    ORG      LABEL+100H
    .
    .
    .
    ORG     XLABEL          ;error
    .
    .
    .
```

The constant expression “10H,” the simple expression “LABEL+100H,” and the simple expression “XLABEL” are specified in operands of ORG directive statements that reside in the relocatable segment CODESEG. The operand “XLABEL” is a simple relocatable symbol that does not reside in the current relocatable segment, so an error will occur.

(2) Restrictions On Operands Of Directives That Define Local Symbols

Simple expressions can be specified in operands of directives that define local symbols, such as EQU directives and CODE directives. However, general expressions that are not simple expressions cannot be specified. Also, forward references to user symbols specified in operands of directives that define symbols are not permitted.

(3) Restrictions On Operands Of Other Directives

There are also restrictions on which expressions can be coded in operands of directives other than those of Section 4.9.3.4.1 and 4.9.3.4.2. Refer to the description of each directive for these restrictions.

(4) Restrictions On Microcontroller Instruction Operands

Within operands of microcontroller instructions, shift widths of rotate/shift instructions and bit positions of bit addressing can be specified only by constant expressions. General expressions can be used for other addressing modes.

Both forward and backward references to user symbols in microcontroller instruction operands are permitted.

4.9.4 Expression Evaluation

4.9.4.1 Operator Precedence

Operator precedence determines the order of evaluation for expressions. Operators are evaluated in order from highest precedence to lowest. Operators with the same precedence are evaluated from left to right in the order they are written.

Operator precedence is as follows. The higher the precedence, the lower the number of precedence.

Table 4-28. Operator Precedence

Precedence	Operators
1	()
2	.
3	! ~ +(unary) - (unary) HIGH MID LOW SEG OFFSET PAGE LREG BPOS SIZE
4	* / %
5	+ (binary) - (binary)
6	<< >>
7	< <= > >=
8	== !=
9	&
10	^
11	|
12	&&
13	||

■ **Example** ■

```
LABEL   DATA    200H

    MOV      LABEL+2*8,#0
    SB      (LABEL+2).7
    SB      LABEL+2.7
```

This example defines the absolute symbol LABEL with usage type DATA. After that are three instruction statements that have expressions using this symbol as operands.

The value of the destination operand of the instruction on the third line is 210H. The operand of the instruction on the fourth line indicates bit 7 of the data memory at address LABEL+2. Pay particular attention to the operand of the last instruction, which does not indicate bit 7 of the data memory at address LABEL+2. This is because the dot operator (.) has higher precedence than the + operator, so the operand is evaluated as “LABEL+(2.7)”.

4.9.4.2 Evaluation Of Expression Values

RAS66K handles numbers as unsigned 32-bit numbers. When it calculates numbers in expressions, calculation results of operators are also handled as unsigned 32-bit numbers. For expressions that express addresses, the physical segment addresses are handled as unsigned 8-bit numbers. RAS66K calculates numbers in expressions according to evaluation order. It checks the valid range corresponding to the appropriate operand for each calculation result.

4.9.4.3 Evaluation Of Expression Attributes

This section explains how RAS66K evaluates attributes of expressions.

For an expression that consists of only one symbol, its attribute will be the attribute of the symbol

For an expression that uses operators, its attribute will differ depending on the operators used and the attributes of operands. The operators used may restrict the types of operands. RAS66K will generate an error or warning for an expression that violates these restrictions. It will generate an error when it cannot calculate the result of an expression. It will generate a warning when it can calculate a result of an expression but that result has no meaning.

The following sections explain for each operator how the attributes of calculated results are determined and how errors and warnings will occur, based on the type of operator and operands. The explanations make use of the following terms to represent types of expressions.

Term	Meaning
address_expression	An expression that represents an address. In other words, an expression with usage type NONE, CODE, DATA, EDATA, CBIT, BIT, or EBIT.
number_expression	An expression that represents a number. In other words, an expression with usage type NUMBER.
code_expression	An expression with usage type CODE.
data_expression	An expression with usage type DATA.
edata_expression	An expression with usage type EDATA.
cbit_expression	An expression with usage type CBIT.
bit_expression	An expression with usage type BIT.
ebit_expression	An expression with usage type EBIT.
none_expression	An expression with usage type NONE.
segment_symbol	A single segment symbol.

(1) Attributes of Parentheses ( )

Calculations that use the operators ( ) are evaluated as follows.

Table 4-29. Evaluation Of Attributes Of Operator ( )

Expression Format	Expression Attribute	Errors
(*address_expression*)	Attribute of *address_expression*	
(*number_expression*)	NUMBER	

■ Description ■

The attribute of an expression enclosed in parentheses will not change.

(2) Attribute Evaluation of Operators + and -

Calculations that use the operators + and - are evaluated as follows.

Table 4-30. Evaluation Of Attributes Of Operators + and -

Expression Format	Expression Attribute	Errors
+ *address_expression*	Attribute of *address_expression*	
+ *number_expression*	NUMBER	
number_expression + *number_expression*	NUMBER	
number_expression + *address_expression*	Attribute of *address_expression*	
address_expression + *number_expression*	Attribute of *address_expression*	
address_expression + *address_expression*	NUMBER	Warning
- *address_expression*	Attribute of *address_expression*	
- *number_expression*	NUMBER	
number_expression - *number_expression*	NUMBER	
number_expression - *address_expression*	Attribute of *address_expression*	
address_expression - *number_expression*	Attribute of *address_expression*	
address_expression - *address_expression*	NUMBER	Possible error

■ Description ■

Expression attributes are inherited unchanged with the unary operators + and - .

If either the left term or right term of a binary operator + or - is an address expression, then the address attribute will be inherited.

Addition of two address expressions will cause a warning. Subtraction of two address expressions is permitted only when both addresses are in the same logical segment.

(3) Attribute Evaluation of Operators *, /, and %

Calculations that use the operators *, /, and % are evaluated as follows.

Table 4-31. Evaluation Of Attributes Of Operators *, /, and %

Expression Format	Expression Attribute	Errors
number_expression * *number_expression*	NUMBER	
number_expression * *address_expression*	NUMBER	
address_expression * *number_expression*	NUMBER	
address_expression * *address_expression*	NUMBER	Warning
number_expression / *number_expression*	NUMBER	
number_expression / *address_expression*	NUMBER	Warning
address_expression / *number_expression*	NUMBER	
address_expression / *address_expression*	NUMBER	Warning
number_expression % *number_expression*	NUMBER	
number_expression % *address_expression*	NUMBER	Warning
address_expression % *number_expression*	NUMBER	
address_expression % *address_expression*	NUMBER	Warning

■ Description ■

Use of the *, /, or % operator with two address expressions will cause a warning. A warning will also occur when the right term of the / or % operator is an address expression.

All calculated results will be of type number.

(4) Attribute Evaluation of Logical Operators

Calculations that use logical operators are evaluated as follows.

Table 4-32. Evaluation Of Attributes Of Logical Operators

Expression Format	Expression Attribute	Errors
number_expression && *number_expression*	NUMBER	
number_expression && *address_expression*	NUMBER	Warning
address_expression && *number_expression*	NUMBER	Warning
address_expression && *address_expression*	NUMBER	Warning
number_expression || *number_expression*	NUMBER	
number_expression || *address_expression*	NUMBER	Warning
address_expression || *number_expression*	NUMBER	Warning
address_expression || *address_expression*	NUMBER	Warning
! *number_expression*	NUMBER	
! *address_expression*	NUMBER	Warning

■ Description ■

Use of a logical operators with an address expression will cause a warning.

All calculated results will be of type number.

(5) Attribute Evaluation of Bitwise Logical Operators

Calculations that use bitwise logical operators are evaluated as follows.

Table 4-31. Evaluation Of Attribute Of Bitwise Logical Operators

Expression Format	Expression Attribute	Errors
number_expression & *number_expression*	NUMBER	
number_expression & *address_expression*	NUMBER	
address_expression & *number_expression*	NUMBER	
address_expression & *address_expression*	NUMBER	Warning
number_expression | *number_expression*	NUMBER	
number_expression | *address_expression*	NUMBER	
address_expression | *number_expression*	NUMBER	
address_expression | *address_expression*	NUMBER	Warning
number_expression ^ *number_expression*	NUMBER	
number_expression ^ *address_expression*	NUMBER	
address_expression ^ *number_expression*	NUMBER	
address_expression ^ *address_expression*	NUMBER	Warning
number_expression << *number_expression*	NUMBER	
number_expression << *address_expression*	NUMBER	Warning
address_expression << *number_expression*	NUMBER	
address_expression << *address_expression*	NUMBER	Warning
number_expression >> *number_expression*	NUMBER	
number_expression >> *address_expression*	NUMBER	Warning
address_expression >> *number_expression*	NUMBER	
address_expression >> *address_expression*	NUMBER	Warning
~*number_expression*	NUMBER	
~*address_expression*	NUMBER	Warning

■ Description ■

Use of a bitwise logical operator with two address expressions will cause a warning. A warning will also occur when the right term of the << or >> operator is an address expression. A warning will also occur when an address expression is used with the ~ operator.

All calculated results will be of type number.

(6) Attribute Evaluation of Relational Operators

Calculations that use relational operators are evaluated as follows.

Table 4-34. Evaluation Of Attribute Of Relational Operators

Expression Format	Expression Attribute	Errors
number_expression > *number_expression*	NUMBER	
number_expression > *address_expression*	NUMBER	Warning
address_expression > *number_expression*	NUMBER	Warning
address_expression > *address_expression*	NUMBER	
number_expression >= *number_expression*	NUMBER	
number_expression >= *address_expression*	NUMBER	Warning
address_expression >= *number_expression*	NUMBER	Warning
address_expression >= *address_expression*	NUMBER	
number_expression < *number_expression*	NUMBER	
number_expression < *address_expression*	NUMBER	Warning
address_expression < *number_expression*	NUMBER	Warning
address_expression < *address_expression*	NUMBER	
number_expression <= *number_expression*	NUMBER	
number_expression <= *address_expression*	NUMBER	Warning
address_expression <= *number_expression*	NUMBER	Warning
address_expression <= *address_expression*	NUMBER	
number_expression == *number_expression*	NUMBER	
number_expression == *address_expression*	NUMBER	Warning
address_expression == *number_expression*	NUMBER	Warning
address_expression == *address_expression*	NUMBER	
number_expression != *number_expression*	NUMBER	
number_expression != *address_expression*	NUMBER	Warning
address_expression != *number_expression*	NUMBER	Warning
address_expression != *address_expression*	NUMBER	

■ Description ■

Use of a relational operator between an address expression and a number expression will cause a warning.

A comparison of two address expressions will also compare their physical segment addresses. However, if the physical segment attribute of either address expression is COMMON, then only the offset addresses will be compared.

All calculated results will be of type number.

(7) Attribute Evaluation of Dot Operator

Calculations that use dot operators are evaluated as follows.

Table 4-35. Evaluation Of Attribute Of Dot Operators

Expression Format	Expression Attribute	Errors
number_expression . number_expression	NUMBER	
code_expression . number_expression	*cbit_expression*	
data_expression . number_expression	*bit_expressione*	
data_expression . number_expression	*ebit_expression*	
cbit_expression . number_expression	NUMBER	Warning
bit_expression . number_expression	NUMBER	Warning
ebit_expression . number_expression	NUMBER	Warning
none_expression . number_expression	*none_expression*	
number_expression . address_expression	NUMBER	Warning
code_expression . address_expression	*cbit_expression*	Warning
data_expression . address_expression	*bit_expression*	Warning
edata_expression . address_expression	*ebit_expression*	Warning
cbit_expression . address_expression	NUMBER	Warning
bit_expression . address_expression	NUMBER	Warning
ebit_expression . address_expression	NUMBER	Warning
none_expression . address_expression	*none_expression*	Warning

■ Description ■

A warning will occur if an address expression is coded as the right term of a dot operator. A warning will also occur if an address expression with usage type CBIT, BIT, or EBIT is coded as the left term.

The attribute of the calculated result will differ depending on the usage type of the expression coded as the right term.

(8) Attribute Evaluation of Special Operators

Calculations that use special operators are evaluated as follows.

Table 4-36. Evaluation Of Attribute Of Special Operators

Expression Format	Expression Attribute	Errors
HIGH *number_expression*	NUMBER	
HIGH *address_expression*	NUMBER	Warning
MID *number_expression*	NUMBER	
MID *address_expression*	NUMBER	Warning
LOW *number_expression*	NUMBER	
LOW *address_expression*	NUMBER	Warning
SEG *number_expression*	NUMBER	Error
SEG *address_expression*	NUMBER	
OFFSET *number_expression*	NUMBER	Warning
OFFSET *address_expression*	NUMBER	
PAGE *number_expression*	NUMBER	Warning
PAGE *address_expression*	NUMBER	
LREG *number_expression*	NUMBER	Warning
LREG *data_expression*	NUMBER	
LREG *address_expression* (other than usage type DATA)	NUMBER	Warning
BPOS *number_expression*	NUMBER	Warning
BPOS *code_expression*	NUMBER	Warning
BPOS *data_expression*	NUMBER	Warning
BPOS *edata_expression*	NUMBER	Warning
BPOS *cbit_expression*	NUMBER	
BPOS *bit_expression*	NUMBER	
BPOS *ebit_expression*	NUMBER	
BPOS *none_expression*	NUMBER	Warning
SIZE *segment_symbol*	NUMBER	

■ Description ■

A warning will occur when an address expression is used with the HIGH, MID, or LOW operators.

The SEG operator returns a physical segment address, so it cannot be used with number expressions.

The OFFSET, PAGE, LREG, and BPOS operators operate on addresses, so a warning will occur when one is used with a number expression. The LREG operator returns a local register bank number, so it will cause an error if used with an address that is not usage type DATA. The BPOS operator returns a bit offset, so it will cause an error if used with an address that is not a bit address.

Only a segment symbol can be coded as the right term of the SIZE operator.

All calculated results will be of type number.

4.10 Check Functions

In addition to syntax checking, RAS66K has numerous other check functions for instruction use and operand contents. These check functions can alert the programmer to possibly erroneous operation in his program.

RAS66K attempts to perform the checks described in this section as much as possible, but in some cases checks cannot be performed during assembly due to lack of necessary information. For example, relocatable symbols may be specified as operands. In such cases, RAS66K will output check information to the object file, and RL66K will perform the checks that could not be performed during assembly instead of RAS66K.

■ Attention ■

Among the check functions explained in this section, the following operate with "assumptions" about appropriate register contents by using the USING directive.

- DSR checks
- TSR checks
- Current page checks
- Flag attribute checks

Please pay close attention wherever an "assumption" is given. The USING directive simply informs RAS66K and RL66K of register values; it does not generate object code for setting registers to values. To actually set a register to a value, use a microcontroller instruction.

4.10.1 Operand Value Checks

The values of instruction and directive operands are checked to be in permitted ranges. If an operand is outside its permitted range, then RAS66K will output an error. Operand value ranges differ depending on instruction addressing modes and directive types.

Value ranges for each addressing mode of instructions are described in Section 4.11, “Addressing Modes.” Value ranges for operands of each directive are described in Section 4.12, “Directives.”

■ Example ■

```
EXTRN   DATA:ZBUF
NUMSYM2 EQU 1000H

    MOVB   ZBUF,#NUMSYM2   ;ERROR
```

In this example, RAS66K checks whether the value of NUMSYM2, the operand of the MOVB instruction, is within the following ranges.

```
      -80H —   -1H (0FFFFFF80H—0FFFFFFFFH)
       0H  —  0FFH
```

The value of NUMSYM2 falls outside these ranges, so RAS66K will output an error.

Because ZBUF is a relocatable symbol, RL66K will check the value of this operand.

4.10.2 Location Counter Value Checks

RAS66K performs the following checks on location counter values.

- It checks whether absolute segment location counter values are within the range of address space.

- It checks whether relocatable segment location counter values are smaller than or equal to the size of the area allocated to the relocatable segments.

■ Example ■

```
PAGE_SEG SEGMENT  DATA WORD IPAGE
    RSEG     PAGE_SEG
TBL1:   DS      80H
TBL2:   DS      80H
TBL3:   DS      80H  ;ERROR
```

In this example, the relocatable segment PAGE_SEG is specified to have the special area attribute INPAGE, so its segment size cannot exceed 100H, which is the maximum size of 1 page. Because the third DS directive causes the segment size to exceed 100H, RAS66K will generate an error at the third DS directive.

4.10.3 Usage Type Checks

RAS66K checks for inconsistencies between operand usage type and instruction or directive function. If there is an inconsistency, then a warning will occur. Refer to Section 4.11, “Addressing Modes,” and Section 4.12, “Directives,” for the contents of the checks.

■ **Example** ■

```
DATSYM2 DATA    200H
CODSYM1 CODE    0FEH

     LC    A,DATSYM2        ;Warning
     L     A,CODSYM1        ;Warning
```

In this example, RAS66K performs the following checks on the operands of the LC and L instructions.

The second operand of an LC instruction should be an address in CODE address space, so RAS66K checks whether the usage type of DATSYM2 is CODE, NONE, or NUMBER. In this example, DATSYM2 has usage type DATA, so a warning will occur.

Similarly, the second operand of an L instruction should be an address in DATA address space, so RAS66K checks whether the usage type of CODSYM1 is DATA, NONE, or NUMBER. In this example, CODSYM1 has usage type CODE, so a warning will occur.

Operands with usage type NUMBER can be used anywhere. Operands with usage type NONE can be used anywhere that an address is expected.

4.10.4 Physical Segment Address Checks

The explanations given in this section take the nX-8/500 CPU core architecture as an example.

OLMS-66K Series data memory space and program memory space are configured from multiple physical segments. The number of physical segments is determined by the type of microcontroller.

To refer a data memory space or program memory space configured from multiple physical segments, a program must manage the physical segment addresses. Physical segment addresses are managed by using the segment registers allocated in the special function registers.

Physical segments of data memory space are set using the data segment register (DSR). When referring program memory space as data, physical segments are set using the table segment register (TSR). A program will refer only the physical segment specified in the appropriate segment register.

The USING DSREG directive supports management of the DSR. The USING TSREG directive supports management of the TSR. These directives can inform RAS66K and RL66K of the values of DSR or TSR set within the program. RAS66K and RL66K compare this value to the physical segment addresses of the instruction and directive operands coded within each specified range. Because of this physical segment checking, physical segment addresses are included in addresses specified as operands of instructions.

Physical segments of program memory space which contain executable instructions are set using the program segment register (CSR). The CSR is automatically overwritten by inter-segment jump and call instructions (FJ and FCAL). The programmer cannot directly manipulate the CSR.

RAS66K has three types of physical segment register checks.

- DSR checks
- TSR checks
- CSR checks

Each of these is explained in the following sections.

(1) DSR Checks

When a DSR value assumption is made using the USING DSREG directive, RAS66K will check if this assumed value matches the physical segment addresses of instruction operands that represent addresses in data memory space. However, if an address is in the COMMON area, then this check will not be performed. If the check does not result in a match, then a warning will occur.

If the operand value and the set segment register value can both be determined during assembly, then RAS66K will perform the check. If either cannot be determined during assembly, then RL66K will perform the check.

■ Example ■

```
EXTRN   DATA:ADRS1 ADRS2
;
    USING   DSREG ADRS1           ;Assume DSR is set physical segment address of ADRS1
;
    MOVB    DSR,#SEG ADRS1        ;Instruction sets DSR to SEG ADRS1 during execution.
    MOV     ER0,ADRS1             ;DSR check performed
    MOV     ER1,ADRS2             ;DSR check performed
;
    USING   DSREG ANY             ;DSR check suspended
;
    MOV     ER2,200H              ;DSR check not performed
```

In this example, the physical segment addresses of the source operands of the two instructions below (ADRS1, ADRS2) are checked to match SEG ADRS1.

```
    MOV     ER0,ADRS1
    MOV     ER1,ADRS2
```

These checks are performed because RAS66K and RL66K are informed by the USING DSREG directive that the DSR value is SEG ADRS1. ADRS1 and ADRS2 are relocatable symbols, so RL66K will perform these checks.

The USING DSREG ANY directive cancels the assumption of DSR value, stopping RAS66K and RL66K from performing DSR checks. A DSR check is not performed for the MOV instruction immediately following the USING DSREG ANY specification.

(2) TSR Checks

When a TSR value assumption is made using the USING TSREG directive, RAS66K will check if this assumed value matches the physical segment addresses of instruction operands that represent addresses in program memory space. If the check does not result in a match, then a warning will occur.

If the operand value and the set segment register value can both be determined during assembly, then RAS66K will perform the check. If either cannot be determined during assembly, then RL66K will perform the check.

■ Example ■

```
EXTRN   CODE:C_TBL1 C_TBL2
;
     USING   TSREG C_TBL1          ;Assume TSR is set physical segment address of
                                   ;C_TBL1.
;
     MOVB    TSR,#SEG C_TBL1       ;Instruction sets TSR to SEG C_TBL1 during execution.
     LC      A,C_TBL1         ;TSR check performed
     LC      A,C_TBL2         ;TSR check performed
;
     USING   TSREG ANY             ;TSR check suspended
;
     LC      A,4000H               ;TSR check not performed
```

In this example, the physical segment addresses of the source operands of the two instructions below (C_TBL1, C_TBL2) are checked to match SEG C_TBL1.

```
     LC      A,C_TBL1
     LC      A,C_TBL2
```

These checks are performed because RAS66K and RL66K are informed by the USING TSREG directive that the TSR value is SEG C_TBL1. C_TBL1 and C_TBL2 are relocatable symbols, so RL66K will perform these checks.

The USING TSREG ANY directive cancels the assumption of TSR value, stopping RAS66K and RL66K from performing TSR checks. A TSR check is not performed for the LC instruction immediately following the USING TSREG ANY specification.

(3) CSR Checks

CSR checks are performed when jump or call instructions within a physical segment (J, SJ, CAL, ACAL, SCAL) are made. They check whether the physical segment addresses of the branch source and branch destination match. If a check does not result in a match, then a warning will occur.

■ Example ■

```
EXTRN   CODE:SUB_PROC1
;
     CSEG    #1
SUB_PROC2:
     .
     .
     .
;
     CSEG    #2
     CAL      SUB_PROC1       ;RL66K will perform a CSR check.
     CAL      SUB_PROC2       ;RAS66K will perform a CSR check.
     .
     .
     .
```

In this example, both CAL instructions are coded in physical segment 2. The branch destination of the first CAL instruction (SUB_PROC1) is an external symbol, but RAS66K cannot know the physical segment of SUB_PROC1. Therefore RL66K will perform the CSR check.

On the other hand, it is determined during assembly that the physical segment address of the branch destination of the second CAL instruction (SUB_PROC2) is 1. In this example, the physical segment addresses do not match, so a warning will occur.

4.10.5 Word Boundary Checks

For OLMS-66K Series data memory, word boundaries exist when accessing words (2 bytes). Word accesses are performed on word boundaries. In other words, word-length accesses are only possible when the first byte of a word is at an even address. Word-length accesses are not possible when the first byte of a word is at an odd address. If such an operand is coded, then the 2 bytes accessed will start at the address with the least significant bit assumed 0.

RAS66K will output a warning when an odd address is given as an operand for which a word access will be performed. However, RAS66K will output a warning only if it can determine the operand's value. If RAS66K cannot determine the operand's value, then RL66K will perform the word boundary check.

RAS66K and RL66K perform this check only when they can determine the address to be accessed. When the operand addressing mode is indirect addressing, RAS66K and RL66K cannot determine the address to be accessed. In such a case, a word boundary check will not be performed. This is because the contents of the register used for indirect addressing would have to be known when the instruction is executed in order to perform the check. That would require the program's entire flow to be examined. This type of processing is not the role of an assembler.

■ **Example** ■

```
MOV     ER0,1000H;No warning
MOV     ER1,1001H;Warning
MOVB     R2,1002H ;No warning
MOVB     R3,1003H ;No warning
```

In this example, the MOV instruction operands are accessed as words. When the operand is the odd address 1001H, a warning will be output.

4.10.6 Special Function Register Access Checks

OLMS-66K Series data memory has special function registers for controlling microcontroller peripheral functions. Corresponding to the peripheral functions they control, some special function registers are limited to read-only or write-only accesses. The restrictions on accesses to these special function registers differ depending on the target microcontroller. RAS66K reads information related to these accesses from the DCL file. Using this information, RAS66K checks whether accesses to special function registers are correct or not. If an access is not correct, then a warning will occur.

RAS66K checks accesses to special function registers only when it can determine the addresses to be accessed. When the operand is indirect addressing, or when RL66K determines the address to be accessed, RAS66K will not check accesses to special function registers.

■ **Example** ■

```
MOVB     WDT,#3CH          ;correct access
MOV      WDT,#003CH        ;warning
MOVB     R1,WDT    ;warning
SB     P5.7        ;warning
```

WDT can only be accessed by byte, and although it can be it written, it cannot be read. Also, bit 7 of P5 cannot be accessed. In this example, the first line's MOVB instruction writes a byte to WDT, so the access is correct. The second line's MOV instruction accesses WDT as a word, so a warning will occur. The third line's MOVB instruction reads WDT, so a warning will occur. The fourth line's SB instruction accesses an inaccessible bit, so a warning will occur.

4.10.7 Current Page Checks

The OLMS-66K Series provides two addressing modes for accessing the current page area of data memory space: current page addressing (off address, \address) and current page SBA area addressing (sbaoff address, \address). When current page addressing is used, the page number of the address coded as an instruction operand must be checked to match the page number actually set when the instruction is executed.

The USING PAGE directive supports management of the current page. This directive can inform RAS66K and RL66K of each page number set within the program. RAS66K and RL66K compare this value to instruction operands with current page addressing. If the check does not result in a match, then a warning will occur.

■ **Example** ■

```
EXTRN DATA : G_DATA
DAT_SEG SEGMENT DATA WORD
    RSEG    DAT_SEG
TABLE1: DS      100H
;
    CSEG
    USING   PAGE  DAT_SEG          ;Assume a page number.
    MOVB     ALRBH,#PAGE DAT_SEG         ;Instruction sets page number during execution.
    MOV     ER0,off G_DATA  ;Current page check performed
    MOV     ER1,off TABLE1  ;Current page check performed
;
    USING   PAGE ANY                ;Current page check suspended
;
    MOV     ER2,off 1234H           ;Current page check not performed
```

In this example, the page numbers of the source operands of the two instructions below (G_DATA, TABLE1) are checked to match the page number of DAT_SEG.

```
    MOV     ER0,off G_DATA
    MOV     ER1,off TABLE1
```

These checks are performed because RAS66K and RL66K are informed by the USING PAGE directive that the current page number is that of DAT_SEG. G_DATA and TABLE1 are relocatable symbols, so RL66K will perform these checks.

The USING PAGE ANY directive cancels the assumption of page number, stopping RAS66K and RL66K from performing current page checks. A current page check is not performed for the MOV instruction immediately following the USING PAGE ANY specification.

4.10.8 Program Memory Space Write Checks

Microcontrollers with the nX-8/500 CPU core can use the ROM window function. The ROM window function allocates a fixed area of program memory to data memory.

The ROM window function allows program memory to be accessed when an instruction that accesses data memory is used. It allows program memory data to be read but not written. RAS66K checks that instructions that access the ROM window area do not write to program memory. When checks cannot be performed during assembly, RL66K performs them.

■ **Example** ■

```
    TYPE   (M66507)
    WINDOW 5000H,6FFFH
;
;
    CSEG  AT  5000H
ROM_TBL:
    DW    100H
    DW    112H
    .
    .
    .
;
    CSEG  AT  1000H
    AND   A,ROM_TBL            ;Read
    MOV   ROM_TBL,ER1   ;Write
```

In this example, the table that starts from ROM_TBL resides in the ROM window area. The AND instruction reads data of ROM_TBL so it causes no problem, but the MOV instruction writes to ROM_TBL so it causes a warning.

4.10.9 Flag Attribute Checks

Some OLMS-66K Series microcontroller instructions are affected by the data descriptor (DD) or the stack flag (SF). Furthermore, some instructions automatically change the states of these flags. (Only microcontrollers with the nX-8/300 CPU core have the stack flag.)

You must be aware of flag states when creating an OLMS-66K Series application program:

- Are flag states correct for instructions affected by flags?
- For branch instructions, do flag states for branch sources and branch destinations differ inappropriately?

RAS66K provides the USING DATA directive and USING OPRT directive to note flag state changes in the program flow and to perform the above two checks. The USING DATA directive gives an assumption about the state of DD, and the USING OPRT directive gives an assumption about the state of SF. RAS66K issues a warning when it encounters an instruction with incorrect flag attributes.

The two flag attribute checks performed by RAS66K are explained below.

(1) Flag Attribute Checks of Instructions Affected By Flags

RAS66K checks if flag states are correct for instructions affected by flags. Refer to the instruction manual of your target microcontroller to see which instructions are affected by which flags.

■ Example ■

```
    TYPE    (M66507)
    .
    .
    .
    USING  DATA  WORD            ;Assume DD state is word.
    SDD                          ;Instruction sets DD.
    AND    A,#0F800H             ;Word instruction is okay.
    ANDB   A,#0F8H          ;Byte instruction causes warning.
;
;
    USING  DATA  BYTE            ;Assume DD state is byte.
    RDD                          ;Instruction resets DD.
    AND    A,#0F800H             ;Word instruction causes warning.
    ANDB   A,#0F8H          ;Byte instruction is okay.
;
;
    USING  DATA  ANY        ;Cancel assumption of DD state.
    AND    A,#0F800H             ;Check is not performed.
    ANDB   A,#0F8H          ;Check is not performed.
```

In the above example, the first USING DATA directive assumes that the state of DD is word. In this state, only word-type instructions can be used from among instructions affected by DD. Thus, the ANDB instruction causes a warning.

The next USING DATA directive assumes that the state of DD is byte. In this state, only byte-type instructions can be used from among instructions affected by DD. Thus, the AND instruction causes a warning.

Finally a USING DATA ANY directive is specified. This declares that RAS66K should make no assumption about DD. In this state, instructions affected by DD are not checked. Thus, the AND and ANDB instructions will not cause warning.

(2) Flag Attribute Checks of Branch Instructions

RAS66K checks whether the flag attributes of branch sources and branch destinations of branch instructions match.

Branch instruction flag attribute checks are performed only when the /CF option or the CHK directive is specified.

■ Example ■

```
    USING  DATA  WORD       ;Assume DD state is word.
    CAL    PROC1            ;DD state matches, so instruction is okay.
    CAL    PROC2            ;DD state differs, so instruction causes warning.
    CAL    PROC3            ;Check not performed.
    .
    .
    .
    USING  DATA  WORD       ;Assume DD state is word.
PROC1:
    .
    .
    .
    USING  DATA  BYTE       ;Assume DD state is byte.
PROC2:
    .
    .
    .
    USING  DATA  ANY        ;Cancel assumption of DD state.
PROC3:
    .
    .
    .
```

In this example, three subroutines are called when DD is assumed to be word. The first CAL instruction's branch destination PROC1 has a DD state of word. This matches the branch source, so no warning will occur. The second CAL instruction's branch destination PROC2 has a DD state of byte. This does not match the branch source, a warning will occur. The third CAL instruction's branch destination PROC3 has no assumed DD state, so no warning will occur.

4.11 Addressing Modes

This section explains the syntax, use, and coding restrictions of OLMS-66K Series addressing modes.

The explanations of this chapter are based on the addressing modes of the top-end CPU core nX-8/500. With a few exceptions, all addressing modes usable with lower-end devices can be used with the nX-8/500 core. Names of addressing modes come from the “nX-8/500 Core Instruction Manual.”

■ Attention ■

• Not explained in this manual

This manual does not explain which instructions can use which addressing modes. Refer to the instruction manual of your target microcontroller for this information.

• Addressing modes not usable with lower-end devices

Some addressing modes usable with only the nX-8/500 core cannot be used with lower-end devices, but this manual will not point these out. When programming, please confirm the addressing modes usable with your target device by referring to the instruction manual.

• Names of addressing modes

This manual uses names of addressing modes consistent with the “nX-8/500 Core Instruction Manual.” Note that instruction manuals of lower-end devices and other documentation may use different names to refer to the same addressing modes.

The terms below are used in the explanations of this section.

Term	**Meaning**
ER*n*	Extended local register (ER0, ER1, ER2, ER3).
R*n*	Local register (R0, R1, R2, R3, R4, R5, R6, R7).
byte_immediate	General numeric expression -80H to +0FFH representing an immediate byte value.
word_immediate	General numeric expression -8000H to +0FFFFH representing an immediate word value.
byte_displacement	General numeric expression -40H to +3FH representing a byte displacement.
bit_displacement	General numeric expression -200H to +1FFH representing a bit displacement.
bit_offset	Constant numeric expression 0 to 7 representing a bit position.
shift_range	Constant numeric expression 1 to 4 representing a rotate/shift width.
data_address	General expression representing an address in data memory space.
data_base	General expression representing a number -8000H to +0FFFFH or an address in data memory space.
data_bit_address	General expression representing a bit address in data memory space.
data_bit_base	General expression representing a number -40000H to +07FFFFH or a bit address in data memory space.
code_address	General expression representing an address in program memory space.
code_base	General expression representing a number -8000H to +0FFFFH or an address in program memory space.

Expressions coded as instruction operands must conform to the following rules.

- Forward references to symbols are permitted in all addressing modes. However, RAM addressing optimizations are not applied to expressions that include forward references.
- General expressions can be used, with the following exceptions.
    - rotate width, shift width (*shift_range*)
    - bit offset (*bit_offset*)
- Addressing that calls for a numeric expression can also be coded as an address expression. In such cases, the physical segment address of the expression will be ignored, and only the offset address will be valid.

- Addressing that calls for an address expression can also be coded as a numeric expression. In such cases, the numeric expression will be handled as an address in the address space of the target addressing.

- Addressing that calls for an address expression has restrictions on the usage type of the expression. When an address expression with an unallowed usage type is coded, an error will occur.

The following sections explain the syntax and meaning of each addressing mode and provide examples of usage. The underlined portions of the examples illustrate the addressing under discussion.

4.11.1 Addressing Modes That Specify Numbers

4.11.1.1 Immediate Addressing

■ **Syntax** ■

Word: `#word_immediate`

Byte: `#byte_immediate`

■ **Description** ■

In this addressing mode, the number coded in the operand is itself the object accessed. The *word_immediate* and *byte_immediate* are general expressions that represent numbers.

■ **Additional Information** ■

RAS66K allows both unsigned and signed expressions for immediate addressing. The range of values for *word_immediate* is -8000H to 0FFFFH. The range of values for *byte_immediate* is -80H to 0FFH.

■ **Example** ■

```
MOV   ER0,#9000H
L   A,#-7FFFH

MOVB  R0,#0FFH
LB   A,#-7FH
```

4.11.1.2 Rotate/Shift Addressing

■ **Syntax** ■

```
shift_range
```

■ **Description** ■

This addressing mode specifies a rotate width or shift width. The *shift_range* is a constant expression that represents the rotate/shift width.

■ **Additional Information** ■

This addressing mode can be used with rotate instructions and shift instructions. The range of values for *shift_range* is 1 to 4.

■ **Example** ■

```
SLL    A,3
ROLB    A,2
```

4.11.2 RAM Addressing

RAM addressing is the addressing modes for specifying program variables in data memory space.

4.11.2.1 Register Addressing

Register addressing refers the contents of the registers themselves.

(1) Accumulator Addressing

■ Syntax ■

Word:	`A`
Byte:	`A`
Bit:	`A.bit_offset`

■ Description ■

This addressing mode accesses the accumulator contents for word instructions. It accesses the contents of the accumulator's lower byte for byte instructions and bit instructions. RAS66K determines whether the accumulator or the accumulator's lower byte is being accessed by instruction mnemonics.

■ Example ■

```
L   A,#1234H
LB  A,#12H
SB  A.4
```

(2) Control Register Addressing

■ Syntax ■

Word:	`SSP`	System stack pointer
	`LRB`	Local register base
	`PSW`	Program status word
Byte:	`PSWH`	Program status word high byte
	`PSWL`	Program status word low byte
Bit:	`PSWH.bit_offset`	Bit in program status word high byte
	`PSWL.bit_offset`	Bit in program status word low byte
	`C`	Carry flag

■ Description ■

This addressing mode accesses the contents of the various control registers.

■ Example ■

```
MOV   SSP,#STACK_TOP
MOV   LRB,ER0
CLR   PSW

CLRB  PSWH
INCB  PSWL

SB    PSWH.2
RB    PSWL.4
MB    C,BITVAR
```

(3) Pointing Register Addressing

■ Syntax ■

Word:	X1	Index register 1
	X2	Index register 2
	USP	User stack pointer
	DP	Data pointer
Byte:	X1L	Index register 1 low byte
	X2L	Index register 2 low byte
	USPL	User stack pointer low byte
	DPL	Data pointer low byte

■ Description ■

This addressing mode accesses the contents of the pointing registers.

■ Additional Information ■

The byte addressing X1L, X2L, USPL, and DPL can only be used with the DJNZ instruction. A byte instruction that can use DP is “JRNZ DP,address.” This instruction is provided for source compatibility between nX-8/100~nX-8/400 and nX-8/500. It is identical to the instruction “DJNZ DPL,address.”

■ Example ■

```
 L   A,X1
ST   A,X2
MOV  DP,#2000H
CLR  USP

DJNZ X1L,LOOP
DJNZ X2L,LOOP
DJNZ DPL,LOOP
DJNZ USPL,LOOP
JRNZ DP,LOOP
```

(4) Local Register Addressing

■ Syntax ■

Word:	`ERn`	Extended local register
Byte:	`Rn`	Local register
Bit:	`Rn.bit_offset`	Bit in local register

■ Description ■

This addressing mode access the contents of local registers.

■ Example ■

```
L   A,ER0
MOV   ER2,ER3

LB   A,R0
MOVB  R5,R3

MB   C,R4.7
SB   R2.0
```

(5) Register Sets

■ Syntax ■

`ER`	Local register set (ER0, ER1, ER2, ER3)
`PR`	Pointing register set (X1, X2, DP, USP)
`CR`	Control register set (A, LRB, PSW)

■ Description ■

This addressing mode indicates register sets. Instead of coded each register name to push or pop all registers in a set with a stack instruction (PUSHS, PUSHU, POPS, POPU), a register set name can be specified.

■ Example ■

```
PUSHS  ER
PUSHU  PR
POPS   CR
```

4.11.2.2 Page Addressing

(1) SFR Page Addressing

■ Syntax ■

```
Word:   sfr data_address
          data_address
Byte:   sfr data_address
          data_address
Bit:    sfr data_bit_address
          data_bit_address
```

■ Description ■

This addressing mode accesses addresses in the SFR area as word, byte, or bit data. The *data_address* is a general expression that represents a byte address in the SFR area. The *data_bit_address* is a general expression that represents a bit address in the SFR area.

RAS66K recognizes SFR page addressing by the “sfr” addressing specifier. If the “sfr” addressing specifier is omitted, then RAS66K will use SFR page addressing only when it recognizes the specified address value as being in the SFR area.

■ Additional Information ■

The range of values of *data_address* is 0 to 0FFH. Its usage type can be DATA, NONE, or NUMBER.

The range of values of *data_bit_address* is 0.0 to 0FFH.7. Its usage type can be BIT, NONE, or NUMBER.

■ Example ■

```
  L    A,sfr P0
L    A,P0

LB   A,sfr P0
LB   A,P0

SB   sfr P0.3
SB   P0.3
```

(2) Fixed Page Addressing

■ Syntax ■

```
Word:   fix data_address
          data_address
Byte:   fix data_address
          data_address
Bit:    fix data_bit_address
          data_bit_address
```

■ Description ■

This addressing mode accesses addresses in the fixed page area as word, byte, or bit data. The *data_address* is a general expression that represents a byte address in the fixed page area. The *data_bit_address* is a general expression that represents a bit address in the fixed page area.

RAS66K recognizes fixed page addressing by the “fix” addressing specifier. If the “fix” addressing specifier is omitted, then RAS66K will use fixed page addressing only when it recognizes the specified address value as being in the fixed page area. In addition, if a bit address is a value 2C0H.0-2FFH.7, then RAS66K will interpret it as sbafix addressing.

■ Additional Information ■

The range of values of *data_address* is 200H to 2FFH. Its usage type can be DATA, NONE, or NUMBER.

The range of values of *data_bit_address* is 200H.0 to 2FFH.7. Its usage type can be BIT, NONE, or NUMBER.

■ Example ■

```
L    A,fix FIXVAR
L    A,FIXVAR

LB   A,fix FIXVAR
LB   A,FIXVAR

SB   fix FIXVAR.4
SB   FIXVAR.4
```

(3) Current Page Addressing

■ Syntax ■

```
Word:   off data_address
        \ data_address
Byte:   off data_address
        \ data_address
Bit:    off data_bit_address
        \ data_bit_address
```

■ Description ■

This addressing mode accesses addresses in the current page area as word, byte, or bit data.

Specify the desired address itself as the operand. In other words, the *data_address* is a general expression that represents a byte address in the current page area. The *data_bit_address* is a general expression that represents a bit address in the current page area.

RAS66K recognizes current page addressing by the "off" or "\" addressing specifier. The "off" and "\" addressing specifiers have the same meaning for word and byte instructions. For bit instructions, if the addressing specifier is "\" and the *data_bit_address* value is an address in the SBA area, then RAS66K will interpret it as sbaoff addressing.

■ Additional Information ■

The *data_address* is an address in DATA address space. Its usage type can be DATA, EDATA, NONE, or NUMBER.

The *data_bit_address* is an address in BIT address space. Its usage type can be BIT, EBIT, NONE, or NUMBER.

The programmer is responsible for managing the current page such that current page addresses specified in operands are actually in the currently selected current page. To assist in this task, RAS66K provides the USING PAGE directive for checking page numbers of current page addressing.

This addressing mode can also be thought of as specifying an offset (0-0FFH) within the current page. Machine code will be generated normally for this use as well, but RAS66K will perform SFR access attribute checks on addresses that it thinks in the SFR area, and may generate warnings.

■ Example ■

```
L    A,off VAR
L    A,\VAR

LB   A,off VAR
LB   A,\VAR

SB   off VAR.4
SB   \VAR.4
```

(4) Fixed Page SBA Area Addressing

■ Syntax ■

```
Bit:    sbafix data_bit_address
               data_bit_address
```

■ Description ■

This addressing mode accesses addresses in the SBA area of the fixed page area (2C0H-2FFH) as bit data. The *data_bit_address* is a general expression that represents a bit address in the SBA area of the fixed page area.

RAS66K recognizes fixed page SBA addressing by the “sbafix” addressing specifier. If the “sbafix” addressing specifier is omitted, then RAS66K will use fixed page SBA addressing only when it recognizes the specified address value as being in the fixed page SBA area.

■ Additional Information ■

The range of values of *data_bit_address* is 2C0H.0 to 2FFH.7. Its usage type can be BIT, NONE, or NUMBER.

■ Example ■

```
SB    sbafix 2C0H.0
SB    2C0H.0
```

(5) Current Page SBA Area Addressing

■ Syntax ■

```
Bit:    sbaoff data_bit_address
        \    data_bit_address
```

This addressing mode accesses addresses in the SBA area of the current page area (xxC0H-xxFFH) as bit data. The *data_bit_address* is a general expression that represents a bit address in the SBA area of the current page area.

RAS66K recognizes current page SBA addressing by the “sbaoff” or “\” addressing specifier. The “sbaoff” addressing specifier always means current page SBA area addressing. However, the “\” addressing specifier means current page SBA area addressing (sbaoff) only if the address value is within the range of the SBA area, and means current page addressing (off) if the address value is outside the range of the SBA area.

■ Additional Information ■

The range of values of *data_bit_address* is xxC0H.0 to xxFFH.7. Its usage type can be BIT, EBIT, NONE, or NUMBER.

The programmer is responsible for managing the current page such that current page addresses specified in operands are actually in the currently selected current page. To assist in this task, RAS66K provides the USING PAGE directive for checking page numbers of current page addressing.

This addressing mode can also be thought of as specifying an offset (0-0FFH) within the current page. Machine code will be generated normally for this use as well, but RAS66K will perform SFR access attribute checks on addresses that it thinks in the SFR area, and may generate warnings.

■ Example ■

```
    SB   sbaoff 12C0H.0
    SB   \12C0H.0
```

4.11.2.3 Direct Addressing

(1) Direct Data Addressing

■ Syntax ■

Word: `dir data_address`
`data_address`
Byte: `dir data_address`
`data_address`
Bit: `dir data_bit_address`
`data_bit_address`

■ Description ■

This addressing mode directly specifies any address of a physical segment (64K bytes) in data memory space. Specified addresses can be accessed as word, byte, or bit data. The *data_address* is a general expression that represents a byte address in data memory space. The *data_bit_address* is a general expression that represents a bit address in data memory space.

RAS66K recognizes direct data addressing by the “dir” addressing specifier. If the “dir” addressing specifier is omitted, then RAS66K will use direct data addressing when it recognizes the specified address value as not being in the SFR area or fixed page area.

■ Additional Information ■

The value of *data_address* is an address in DATA address space. Its usage type can be DATA, EDATA, NONE, or NUMBER.

The value of *data_bit_address* is an address in BIT address space. Its usage type can be BIT, EBIT, NONE, or NUMBER.

■ Example ■

```
L    A,dir GLDATA
L    A,GLDATA

LB   A,dir GLDATA
LB   A,GLDATA

SB   dir GLDATA.4
SB   GLDATA.4
```

4.11.2.4 Pointing Register Indirect Addressing

(1) DP/X1 Indirect Addressing

■ Syntax ■

Word:	`[X1]`
	`[DP]`
Byte:	`[X1]`
	`[DP]`
Bit:	`[X1].bit_offset`
	`[DP].bit_offset`

■ Description ■

This addressing mode specifies an address of data memory space with the contents of a pointing register. Data at specified addresses can be accessed as word, byte, or bit data.

■ Example ■

```
 L   A,[DP]
MOV  ER0,[X1]

LB   A,[DP]
MOVB R0,[X1]

SB  [DP].4
MB   C,[X1].4
```

(2) Indirect Addressing With Post-Increment

■ Syntax ■

Word:	`[DP+]`
	`[X1+]`
	`[X2+]`
Byte:	`[DP+]`
	`[X1+]`
	`[X2+]`
Bit:	`[DP+].bit_offset`

■ Description ■

This addressing mode specifies an address of data memory space with the contents of a pointing register. Data at specified addresses can be accessed as word, byte, or bit data.

After the address is accessed, the register contents are incremented by 2 for word data and by 1 for byte and bit data.

■ Additional Information ■

[X1+] and [X2+] can only be used with string instructions.

■ Example ■

```
L      A,[DP+]
SMOV     [X2+],[X1+],R0

LB     A,[DP+]
SMOVB    [X2+],[X1+],R0

SB     [DP+].2
```

(3) Indirect Addressing With Post-Decrement

■ Syntax ■

```
Word:   [DP-]
        [X1-]
        [X2-]
Byte:   [DP-]
        [X1-]
        [X2-]
Bit:    [DP-].bit_offset
```

■ Description ■

This addressing mode specifies an address of data memory space with the contents of a pointing register. Data at specified addresses can be accessed as word, byte, or bit data.

After the address is accessed, the register contents are decremented by 2 for word data and by 1 for byte and bit data.

■ Additional Information ■

[X1-] and [X2-] can only be used with string instructions.

■ Example ■

```
L      A,[DP-]
SMOV     [X2-],[X1-],R0

LB     A,[DP+]
SMOVB    [X2-],[X1-],R0

SB     [DP-].2
```

(4) DP/USP Indirect Addressing With 7-Bit Displacement

■ Syntax ■

```
Word:   byte_displacement [DP]
        byte_displacement [USP]
Byte:   byte_displacement [DP]
        byte_displacement [USP]
Bit:    byte_displacement [DP].bit_offset
        byte_displacement [USP].bit_offset
        bit_displacement [DP]
        bit_displacement [USP]
```

■ Description ■

This addressing mode specifies an address of data memory space with the contents of a pointing register as a base, and a 7-bit unsigned displacement (-64 to +63). Data at specified addresses can be accessed as word, byte, or bit data.

The *byte_displacement* is a general expression that represents a displacement in byte units. The *bit_displacement* is a general expression that represents a displacement in bit units.

Two methods are provided for expressing bits. One is to express the displacement as a byte displacement (*byte_displacement*) with a bit offset (*bit_offset*), and the other is to express it as a bit displacement (*bit_displacement*).

■ Additional Information ■

The range of values of *byte_displacement* is -64 to +63. The range of values of *bit_displacement* is -512 to +511.

nX-8/100~400 can only use DP. There is one more displacement bit, allowing a range -128 to +127.

■ Example ■

```
L     A,12[DP]
MOV   ER0,-8[USP]

LB    A,12[DP]
MOVB  R0,-8[USP]

SB    12[DP].0
RB    80H[DP]
MB    C,-8[USP].2
MB    -32H[USP],C
```

(5) X1/X2 Indirect Addressing With 16-Bit Base

■ Syntax ■

Word:	`data_base [X1]`
	`data_base [X2]`
Byte:	`data_base [X1]`
	`data_base [X2]`
Bit:	`data_base [X1].bit_offset`
	`data_base [X2].bit_offset`
	`data_bit_base [X1]`
	`data_bit_base [X2]`

■ Description ■

This addressing mode specifies an address of data memory space with the contents of an index register (X1 or X2) added to the address specified as a base. Data at specified addresses can be accessed as word, byte, or bit data.

The *data_base* is a general expression that represents a byte address in data memory space. The *data_bit_base* is a general expression that represents a bit address in data memory space.

Two methods are provided for expressing bits. One is to express the base address as a byte base address (*data_base*) with a bit offset (*bit_offset*), and the other is to express it as a bit base address (*data_bit_base*).

This addressing mode can also be thought of as if the index register contents are a base address with a 16-bit displacement. This is why *data_base* is also permitted to be specified as a numeric expression -8000H to -1, and *data_bit_base* is also permitted to specified as a numeric expression -40000H to -1.

■ Additional Information ■

The range of values of *data_base* is -8000H to +0FFFFH. Its usage type can be DATA, EDATA, NONE, or NUMBER, but it can be NUMBER only when the value is negative.

The range of values of *data_bit_base* is -40000H to +7FFFFH. Its usage type can be BIT, EBIT, NONE, or NUMBER, but it can be NUMBER only when the value is negative.

■ **Example** ■

```
L    A,1234H[X1]
MOV  ER0,-200H[X2]

LB   A,1234H[X1]
MOVB R0,-200H[X2]

SB   1000H[X1].2
SB   8002H[X1]
MB   C,-10H[X2].1
MB   C,-7EH[X2]
```

(6) X1 Indirect Addressing With 8-Bit Register Displacement

■ **Syntax** ■

Word:	`[X1+R0]`
	`[X1+A]`
Byte:	`[X1+R0]`
	`[X1+A]`
Bit:	`[X1+R0].bit_offset`
	`[X1+A].bit_offset`

■ **Description** ■

This addressing mode specifies an address of data memory space generated by adding the contents of index register X1 as a base to the contents of the accumulator’s low byte or of local register R0. Data at specified addresses can be accessed as word, byte, or bit data.

■ **Example** ■

```
MOV   ER0,[X1+A]
MOV   ER1,[X1+R0]

MOVB  R0,[X1+A]
MOVB  R1,[X1+R0]

SB    [X1+A].3
RB    [X1+R0].4
```

4.11.3 Table Data Addressing

Table data addressing is used to access data in program memory space.

4.11.3.1 Direct Addressing

(1) Direct Table Addressing

■ Syntax ■

Word:	`code_address`
Byte:	`code_address`

■ Description ■

This addressing mode accesses word or byte data at a specified address in program memory space. The *code_address* is a general expression that represents an address in program memory space.

■ Additional Information ■

The value of *code_address* is an address in CODE address space. Its usage type can be CODE, NONE, or NUMBER.

■ Example ■

```
    CSEG
CodeTable:
   DB     10H,20H,30H,40H,50H,60H
   .
   .
   .
   LC    A,CodeTable
   CMPC  A,CodeTable

   LCB   A,CodeTable
   CMPCB A,CodeTable
```

4.11.3.2 Indirect Addressing

(1) RAM Addressing Indirect Table Addressing

■ Syntax ■

Word:	`[word RAM addressing]`
Byte:	`[word RAM addressing]`

■ **Description** ■

This addressing mode accesses addresses in program memory space pointed to by the contents of data memory specified by word RAM addressing (described in Section 4.11.2, "RAM Addressing."). These addresses can be accessed as word or byte data.

■ **Example** ■

```
LC    A,[ER0]
LC    A,[X1]
LC    A,[fix VAR]
LC    A,[2000H[X1]]
LC    A,[[X1+R0]]



LCB   A,[ER2]
LCB   A,[X2]
LCB   A,[off VAR]
LCB   A,[2000H[X2]]
LCB   A,[[X1+A]]
```

(2) RAM Addressing Indirect Addressing With 16-Bit Base

■ **Syntax** ■

Word:	`code_base [word RAM addressing]`
Byte:	`code_base [word RAM addressing]`

■ **Description** ■

This addressing mode accesses addresses in program memory space generated by adding *code_base* as a base to the contents of data memory specified by word RAM addressing (described in Section 4.11.2, "RAM Addressing."). These addresses can be accessed as word or byte data. The *code_base* is a general expression that represents an address in program memory space.

This addressing mode can also be thought of as if the word RAM addressing contents are a base address with the 16-bit displacement of *code_base*. This is why *code_base* is also permitted to be specified as a numeric expression -8000H to -1.

■ **Additional Information** ■

The range of values of *code_base* is -8000H to +0FFFFH. Its usage type can be CODE, NONE, or NUMBER, but it can be NUMBER only when the value is negative.

■ **Example** ■

```
   CSEG
RomTable:
   DB     10H,20H,30H,40H,50H,60H
   .
   .
   .
   LC   A,2000H[ER0]
   LC   A,RomTable[X1]
   LC   A,-10H[2000H[X1]]

   LCB   A,1000H[ER2]
   LCB   A,RomTable[X2]
   LCB   A,-20H[2000H[X2]]
```

4.11.4 Program Code Addressing

Program code addressing represents branch destinations of jump and call instructions.

4.11.4.1 Direct Addressing

(1) Near Code Addressing

■ Syntax ■

```
    code_address
```

■ Description ■

This addressing mode directly specifies the branch destination address of a J instruction or CAL instruction. It specifies a branch destination address within the physical segment selected by the current CSR. The *code_address* is a general expression that represents an address in program memory space.

■ Additional Information ■

The value of *code_address* must be within the range of addresses of program memory space. Its usage type can be CODE, NONE, or NUMBER.

■ Example ■

```
    J  NEXT
   CAL FUNC
   .
   .
   .
NEXT:
   .
   .
   .
FUNC:
   .
   .
   .
   RT
```

(2) Far Code Addressing

■ Syntax ■

```
    code_address
```

■ Description ■

This addressing mode directly specifies the branch destination address of a FJ instruction or FCAL instruction. The *code_address* is a general expression that represents an address in program memory space. The *code_address* can specify an address of any physical segment of program memory space.

■ Additional Information ■

The value of *code_address* must be within the range of addresses of program memory space. Its usage type can be CODE, NONE, or NUMBER.

If the usage type of *code_address* is NUMBER, then RAS66K will assume a branch to the same physical segment that the instruction is in.

■ Example ■

```
    CSEG #1
   FJ  NEXT
   FCAL FUNC
   .
   .
   .
   CSEG #2
NEXT:
   .
   .
   .
   CSEG #3
FUNC:
   .
   .
   .
   FRT
```

4.11.4.2 Relative Addressing

(1) Relative Code Addressing

■ Syntax ■

```
code_address
```

■ Description ■

This addressing mode specifies the branch destination address of an SJ instruction or a conditional branch instruction. The *code_address* is a general expression that represents an address in program memory space.

■ Additional Information ■

The displacement between *code_address* and the address of the next instruction must be -128 to +127. The usage type of *code_address* can be CODE, NONE, or NUMBER.

■ Example ■

```
 SJ  LOOP1
JC   NE,NEXT
```

4.11.4.3 Special Code Addressing For Particular Instructions

(1) ACAL Code Addressing

■ Syntax ■

```
    code_address
```

■ Description ■

This addressing mode specifies the branch destination of an ACAL instruction. The *code_address* is a general expression that represents an address in the ACAL area of program memory space.

■ Additional Information ■

The value of *code_address* is the address range of the ACAL area, 1000H to 1AFFH. Its usage type can be CODE, NONE, or NUMBER.

■ Example ■

```
   ACAL   AcalFunc
   .
   .
   .
   CSEG   AT 1000H   ;ACAL area
AcalFunc:
   .
   .
   .
   RT
```

(2) VCAL Code Addressing

■ Syntax ■

```
    code_address
```

■ Description ■

This addressing mode specifies an address in the VCAL table area. The *code_address* is a general expression that represents an address in the VCAL table of program memory space.

■ Additional Information ■

The value of *code_address* is an even address in the VCAL table area. Its usage type can be CODE, NONE, or NUMBER. The physical segment address of *code_address* must be 0.

■ Example ■

```
    TYPE   (M66507)
    .
    .
    .
    CSEG   AT 4AH            ;VCAL table area
VcalVct00:
    DW     VcalFunc00
VcalVct01:
    DW     VcalFunc01
    .
    .
    .
    VCAL   VcalVct00
    VCAL   VcalVct01
    .
    .
    .
VcalFunc00:
    ;
    ;
    RT
VcalFunc01:
    ;
    ;
    RT
```

4.11.4.4 Indirect Addressing

(1) RAM Addressing Indirect Code Addressing

■ Syntax ■

```
   [word RAM addressing]
```

■ Description ■

This addressing mode indicates branches to addresses in program memory space which are pointed to by the contents of data memory specified by word RAM addressing (described in Section 4.11.2, "RAM Addressing.").

This addressing can be used with J instructions and CAL instructions.

■ Example ■

```
   J  [X1]
  J  [1234H[X2]]
  CAL [A]
```

4.11.5 ROM Window Addressing

■ **Syntax** ■

Same as Section 4.11.2, “RAM Addressing”

■ **Description** ■

This addressing mode accesses the ROM window area of program memory space using RAM addressing (except for register direct). Thus its syntax is exactly the same as for RAM addressing. However, instead of specifying an address in data memory space, this addressing mode specifies an address in program memory space.

■ **Additional Information** ■

ROM window addressing is a fixed function of the nX-8/500.

Data in the ROM window area can be read but not written.

■ **Example** ■

```
      TYPE  (M66507)
     WINDOW 4000H,4FFFH
     .
     .
     .
     CSEG   AT 4000H        ;ROM Window Area
RomWinTable:
     DW     1234H
     DW     0FF00H
     .
     .
     .
     CSEG
     ADD   A,RomWinTable
     XOR   A,RomWinTable[X1]
```

4.11.6 Addressing For nX-8/100 to nX-8/400

4.11.6.1 Zero Page Addressing

■ **Syntax** ■

Word:	data_address
Byte:	data_address
Bit:	data_bit_address

■ Description ■

This addressing mode accesses word, byte, or bit data at an address in the zero page area (00H-0FFH). The *data_address* is a general expression that represents a byte address in the zero page area. The *data_bit_address* is a general expression that represents a bit address in the zero page area.

■ Additional Information ■

The range of values of *data_address* is 0 to 0FFH. Its usage type can be DATA, NONE, or NUMBER.

The range of values of *data_bit_address* is 0.0 to 0FFH.7. Its usage type can be BIT, NONE, or NUMBER.

■ Example ■

```
MOV   ER0,80H
L   A,90H

MOVB  R1,80H
LB   A,90H

SB   80H.1
RB   90H.2
```

4.11.6.2 USP Indirect Addressing With Pre-Increment

■ Syntax ■

Word: `[+USP]`

■ Description ■

This addressing mode specifies an address in data memory space with the contents of USP incremented by 2. The specified address accesses word data.

■ Additional Information ■

This is a fixed addressing mode of the nX-8/300 CPU core. It can only be used with word-based calculation instructions (ADD, ADC, SUB, SBC, AND, OR, XOR).

■ **Example** ■

```
ADD    A,[+USP]
ADC    A,[+USP]
SUB    A,[+USP]
SBC    A,[+USP]
AND    A,[+USP]
OR     A,[+USP]
XOR    A,[+USP]
```

4.11.7 Optimization Of Addressing

The OLMS-66K Series provides many addressing modes for directly referring RAM addresses. Usually a programmer will code addressing specifiers to specify particular addressing modes. However, it is bothersome to the programmer to always stay aware of addressing types during programming. Therefore RAS66K also has a function for determining the optimal addressing for specified address values.

4.11.7.1 Optimization Of RAM Address Specifications Without Addressing Specifiers

■ Syntax ■

Word:	`data_address`
Byte:	`data_address`
Bit:	`data_bit_address`

■ Description ■

RAS66K determines the addressing of a RAM address specification without an address specifier in the following order.

(1) For a byte or bit instruction, RAS66K will use control register addressing (PSWL, PSWH) if the address value matches the address of PSWL or PSWH (4H, 5H).

(2) If the address value is an address in the SFR area, then RAS66K will use SFR page addressing (sfr address).

(3) If the address value is an address in the fixed page area, then RAS66K will use fixed page addressing (fix address). However, for a bit instruction, RAS66K will use fixed page SBA area addressing (sbafix address) if the address value is an address in the SBA area.

(4) If the address does not fit one of the above conditions, then RAS66K will use direct data addressing (dir address).

However, if the specified expression includes a forward reference, then RAS66K will not perform optimization processing, but will instead use the addressing mode that requires the longest code. For nX-8/100~400 CPU cores, optimization for control register addressing is performed, but everything else will be zero page addressing.

Examples of optimizations of RAM address specifications without addressing specifiers are shown below.

■ **Example** ■

```
    LB   A,4H        ;PSWL
    LB   A,5H        ;PSWH
    LB   A,20H       ;sfr 20H
    LB   A,200H           ;fix 200H
    LB   A,1200H     ;dir 1200H

    SB   4H.0        ;PSWL.0
    SB   5H.0        ;PSWH.0
    SB   20H.0       ;sfr 20H
    SB   200H.0           ;fix 200H
    SB   2C0H.0           ;sbafix 2C0H
    SB   1200H.0     ;dir 1200H
```

In this example, the comments show how RAS66K actually determines the addressing.

This example shows constant expressions, but RAS66K can also determine fixed page addressing (fix address) and fixed page SBA area addressing (sbafix address) for relocatable expressions. For this to happen, the relocatable symbols included in expressions need to be declared in advance to be addresses in the fixed page area or SBA area when the symbols are defined. Examples of optimizations of relocatable expressions are shown below.

■ **Example** ■

```
SEGSYM   SEGMENT  DATA  WORD SBAFIX
     RSEG     SEGMENT
RELSYM:  DS      2
COMSYM   SEGMENT  DATA  2H SBAFIX
EXTRN    DATA SBAFIX : EXTSYM
     .
     .
     .
     CSEG
     LB   A,SEGSYM ;fix SEGSYM
     LB   A,RELSYM ;fix RELSYM
     LB   A,COMSYM ;fix COMSYM
     LB   A,EXTSYM ;fix EXTSYM

     SB   SEGSYM.0 ;sbafix SEGSYM
     SB   RELSYM.0 ;sbafix RELSYM
     SB   COMSYM.0 ;sbafix COMSYM
     SB   EXTSYM.0 ;sbafix EXTSYM
```

In the above example, each relocatable symbol is specified with the SBAFIX keyword when defined. The SBAFIX specification declares that the address is in the SBA area of the fixed page area.

4.11.7.2 Optimization Of RAM Address Specifications With The Addressing Specifier \

■ **Syntax** ■

Bit: \ data_bit_address

■ **Description** ■

RAM address specifications with the addressing specifier \ are optimized only when used with bit instructions. For word and byte instructions, they will be the same as "off" addressing.

RAS66K determines the addressing in the following order.

(1) For a bit instruction, RAS66K will use current page SBA area addressing (sbaoff address) if the address value is an address in the SBA area.

(2) If the address does not fit the above condition, then RAS66K will use current page addressing (off address).

However, if the specified expression includes a forward reference, then RAS66K will always use current page addressing (off address).

For RAS66K to determine current page SBA area addressing (sbaoff address) for a relocatable expression, the relocatable symbols included in expressions need to be declared in advance to be addresses in the SBA area when the symbols are defined.

Examples of optimizations of RAM address specifications with the addressing specifier \ are shown below.

■ **Example** ■

```
SEGSYM    SEGMENT  BIT SBA
      RSEG     SEGSYM
RELSYM:   DBIT     4
COMSYM    COMM     BIT 2 SBA
EXTRN     BIT SBA:EXTSYM
      .
      .
      .
      CSEG
      SB  \ 1000H  ;off 1000H
      SB  \ 10C0H  ;sbaoff 10C0H
      SB  \ SEGSYM ;sbaoff SEGSYM
      SB  \ RELSYM ;sbaoff RELSYM
      SB  \ COMSYM ;sbaoff COMSYM
      SB  \ EXTSYM ;sbaoff EXTSYM
```

4.12 Directives

Directives are instructions provided uniquely by RAS66K. While microcontroller instructions determine program operation, directives manage program structure, control RAS66K operation, and specify output files.

This section describes the syntax and functions of all directives provided by RAS66K. Comment fields are omitted from descriptions of directive syntax. The syntax of directives that can specify labels is shown starting with [label:]. Directives that do not show this syntax cannot specify labels.

4.12.1 DCL File Specification (TYPE)

■ Syntax ■

```
    TYPE (dcl_name)
```

■ Description ■

The TYPE directive specifies the DCL file name corresponding to the target microcontroller. RAS66K will read information from the DCL file that has *dcl_name* as a base name and ".DCL" as an extension. The microcontroller name and DCL file base name are similar, but they differ in that microcontroller names start with "MSM" and DCL file base names start with "M." For example, if the microcontroller name is "MSM66507," then specify "M66507" in the TYPE directive.

The DCL file search will proceed as follows.

(1) Current directory
(2) Directory in which RAS66K.EXE exists
(3) Directory specified in environment variable DCL

RAS66K reads DCL file contents before assembly processing. It will read to the end of the DCL file even if the file contents contain errors. RAS66K will then display all generated DCL errors and terminate. If the DCL file read is okay, then RAS66K will continue and assemble the source file.

■ Additional Information ■

A TYPE directive must be specified in the program. If no TYPE directive is specified or if the specified DCL file cannot be found, then RAS66K will terminate without assembling the source file.

Specify the TYPE directive at the start of the program. For details on coding, refer to Section 4.4.1.5, "Code Position Restrictions."

Refer to the DCL66K.DOC file for details on the contents of DCL files.

■ Example ■

```
/*
    TEST PROGRAM
*/
    TYPE(M66507)

    CSEG AT 0H
    DW   START
    .
    .
    .
START:
    .
    .
    .
```

In this example, the target microcontroller is MSM66507. RAS66K will read the DCL file M66507.DCL.

4.12.2 Memory Model Specification (MODEL)

■ Syntax ■

```
   MODEL memory_model
```

■ Description ■

The MODEL directive specifies the memory model used.

When the microcontrollers with the nX-8/500 CPU core and multiple physical segments, the MODEL directive is specified to inform RAS66K of the memory model. The *memory_model* is one of the following.

memory_model	**Memory Model**
SMALL (default)	SMALL memory model
COMPACT	COMPACT memory model
MEDIUM	MEDIUM memory model
LARGE	LARGE memory model

The number of physical segments usable by a program is determined by the microcontroller memory configuration defined in the DCL file and by the memory model set with the MODEL directive.

The MODEL directive may be omitted. If the MODEL directive is not specified, the SMALL memory model will be set.

■ Additional Information ■

The FJ instruction and FCAL instruction cannot be used unless the memory model is MEDIUM or LARGE.

The MODEL directive informs RAS66K of the *memory model*, so it does not generate object code that sets the memory model. To actually set the *memory model*, use a microcontroller instruction.

Specify the MODEL directive at the start of the program. For details on coding, refer to Section 4.4.1.5, “Code Position Restrictions.”

Refer to Section 3.2.4.3, “Memory Models,” for details on memory models.

■ Example ■

```
    .
    .
    .
    MODEL   LARGE

    DSEG    #3  AT 1000H
    .
    .
    .
    CSEG    #5  AT 2000H
FUNC:
    .
    .
    .
    CSEG    #6  AT 1000H
    FCAL    FUNC
    .
    .
    .
```

This example sets the LARGE memory model.

4.12.3 COMMON Area Specification (COMMON)

■ Syntax ■

```
COMMON bcb_value
```

■ Description ■

In microcontrollers that have multiple physical segments in data memory space, an area exists that is common to all physical segments, called the COMMON area. There are four possible end addresses for the COMMON area. The programmer selects one of these by setting the BCB of the PSW to a value 0-3.

The COMMON directive informs RAS66K of the number in BCB of the PSW for the COMMON area's end address. The *bcb_value* must be a constant expression that does not include forward references. Set the *bcb_value* to the value 0-3 set in the BCB.

The COMMON directive may be omitted. If the COMMON directive is not specified, then RAS66K will assume the BCB value is 0.

■ Additional Information ■

The COMMON directive informs RAS66K of the end address of the COMMON area, so it does not generate object code that sets BCB. To actually set BCB, use a microcontroller instruction.

RAS66K memory management assumes that the size of the COMMON area (the value of BCB) will not be changed during the program. If creating a program that changes the value of BCB, then the programmer needs to manage memory himself.

Specify the COMMON directive at the start of the program. For details on coding, refer to Section 4.4.1.5, "Code Position Restrictions."

■ Example ■

```
TYPE   (M66507)

COMMON 2
```

4.12.4 ROM Window Area Specification (WINDOW)

■ **Syntax** ■

```
   WINDOW start_address ,end_address
```

■ **Description** ■

The WINDOW directive specifies the start address and end address of the ROM window area.

The ROM window function is available when using a microcontroller with the nX-8/500 CPU core. The ROM window function allocates a certain area in data memory space to the same address range in program memory space. In order to use the ROM window function, values representing the address range of the ROM window area must be written in the SFR's ROMWIN register. In order to inform RAS66K how the ROM window function is being used, specify the start address and end address of the ROM window area as operands of the WINDOW directive.

The *start_address* represents the start address of the ROM window area. It must be a constant expression that does not include forward references. The value of *start_address* must be 1000H or higher, and its lower 12 bits must be 000H.

The *end_address* represents the end address of the ROM window area. It must be a constant expression that does not include forward references. The value of *end_address* must be 1000H or higher, and must be greater than *start_address*. Its lower 12 bits must be FFFH.

If the WINDOW directive is omitted, then RAS66K will assume that no ROM window area exists.

■ **Additional Information** ■

The WINDOW directive informs RAS66K of the ROM window area range, so it does not generate object code that sets ROMWIN register. To actually set the ROMWIN register, use a microcontroller instruction.

RAS66K memory management assumes that the range of the ROM window area (the value of the ROMWIN register) will not be changed during the program. If creating a program that changes the value of the ROMWIN register, then the programmer needs to manage memory himself.

Specify the WINDOW directive at the start of the program. For details on coding, refer to Section 4.4.1.5, "Code Position Restrictions."

■ **Example** ■

```
     TYPE    (M66507)

WIN_START EQU     5000H
WIN_END   EQU     6FFFH
WIN_REG   EQU     ((WIN_END&0F000H)>>8)|(WIN_START>>12)
     WINDOW  WIN_START,WIN_END
     .
     .
     .
     MOVB    ROMWIN,#WIN_REG
     .
     .
     .
     CSEG    AT WIN_START
     DW      10H,20H,30H,40H
     .
     .
     .
```

This example uses 5000H to 6FFFH as the ROM window area.

4.12.5 Local Symbol Definition (EQU, SET)

■ **Syntax** ■

```
   symbol EQU simple_expression
   symbol SET simple_expression
```

■ **Description** ■

The EQU directive and SET directive define a local symbol. The symbol specifies the symbol to be defined. The *simple_expression* specifies a simple expression that does not include forward references.

Symbols defined with the EQU and SET directives will be given the attribute and the value of *simple_expression*. In other words, if *simple_expression* is a constant expression, then the defined symbol will become an absolute symbol. If *simple_expression* is a simple relocatable expression, then the defined symbol will become a simple relocatable symbol. If *simple_expression* is a numeric expression, then the symbol's usage type will be NUMBER. If *simple_expression* is an address expression, then the symbol will be given the address characteristics of *simple_expression*.

A symbol defined with the EQU directive cannot be defined again within the same program. A symbol defined with the SET directive can be defined again any number of times using the SET directive.

■ Additional Information ■

The *simple_expression* cannot be specified as an address constant (more precisely, as an expression of type NONE). This is because RAS66K does not allow a symbol to be defined without a clear address space for it.

If a symbol defined with the SET directive is used before its first definition, then the symbol value will be that of the last definition. RAS66K will issue a warning in this case and output the information from the last definition to the symbol list. If the symbol is also declared public with the PUBLIC directive, RAS66K will also output the information from the last definition to the object file.

■ Example 1 ■

```
SEGSYM  SEGMENT DATA 2
     RSEG    SEGSYM
BUF1:   DS     4

BASE    EQU     10H
BUFSIZE EQU      4H
VALUE   EQU      BASE+BUFSIZE
BUFX    EQU      BUF1+BUFSIZE
```

In this example, four local symbols are defined using the EQU directive. BASE, BUFSIZE, and VALUE will become absolute symbols with usage type NUMBER. BUFX will become a simple relocatable symbol with usage type DATA.

■ Example 2 ■

```
    MOV ER0,#SETSYM
    .
    .
    .
SETSYM  SET 100H
    MOV ER0,#SETSYM
    .
    .
    .
SETSYM  SET 200H
    MOV ER0,#SETSYM
```

In this example, the absolute symbol SETSYM with usage type NUMBER is defined using the SET directive. In the MOV instruction immediately after the first SET directive, SETSYM’s value is 100H, but in the MOV instruction immediately after the second SET directive, SETSYM’s value is 200H. In the first MOV instruction, SETSYM’s value will be the 200H specified with the last SET directive. This instruction statement’s reference to SETSYM is before the first SET directive, so it will cause a warning.

4.12.6 Definition of Local Symbols That Represent Addresses (CODE, CBIT, DATA, BIT, EDATA, EBIT)

The directives explained in this section define local symbols that represent addresses in the various address spaces. The defined symbols will have the usage type corresponding to their directive.

■ Syntax ■

```
    symbol CODE simple_expression
```

■ Description ■

The CODE directive defines a local symbol that represents a byte address in CODE address space. The *symbol* specifies the local symbol to be defined. The *simple_expression* represents an address in CODE address space. It is specified as a simple expression that does not include forward references.

If *simple_expression* is a constant expression, then symbol will be an absolute symbol. If *simple_expression* is a simple relocatable expression, then symbol will be a simple relocatable symbol. The symbol will be given the address value of *simple_expression* and usage type CODE.

■ Additional Information ■

The usage type of *simple_expression* must be CODE, NONE, or NUMBER. Other usage types will cause an error. If the usage type of *simple_expression* is NUMBER, then the physical segment address of symbol will be 0.

■ Example ■

```
CODE_SYM1  CODE   1000H

CODE_SYM2  CODE   2:2000H

      CSEG   #3 AT 3000H
LABEL:
CODE_SYM3  CODE   LABEL+100H
```

In this example, three absolute symbols with usage type CODE are defined using the CODE directive (CODE_SYM1, CODE_SYM2, CODE_SYM3). CODE_SYM1 will be a symbol representing offset address 1000H in physical segment 0. CODE_SYM2 will be a symbol representing offset address 2000H in physical segment 2. CODE_SYM3 will be a symbol representing offset address 3100H in physical segment 3.

■ Syntax ■

```
   symbol CBIT simple_expression
```

■ Description ■

The CBIT directive defines a local symbol that represents a bit address in CODE address space. The *symbol* specifies the local symbol to be defined. The *simple_expression* represents a bit address in CODE address space. It is specified as a simple expression that does not include forward references.

If *simple_expression* is a constant expression, then symbol will be an absolute symbol. If *simple_expression* is a simple relocatable expression, then symbol will be a simple relocatable symbol. The symbol will be given the address value of *simple_expression* and usage type CBIT.

■ Additional Information ■

The usage type of *simple_expression* must be CBIT, NONE, or NUMBER. Other usage types will cause an error. If the usage type of *simple_expression* is NUMBER, then the physical segment address of symbol will be 0.

■ Example ■

```
CBIT_SYM1  CBIT   1000H.1

CBIT_SYM2  CBIT   2:2000H.4

      CSEG   #3 AT 3000H
LABEL:    DS    2
CBIT_SYM3  CBIT   LABEL.4
```

In this example, three absolute symbols with usage type CBIT are defined using the CBIT directive (CBIT_SYM1, CBIT_SYM2, CBIT_SYM3). CBIT_SYM1 will be a symbol representing bit 1 of offset address 1000H in physical segment 0. CBIT_SYM2 will be a symbol representing bit 4 of offset address 2000H in physical segment 2. CBIT_SYM3 will be a symbol representing bit 4 of offset address 3000H in physical segment 3.

■ Syntax ■

```
symbol DATA simple_expression
```

■ Description ■

The DATA directive defines a local symbol that represents an address in DATA address space. The *symbol* specifies the local symbol to be defined. The *simple_expression* represents an address in DATA address space. It is specified as a simple expression that does not include forward references.

If *simple_expression* is a constant expression, then *symbol* will be an absolute symbol. If *simple_expression* is a simple relocatable expression, then *symbol* will be a simple relocatable symbol. The *symbol* will be given the address value of *simple_expression* and usage type DATA.

■ Additional Information ■

The usage type of *simple_expression* must be DATA, NONE, or NUMBER. Other usage types will cause an error. If the usage type of *simple_expression* is NUMBER, then the physical segment address of symbol will be 0.

■ Example ■

```
DATA_SYM1  DATA   1000H

DATA_SYM2  DATA   2:2000H

      DSEG   #3 AT 3000H
LABEL:    DS    200H
DATA_SYM3  DATA   LABEL+100H
```

In this example, three absolute symbols with usage type DATA are defined using the DATA directive (DATA_SYM1, DATA_SYM2, DATA_SYM3). DATA_SYM1 will be a symbol representing offset address 1000H in physical segment 0. DATA_SYM2 will be a symbol representing offset address 2000H in physical segment 2. DATA_SYM3 will be a symbol representing offset address 3100H in physical segment 3.

■ Syntax ■

```
    symbol BIT simple_expression
```

■ Description ■

The BIT directive defines a local symbol that represents an address in BIT address space. The *symbol* specifies the local symbol to be defined. The *simple_expression* represents an address in BIT address space. It is specified as a simple expression that does not include forward references.

If *simple_expression* is a constant expression, then *symbol* will be an absolute symbol. If *simple_expression* is a simple relocatable expression, then *symbol* will be a simple relocatable symbol. The *symbol* will be given the address value of *simple_expression* and usage type BIT.

■ Additional Information ■

The usage type of *simple_expression* must be BIT, NONE, or NUMBER. Other usage types will cause an error. If the usage type of *simple_expression* is NUMBER, then the physical segment address of symbol will be 0.

■ Example ■

```
BIT_SYM1   BIT   1000H.1

BIT_SYM2   BIT   2:2000H.3

       DSEG  #3 AT 3000H
LABEL:    DS   2
BIT_SYM3   BIT   LABEL.4
```

In this example, three absolute symbols with usage type BIT are defined using the BIT directive (BIT_SYM1, BIT_SYM2, BIT_SYM3). BIT_SYM1 will be a symbol representing bit 1 of offset address 1000H in physical segment 0. BIT_SYM2 will be a symbol representing bit 3 of offset address 2000H in physical segment 2. BIT_SYM3 will be a symbol representing bit 4 of offset address 3000H in physical segment 3.

■ Syntax ■

```
    symbol EDATA simple_expression
```

■ Description ■

The EDATA directive defines a local symbol that represents an address in EDATA address space. The *symbol* specifies the local symbol to be defined. The *simple_expression* represents an address in EDATA address space. It is specified as a simple expression that does not include forward references.

If *simple_expression* is a constant expression, then *symbol* will be an absolute symbol. If *simple_expression* is a simple relocatable expression, then *symbol* will be a simple relocatable symbol. The *symbol* will be given the address value of *simple_expression* and usage type EDATA.

■ Additional Information ■

RAS66K always assumes that the EEPROM area is allocated to physical segment 0. Therefore symbols and expressions of type EDATA cannot have physical segment addresses other than 0.

The usage type of *simple_expression* must be EDATA, NONE, or NUMBER. Other usage types will cause an error. If the usage type of *simple_expression* is NUMBER and the EEPROM area is included in the COMMON area, then the physical segment attribute will be COMMON. If the usage type of *simple_expression* is NUMBER and the EEPROM area is not included in the COMMON area, then the physical segment address of symbol will be 0.

■ Example ■

```
EDATA_SYM1  EDATA   5000H

EDATA_SYM2  EDATA   0:5010H

        ESEG    AT 5000H
LABEL:      DS    40H
EDATA_SYM3  EDATA   LABEL+20
```

In this example, three absolute symbols with usage type EDATA are defined using the EDATA directive (EDATA_SYM1, EDATA_SYM2, EDATA_SYM3). EDATA_SYM1 will be a symbol representing offset address 5000H in physical segment 0. EDATA_SYM2 will be a symbol representing offset address 5010H in physical segment 0. EDATA_SYM3 will be a symbol representing offset address 5020H in physical segment 0.

■ Syntax ■

```
    symbol EBIT simple_expression
```

■ Description ■

The EBIT directive defines a local symbol that represents an address in EBIT address space. The *symbol* specifies the local symbol to be defined. The *simple_expression* represents an address in EBIT address space. It is specified as a simple expression that does not include forward references.

If *simple_expression* is a constant expression, then symbol will be an absolute symbol. If *simple_expression* is a simple relocatable expression, then *symbol* will be a simple relocatable symbol. The *symbol* will be given the address value of *simple_expression* and usage type EBIT.

■ Additional Information ■

RAS66K always assumes that the EEPROM area is allocated to physical segment 0. Therefore symbols and expressions of type EBIT cannot have physical segment addresses other than 0.

The usage type of *simple_expression* must be EBIT, NONE, or NUMBER. Other usage types will cause an error. If the usage type of *simple_expression* is NUMBER and the EEPROM area is included in the COMMON area, then the physical segment attribute will be COMMON. If the usage type of *simple_expression* is NUMBER and the EEPROM area is not included in the COMMON area, then the physical segment address of symbol will be 0.

■ Example ■

```
EBIT_SYM1   EBIT   5000H.0


EBIT_SYM2   EBIT   0:5010H.4


       ESEG    AT 5000H
LABEL:     DS     40H
EBIT_SYM3   EBIT   (LABEL+20).5
```

In this example, three absolute symbols with usage type EBIT are defined using the EBIT directive (EBIT_SYM1, EBIT_SYM2, EBIT_SYM3). EBIT_SYM1 will be a symbol representing bit 0 of offset address 5000H in physical segment 0. EBIT_SYM2 will be a symbol representing bit 4 of offset address 5010H in physical segment 0. EBIT_SYM3 will be a symbol representing bit 5 of offset address 5020H in physical segment 0.

4.12.7 Absolute Segment Definition (CSEG, DSEG, BSEG, ESEG, EBSEG)

■ Syntax ■

```
CSEG [ #physical_segment_address ] [ AT start_address ]
CSEG [ AT start_address ] [ #physical_segment_address ]

DSEG [ {#physical_segment_address |COMMON} ] [ AT start_address ]
DSEG [ AT start_address ] [ {#physical_segment_address |COMMON ]

BSEG [ {#physical_segment_address |COMMON} ] [ AT start_address ]
BSEG [ AT start_address ] [ {#physical_segment_address |COMMON} ]

ESEG [ AT start_address ]

EBSEG [ AT start_address ]
```

■ Description ■

These directives declare the start of an absolute segment definition.

Programs created in RAS66K assembly language are defined as a collection of multiple (at least 1) logical segments. RAS66K needs to be informed when the logical segment type is switched within a program.

The directives described in this section are specified to place code in their corresponding absolute segments. To start a relocatable segment, specify the RSEG directive. Use of the RSEG directive is described in Section 4.12.8.3, “Relocatable Segment Definition (RSEG).”

The relationship between the directives and the segment types of the defined logical segments is shown below.

Directive	Corresponding Segment
CSEG	Absolute CODE segment
DSEG	Absolute DATA segment
BSEG	Absolute BIT segment
ESEG	Absolute EDATA segment
EBSEG	Absolute EBIT segment

Each directive can have parameters that indicate the start address and physical segment address of the defined logical segment. The meanings of these parameters are explained next.

(1) *#physical_segment_address*

The *#physical_segment_address* specifies the physical segment address of the logical segment to be defined. The *physical_segment_address* represents a physical segment address. It must be a constant expression that does not include forward references. The *#physical_segment_address* can be specified with the CSEG, DSEG, and BSEG directives.

■ Example ■

```
 CSEG  #3
.
.
.
DSEG  #4
.
.
.
BSEG  #2
.
.
.
```

This example defines an absolute CODE segment in physical segment 3, an absolute DATA segment in physical segment 4, and an absolute BIT segment in physical segment 2.

(2) COMMON

COMMON declares that the logical segment is defined to reside in the COMMON area. It can be specified with the DSEG and BSEG directives.

As explained in Section 4.5.3, “COMMON Area,” RAS66K handles the COMMON area and each physical segment of data memory space as separate spaces. By specifying COMMON, the logical segment will reside in the COMMON area. RAS66K manages the addresses of logical segments with the COMMON specification and addresses of each physical segment independently.

■ Example ■

```
DSEG  COMMON
.
.
.
BSEG  COMMON
.
.
.
```

This example defines an absolute DATA segment and absolute BIT segment that reside in the COMMON area.

(3) AT *start_address*

AT *start_address* specifies the start address of the logical segment to be defined. The *start_address* must be a constant expression that does not include forward references. By specifying AT *start_address,* the value of the location counter will be changed to that of the specified address.

The meaning of AT *start_address* differs when *start_address* is a numeric expression and when it is an address expression. When *start_address* is a numeric expression, it only specifies an offset address. On the other hand, when *start_address* is a numeric expression, it specifies both a physical segment address and an offset address. In such cases, *#physical_segment_address* and COMMON cannot be specified at the same time.

■ Example ■

```
CSEG  #4 AT 2000H
CSEG     AT 4:2000H
CSEG  #5 AT 2:3000H  ;ERROR
```

In this example, the first CSEG directive specifies #4 as the physical segment address and 2000H as the offset address. The second CSEG directive expresses the identical specification of the first, but expresses the address as 4:2000H. The third CSEG directive specifies an address constant 2:3000H with AT even though it also specifies a physical segment address with #5. In this case, RAS66K cannot determine if the physical segment should be 5 or 2, so an error will occur.

Parameters can be divided into two types.

- Parameters for specifying physical segments (*#physical_segment_address* and COMMON)
- Parameters for specifying start addresses (AT *start_address*)

Either of these parameters can be specified first. Also, these parameters may be omitted if not necessary. If the parameters are omitted, then the segment will inherit its settings from the previous logical segment of the same segment type.

(1) Omitting parameter for specifying physical segment

If both *#physical_segment_address* and COMMON are omitted, then the segment will inherit its setting from the previous physical segment of the same segment type. This is true, however, only when *start_address* is a numeric expression.

■ Example ■

```
  CSEG    #2  AT 1000H
  .
  .
  .
  CSEG    AT 3000H   ;#2
;
  DSEG     COMMON AT 200H
  .
  .
  .
  DSEG    AT 300H    ;COMMON
```

The first CSEG directive specifies physical segment 2. The next CSEG directive has no physical segment specification, but specifies only a start address of 3000H. In this case, the physical segment address will be set to 2.

Similarly, the first DSEG directive specifies the physical segment COMMON. The next DSEG directive has no physical segment specification, but specifies only a start address of 300H. In this case, the physical segment will be set to COMMON.

(2) Omitting parameter for specifying start address

If AT *start_address* is omitted, then the segment will inherit its offset address from the previous logical segment of the same segment type.

■ Example ■

```
    DSEG  #2  AT 2000H
    DS    10H   ;2000H+10H
;
    .
    .
    .
    DSEG  #2    ;start address 2010H
```

The first DSEG directive specifies a start address of 2000H in the physical segment #2. Since the DS directive reserves 10 bytes, the location counter of the #2 is changed to 2010H. The next DSEG directive specifies a physical segment address but not a start address. In this case, the start address is set to 2010H, which is the value of the previous location counter.

(3) Specifying no parameter

If no parameter is specified, then the segment will inherit its settings and offset address from the previous physical segment of the same segment type.

■ Example ■

```
    DSEG  #2  AT 2000H
    DS    10H   ;2000H+10H
;
    DSEG  #3  AT 3000H
    DS    20H   ;3000H+20H
;
    .
    .
    .
    DSEG   ;start address 3:3020H
```

The last DSEG directive does not specify a physical segment or a start address. In this case, it will inherit its settings from the previous DSEG directive. The physical segment address will be set to 3, and the offset address will be set to 3020H.

4.12.8 Using Relocatable Segments

Relocatable segments are logical segments for which absolute addresses are not determined during assembly, but during linking instead. The basic concepts of relocatable segments are explained in Section 4.5.2.2, “Relocatable Segments,” so this section explains only the use of directives for them.

Relocatable segments are distinguished by names called segment symbols. The procedure for using relocatable segments is as follows.

(1) Define a segment symbol using the SEGMENT directive.
(2) Define the relocatable segment by specifying this segment symbol in the operand of an RSEG directive.

4.12.8.1 Segment Symbol Definition (SEGMENT)

■ **Syntax** ■

```
segment_symbol SEGMENT segment_type [ boundary_attr ] [ relocation_attr ]
```

■ **Description** ■

The SEGMENT directive defines a segment symbol. Up to 65,535 segment symbols can be defined within a single source file.

The *segment_symbol* specifies the segment symbol to be defined. It is used to distinguish between relocatable segments by specifying it in the operand of RSEG directives. It can also be used in the operand of instructions. In such cases, the *segment_symbol* will represent the base address of a relocatable segment, with a value of 0 during assembly.

There are three parameters in a SEGMENT directive. The *segment_type* must be specified, but if *boundary_attr* and *relocation_attr* are not needed, then they may be omitted. The meanings of these parameters are explained below.

Segment Type (*segment_type*)

The *segment_type* specification represents the type of address space to which the relocatable segment will be allocated. One of the following segment types can be specified.

segment_type	Description
CODE	Allocate to CODE address space.
DATA	Allocate to DATA address space.
BIT	Allocate to BIT address space.
EDATA	Allocate to EDATA address space.
EBIT	Allocate to EBIT address space.

Boundary Value Attribute (*boundary_attr*)

The *boundary_attr* specifies the boundary value of the first address when the relocatable segment is allocated. This is called the boundary value attribute. A symbol or integer constant expression representing the boundary value is specified for *boundary_attr*. The meanings of each type of *boundary_attr* and the segment types for which they can be specified are shown below.

boundary_attr	Meaning	Segment Types
UNIT	Byte boundary for CODE, DATA, EDATA. Bit boundary for BIT, EBIT.No restriction	No restriction
WORD	2-byte boundary.	CODE, DATA, EDATA
OCT	8-byte boundary.	CODE, DATA, EDATA
PAGE	1-page (256-byte) boundary.	CODE, DATA, EDATA
integer constant	Specified value will be boundary. Value can be one of following: 1 2 4 8 16 32 64 128 256 512 1024 2048	No restriction

If *boundary_attr* is omitted, then UNIT will be assigned automatically.

Special Area Attribute (*relocation_attr*)

The *relocation_attr* specifies the area where the relocatable segment is to be allocated. This is called the special area attribute of a logical segment. The meanings of each type of *relocation_attr* and the segment types for which they can be specified are shown below.

relocation_attr	**Meaning**	**Segment Types**
#phy_seg_addr	Allocate to physical segment address phy_seg_addr. The #phy_seg_addr is a constant expression.	CODE, DATA, BIT
COMMON	Allocate to the COMMON area.	DATA, BIT
WINDOWALL	Allocate to the ROM window area.	CODE
WINDOW	Allocate to the ROM window area, except for the address ranges of the EEPROM area, the dual port RAM area, and the internal RAM area.	CODE
ACAL	Place at least the start of the logical segment in the ACAL area.	CODE
INACAL	Place the entire logical segment in the ACAL area.	CODE
INPAGE	Allocate within a page. RL66K will determine the page number.	No restriction
INPAGE(*page*)	Allocate within the specified page. The page must be a constant expression.	No restriction
SBA	Allocate within an SBA area. RL66K will determine the page number.	No restriction
SBA(*page*)	Allocate within the SBA area of the specified page. The page must be a constant expression.	No restriction
ZERO	Allocate within the zero page area. This has the same meaning as INPAGE(0).	DATA, BIT
FIX	Allocate within the fixed page area. This has the same meaning as INPAGE(2).	DATA, BIT
SBAFIX	Allocate to the SBA area in the fixed page area. This has the same meaning as SBA(2).	DATA, BIT
DUAL	Allocate within the dual port RAM area.	DATA, BIT
LREG	Allocate within the local register area.	DATA
DYNAMIC	Reserve the maximum possible size.	DATA

The following special area attributes can be specified only with nX-8/500.

WINDOW WINDOWALL ACAL INACAL SBA FIX SBAFIX DUAL LREG

The ZERO attribute can be specified with nX-8/100~400.

Both the WINDOW and WINDOWALL attributes indicate allocation to the ROM window area. However, when the WINDOW attribute is specified, RL66K will perform allocation avoiding the address ranges of the EEPROM area, the dual port RAM area, and the internal RAM area.

There are two types of special area attributes for the ACAL area: the ACAL attribute and the INACAL attribute. The ACAL attribute guarantees that the start address of the logical segment will be placed in the ACAL area. The INACAL attribute guarantees that the entire logical segment will be placed within the ACAL area. If the logical segment is a single subroutine called with the ACAL instruction, then specify the ACAL attribute. If the logical segment includes multiple subroutines called with the ACAL instruction, then the INACAL attribute is safer.

The INPAGE attribute guarantees that the logical segment will be placed within a single page. Specify the INPAGE attribute to use current page addressing for a particular data area.

The FIX, SBA, and SBAFIX attributes are specified to use efficient addressing with a data area defined as a relocatable segment. These specifications guarantee allocation to the fixed page area or SBA area. Page addressing (fix address, sbaoff address, sbafix address) can be used for addresses in logical segments that have these specifications. In addition, RAS66K addressing optimization (Section 4.11.7, “Addressing Optimization”) is provided for addresses of relocatable segments that have these special area attributes.

The DYNAMIC attribute is provided for implementation of automatic memory allocation for standard high-level languages. After RL66K allocates all other logical segments to address spaces, it will allocate a logical segment with the DYNAMIC attribute to the remaining area with the largest size.

Special area attributes can be combined as needed. There is no restriction on the order of combination, but special area attributes that indicate completely different areas cannot be combined.

Depending on the type of special area attribute, the size of the relocatable segment might have restrictions. For example, relocatable segments with the INPAGE attribute specified must fit within the range of a page, so they cannot have a size greater than 100H bytes.

If no special area attribute is necessary, none needs to be specified. If special area attributes are omitted, then RL66K will determine the physical segment and address where the relocatable segment will be allocated.

Examples of SEGMENT directive use are shown next.

■ Example 1 ■

This is an example of the simplest definitions.

```
SEG_C     SEGMENT CODE

SEG_D     SEGMENT DATA

SEG_B     SEGMENT BIT
```

SEG_C will be allocated to CODE address space. SEG_D will be allocated to DATA address space. SEG_B will be allocated to BIT address space.

■ Example 2 ■

This is an example of boundary value attribute specifications.

```
DATA_TBL1 SEGMENT DATA WORD

DATA_TBL2 SEGMENT DATA PAGE

ROM_TBL   SEGMENT CODE 400H
```

DATA_TBL1 will be allocated on a word boundary. DATA_TBL2 will be allocated on a page boundary. ROM_TBL will be allocated on a 400H-byte boundary.

■ Example 3 ■

This is an example of special area attribute specifications.

```
COM_SEG   SEGMENT DATA COMMON

PHY2SEG   SEGMENT BIT  #2

WIN_DAT   SEGMENT CODE WINDOW

ACAL_GR   SEGMENT CODE INACAL

FIX_SEG   SEGMENT DATA FIX
```

COM_SEG will be allocated to the COMMON area. PHY2SEG will be allocated to physical segment 2. WIN_DAT will be allocated to the ROM window area. ACAL_GR will be allocated to the ACAL area. FIX_SEG will be allocated to the fixed page area.

■ Example 4 ■

This is an example of specifications combining boundary value attributes and multiple special area attributes.

```
PG10SEG   SEGMENT DATA   PAGE INPAGE(10H)  #0

COM_FIX   SEGMENT DATA   WORD COMMON FIX

ROM_TBL   SEGMENT CODE   WINDOWALL #5  INPAGE
```

PG10SEG will be allocated in physical segment 0 on a 10H-page boundary. COM_FIX will be allocated to the fixed page area in the COMMON area on a word boundary. ROM_TBL will be allocated within a page in the ROM window area of physical segment 5.

4.12.8.2 Stack Segment Definition (STACKSEG)

■ Syntax ■

```
    STACKSEG   stack_size
```

■ Description ■

The STACKSEG directive defines the stack segment.

The *stack_size* specifies the size of the stack segment. It must be a constant expression that does not include forward references.

When the STACKSEG directive is specified, RAS66K automatically generates a stack segment with the name $STACK. $STACK is a relocatable segment, but it cannot be specified as an operand of the RSEG directive.

■ Additional Information ■

The initial value of SSP (the address that is 1 less than the end address of the stack segment) can be referred with _$$SSP. To refer this symbol, it must be declared an external reference with the EXTRN directive.

For an overview of the stack segment, refer to Section 4.5.4, “Stack Segment.”

■ Example ■

```
STACKSEG  200H               ;Defines stack segment $STACK
EXTRN      DATA:_$$SSP
```

In this example, 200H bytes are reserved as a stack area and the EXTRN directive defines _$$SSP to refer the end address of the stack segment.

4.12.8.3 Relocatable Segment Definition (RSEG)

■ **Syntax** ■

```
RSEG segment_symbol
```

■ **Description** ■

The RSEG directive defines a relocatable segment.

The *segment_symbol* specifies segment symbol that represents the relocatable segment to be defined. The *segment_symbol* must have been defined using a SEGMENT directive before the position of the RSEG directive.

The start address of the relocatable segment during assembly will always be set to 0. Thus, when the code of a relocatable segment starts with an RSEG directive, the location counter will be set to 0.

A single relocatable segment can also be defined divided in multiple blocks. In such cases, use of the RSEG directive declares that a relocatable segment is to be defined. If it is defining again a relocatable segment that was previously defined one or more times, then it will inherit the last address of the immediately previous definition as its location counter value.

■ **Example** ■

```
WORKDAT SEGMENT DATA WORD COMMON  ;Define segment symbol
     RSEG    WORKDAT              ;Define relocatable DATA segment
     DS      10H

CHARBUF SEGMENT DATA WORD         ;Define segment symbol
     RSEG    CHARBUF              ;Define relocatable DATA segment
     DS      2H

SUB1    SEGMENT CODE              ;Define segment symbol
SUB2    SEGMENT CODE #0           ;Define segment symbol
SUB3    SEGMENT CODE #1           ;Define segment symbol

     RSEG    SUB1                 ;Define relocatable CODE segment
     MOVB    WORKDAT,#1
     MOVB    DSR,#SEG CHARBUF
     MOVB    CHARBUF,#1
     .
     .
     .
     RSEG    SUB2                 ;Define relocatable CODE segment
     MOVB    WORKDAT,#2
     MOVB    DSR,#SEG CHARBUF

     RSEG    SUB3                 ;Define relocatable CODE segment
     MOVB    WORKDAT,#3
```

```
    MOVB    DSR,#SEG CHARBUF
    MOVB    CHARBUF,#3
    .
    .
    .
    RSEG    SUB2                        ;Inherit previously defined
                                        ;relocatable segment SUB2
    MOVB    CHARBUF,#2
    .
    .
    .
```

This example defines two relocatable DATA segments and three relocatable CODE segments.

The relocatable DATA segment WORKDAT will be allocated within the COMMON area of DATA address space on a 2-byte boundary. The relocatable DATA segment CHARBUF will be allocate in DATA address space on a 2-byte boundary. The relocatable CODE segment SUB1 will be allocated somewhere in CODE address space. The relocatable CODE segment SUB2 will be allocated somewhere in physical segment 0 of CODE address space. The relocatable segment SUB3 will be allocated somewhere in physical segment 1 of CODE address space. The relocatable CODE segment SUB2 is defined twice using the RSEG directive, so the second definition of SUB2 inherits the end address of the first definition.

4.12.9 Segment Group Definition (GROUP)

When the target microcontroller has multiple physical segments, segment register contents must be written each time data of a different physical segment is referred. By using the GROUP directive when coding a program that handles multiple relocatable segments simultaneously, the relocatable segments all can be allocated to the same physical segment, making segment register management much easier.

■ Syntax ■

```
GROUP segment_symbol [ segment_symbol... ] [ #physical_segment_address ]
```

■ Description ■

The GROUP directive tells RL66K to allocate multiple relocatable segments in a single physical segment. A group of relocatable segments to be located in a single physical segment is called a segment group.

Each *segment_symbol* specifies a segment symbol that represents a relocatable segment to reside in the segment group. Each must have been defined with a SEGMENT directive before the GROUP directive is specified. At least one *segment_symbol* must be specified. When multiple *segment_symbols* are specified, they are delimited with spaces.

The *#physical_segment_address* represents the physical segment address where the relocatable segments will be allocated. It must be a constant expression that does not include forward references.

The segments of a segment group are located in the same memory space. This means that the segment types of the segment symbols must meet one of the following two conditions.

- All segment symbols must have segment type CODE.
- All segment symbols must be a combination of segment types DATA and BIT.

The EEPROM area exists in physical segment 0, so EDATA segments and EBIT segments will always be allocated to physical segment 0. Therefore, EDATA and EBIT segments cannot be registered in segment groups.

(1) If *#physical_segment_address* is not specified

If a segment is defined for which a physical segment address is specified, then the segment group will be allocated to that physical segment. If multiple segment symbols are defined for which physical segment addresses are specified, then they must all be the same. If any of the segment symbols has a different physical segment address, then an error will occur with RAS66K.

If no segment is defined for which physical segment address is specified , then RL66K will determine the physical segment to which the segment group will be allocated.

(2) If *#physical_segment_address* is specified

The segment group will be located in the physical segment specified in the GROUP directive operand.

If a segment symbol is defined for which a physical segment address is specified, then the physical segment address of the GROUP directive operand must be the same as that of the segment symbol.

■ Additional Information ■

The same relocatable segment cannot be registered to multiple segment groups.

■ Example ■

```
DATSEG1 SEGMENT DATA
    RSEG    DATSEG1         ;Define relocatable DATA segment
    DS      10H

BITSEG1 SEGMENT BIT
    RSEG    BITSEG1         ;Define relocatable BIT segment
    DBIT    8

DATSEG2 SEGMENT DATA
    RSEG    DATSEG2         ;Define relocatable DATA segment
BUF0:
    DS      10H

BITSEG2 SEGMENT BIT
    RSEG    BITSEG2         ;Define relocatable BIT segment
FLAG0:
    DBIT    8

GROUP   DATSEG1 BITSEG1 #1      ;Define segment group
GROUP   DATSEG2 BITSEG2         ;Define segment group

SUB1    SEGMENT CODE

    RSEG    SUB1            ;Define relocatable CODE segment
    MOVB    DSR,#SEG DATSEG2
    MOVB    R0,BUF0
    SB      FLAG0           ;No need to write to DSR
```

This example defines four relocatable segments allocated to DATA address space and BIT address space.

The GROUP directive is used to define DATSEG1 and BITSEG1 to a group, and DATSEG2 and BITSEG2 to another group. For DATSEG1 and BITSEG1, the GROUP directive operand is specified as #1, so they will be allocated to physical segment 1. DATSEG2 and BITSEG2 will be allocated to the same physical segment, but RL66K will determine the physical segment address.

4.12.10 Location Counter Setting (ORG)

■ Syntax ■

```
ORG address
```

■ Description ■

The ORG directive sets the value of the location counter of the logical segment that contains it to the value of address. The function of the ORG directive will differ depending on whether its logical segment is an absolute segment or a relocatable segment. Both cases are discussed below.

(1) ORG Directives In Absolute Segments

The address sets the value of the location counter with a constant expression that does not include forward references. The constant expression must have a value greater than or equal to the starting address of the logical segment in which the directive is placed. It also must have a value within the target address space. If the constant expression is an address expression, then its physical segment address must be the same as the physical segment address of the absolute segment.

■ Example ■

```
CSEG #1 AT 1000H
.
.
.
ORG 1030H
.
.
.
ORG 1100H
.
.
.
ORG 200H              ;error
```

In this example, ORG directives are used in the absolute CODE segment in physical segment address 1. Its starting address is 1000H. Accordingly, ORG directive operands must have values of 1000H or higher. The value of the last ORG directive operand is 200H, so an error will occur.

(2) ORG Directives In Relocatable Segments

The address sets the value of the location counter with a simple expression that does not include forward references. If the simple expression includes simple relocatable symbols, then the relocatable segment in which they reside must be the same as the current relocatable segment.

If the operand is a constant expression, then its value represents the offset from the starting address of the relocatable segment in which the directive resides.

■ Example ■

```
DATSEG3 SEGMENT DATA
    RSEG    DATSEG3
LABEL1: DS      10H
    ORG     LABEL1+30H
LABEL2: DS      10H
    ORG     100H
LABEL3:
```

In this example, two ORG directives are used in the relocatable segment. The operand of the first ORG directive uses a label of its relocatable segment. The operand of the second ORG directive is a constant expression. Its 100H represents the offset from the starting address of its relocatable segment.

4.12.11 Memory Allocation (DS, DBIT)

The DS and DBIT directives allocate areas of specified sizes to address space. These directives are normally used to allocate areas in DATA address space. These areas are used to store process data of the program.

■ Syntax ■

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

■ Description ■

The DS directive allocates an area of the number of bytes specified by size to the logical segment in which the directive resides. The DBIT directive allocates an area of the number of bits specified by size to the logical segment in which the directive resides. The specified size is added to the location counter of the logical segment in which the directive resides.

The size specifies the number of bytes or bits to be allocated to address space with a constant expression. The constant expression must not include forward references.

The DS directive is used to reserve areas in CODE, DATA, or EDATA segments. The DBIT directive is used to reserve areas in BIT or EBIT segments.

■ Example 1 ■

This example uses the DS directive.

```
BUFFER  SEGMENT DATA WORD COMMON
        RSEG    BUFFER
BUFL:   DS      10H                 ;Memory allocation

FUNC2   SEGMENT CODE
        RSEG    FUNC2
        MOV     BUFL,ER0
```

In this example, a 10H-byte area will be allocated in the relocatable DATA segment BUFFER.

■ Example 2 ■

This example uses the DBIT directive.

```
FLAG_FIELD  SEGMENT BIT
       RSEG    FLAG_FIELD
FLAG1:
       DBIT    8

FUNC3       SEGMENT CODE
       RSEG    FUNC3
       SB      FLAG1
```

In this example, an 8-bit area will be allocated in the relocatable BIT segment FLAG_FIELD.

4.12.12 Program Memory Initialization (DB, DW)

The DB and DW directives initialize program memory or EEPROM areas to specified values. They are used to define data in program memory.

■ Syntax ■

```
[ label: ] DB{expression | string_constant}[,{expression | string_constant}] ...
```

■ Description ■

The DB directive initializes program memory or the EEPROM area in bytes. The operands can be specified as expressions (*expression*) or string constants (*string_constant*).

Each expression can be specified as a general expression and may include forward references. Each expresses one byte of data, so the values must be within the following range.

```
-80H  —  -1H  (0FFFFFF80H—0FFFFFFFFH)
 0H  — 0FFH
```

The data of a string constant (*string_constant*) will be initialized in the order of the characters.

■ Additional Information ■

The DB directive can be used in CODE and EDATA segments.

■ Syntax ■

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

■ Description ■

The DW directive initializes program memory or the EEPROM area in words. The operands can be specified as expressions (*expression*).

Each expression can be specified as a general expression and may include forward references. Each expresses one word (two bytes) of data, so the values must be within the following range.

```
-8000H  –    -1H  (0FFFF8000H–0FFFFFFFFH)
  0H  – 0FFFFH
```

■ Additional Information ■

The DW directive can be used in CODE and EDATA segments.

■ Example ■

This example uses DB and DW directives to initialize program memory and the EEPROM area.

```
TABLE          SEGMENT CODE
         RSEG    TABLE
         DW      -1, -4, -9, -16
         DW     1, 4, 9, 16

CHAR_TABLE      SEGMENT CODE
         RSEG    CHAR_TABLE
         DB     'A','B','C','D','E','F'

STRING_TABLE   SEGMENT EDATA
         RSEG    STRING_TABLE
         DB     " define  byte "
         DB     " defie " , " byte "
```

4.12.13 Creating Programs From Multiple Source Files

A single program can be developed dividing it into multiple source files. To refer a symbol shared in all source files, the following declarations are necessary.

• In the source file that defines a symbol, a declaration is needed to refer the symbol in other source files (public declaration).

• In a source file that refers a symbol, a declaration is needed if the symbol was defined in another source file (external declaration).

A publicly declared symbol is called a public symbol. An externally declared symbol is called an external symbol. Use the PUBLIC directive to declare public symbols, and the EXTRN directive to declare external symbols. If an external symbol is declared an a source file, then a public symbol with the same name must exist in another source file.

There are also communal symbols, which are symbols with features of both public symbols and external symbols. Use the COMM directive to define communal symbols. When communal symbols with the same name are defined in multiple source files, they will all represent a common memory area.

As described above, public symbols and communal symbols can be referred from multiple source files. Based on this meaning, the two types of symbols are together called global symbols.

Segment symbols defined with the SEGMENT directive cannot be declared external symbols with the EXTRN directive. To use a segment symbol in multiple source files, the symbol must be defined separately in each source file.

Public symbols, external symbols, communal symbols, and segment symbols are described below.

4.12.13.1 Public Symbol Declaration (PUBLIC)

■ Syntax ■

```
   PUBLIC symbol [ symbol ] …
```

■ Description ■

The PUBLIC directive declares local symbols as public symbols. By declaring a local symbol as a public symbol, that symbol can be used in other source files.

The symbol specifies a local symbol. It does not matter whether the local symbol's definition or its public declaration is first.

Multiple symbols can be specified as operands of the PUBLIC directive.

■ Additional Information ■

To refer a public symbol from another source file, the source file that contains the reference must declare an external symbol with the same name using the EXTRN directive.

Public symbols with the same name cannot be declared in multiple source files.

If a user symbol that has been redefined with the SET directive is declared public, then the public symbol will have the value of the last definition.

■ Example ■

```
PUBLIC  GLOBAL_NUMBER  GLOBAL_LABEL

GLOBAL_NUMBER   EQU     1

    DATSEG   SEGMENT DATA 2
         RSEG    DATSEG
GLOBAL_LABEL:   DS      10H
```

In this example, the absolute symbol GLOBAL_NUMBER1 and the simple relocatable symbol GLOBAL_LABEL1 are declared public.

4.12.13.2 External Symbol Declaration (EXTRN)

■ **Syntax** ■

```
EXTRN usage_type [ attribute ] :symbol [ symbol ] …
          [ usage_type [ attribute ] :symbol [ symbol ] … ] …
```

■ **Description** ■

The EXTRN directive declares external symbols. The *usage_type* specifies the usage type of the external symbol. A *usage_type* specification is valid until a different one is specified in an operand. One of the following usage types can be specified.

usage_type	**Description**
CODE	Indicates an address in CODE address space.
CBIT	Indicates an bit address in CODE address space.
DATA	Indicates an address in DATA address space.
BIT	Indicates an address in BIT address space.
EDATA	Indicates an address in EDATE address space.
EBIT	Indicates an address in EBIT address space.
NUMBER	Indicates a number.

The symbol specifies an external symbol. An external symbol will refer a public symbol or communal symbol that has been declared in another source file.

The attribute declares if the symbol's address is in the fixed page area or SBA area. Specify an attribute for a declared symbol when performing RAM addressing optimization. The attribute types and their meanings are as follows.

usage_type	**Meaning**
FIX	Address in fixed page area.
SBA	Address in SBA area.
SBAFIX	Address in SBA area of fixed page area.

The *usage_type* can be specified as FIX or SBAFIX for DATA or BIT symbols. It can be specified as SBA for symbols other than NUMBER symbols.

■ Additional Information ■

The usage type of external symbols and their corresponding public symbol must match.

The total number of external symbols within one source file must be 65,535 or less. If the same external symbol is defined two or more times within one source file, then an error will occur.

By using the /X option, RAS66K will automatically create a file containing the external declarations corresponding to publicly defined symbols. Refer to Section 4.14, "EXTRN Declaration Files."

■ Example ■

Assume the following public symbol definitions are in some source file.

```
PUBLIC    BUFSIZE  DATA_TBL   SUB_FUNC

BUFSIZE   EQU      100H

      DSEG     AT  200H
DATA_TBL: DS       10H

      CSEG     AT  1000H
SUB_FUNC:
```

A source file that refers these symbols will perform external declarations.

```
EXTRN   NUMBER:BUFSIZE
EXTRN   DATA FIX:DATA_TBL
EXTRN   CODE:SUB_FUNC
```

4.12.13.3 Communal Symbol Declaration (COMM)

Communal symbols define common data areas of multiple source files.

A communal symbol defined in multiple source files will represent the first address of the common data area. RL66K will determine the address of the data area defined by the communal symbol.

This means that communal symbols are similar to relocatable segments. However, the areas reserved for communal symbols cannot have labels defined or data initialized. Also, relocatable segments defined in multiple source files represent independent areas in each of the source files (refer to Section 4.12.13.5, "Using Partial Segments"), while communal symbols defined in multiple source files represent a common area in each source file.

Use the COMM directive to declare communal symbols. The syntax of the COMM directive is as follows.

■ Syntax ■

```
communal_symbol COMM segment_type size [ relocation_attr ]
```

■ Description ■

The syntax of the COMM directive is very similar to that of the SEGMENT directive. However, immediately after the *segment_type* is a size specification, representing the size of the area to reserve. Also, there is no boundary value attribute specification. The meanings and specifications of the segment type (*segment_type*) and special area attribute (*relocation_attr*) are the same as for the SEGMENT directive. Refer to Section 4.12.8.1, "Segment Symbol Definition," for their specification method.

The size is an integer constant expression that represents the size of the area to reserve for the communal symbol. The units of this size will differ depending on the *segment_type*. It is bytes when *segment_type* is CODE, DATA, or EDATA, and bits when *segment_type* is BIT or EBIT.

Unlike the SEGMENT directive, the COMM directive does not have a boundary value attribute specification. If the segment type is CODE, DATA, or EDATA, then the area reserved for the communal symbol will be allocated on a 1-byte boundary if size is 1, or a 2- byte boundary if size is 2 or greater. If the segment type is BIT or EBIT, then the area will be allocated on a 1-bit boundary.

When communal symbols with the same name are defined in multiple source files, a common area will be reserved with each source file. Take the following example.

```
COMM_AREA COMM  DATA 2
```

When this source statement is specified in multiple source files, the symbol COM_AREA will represent an address of a 2-byte area in DATA address space that is common to each source file. The key point here is that even though COM_AREA is defined in multiple source files, the reserved area will only be 2-bytes.

■ Additional Information ■

The total number of communal symbols within one source file must be 65,535 or less. If the same communal symbol is defined two or more times within one source file, then an error will occur.

If the size of the communal symbol differs in each source file, then the area of the maximum size specified will be allocated.

Communal symbols declared in other source files may be externally referred using the EXTRN directive.

Symbols declared public in other source files can be defined using the COMM directive, but instead of reserving an area of the size specified in its operand, it will simply declare, “refer another source file’s symbol.” In other words, it will behave exactly the same as if an EXTRN directive were coded. Cases like these may exist in source files generated by the C compiler CC66K. However, this is not recommended practice for programming in assembly language.

■ Example ■

The example below shows the use of the COMM directive.

```
     TYPE(M66507)
GL_BUF1 COMM DATA 100H
GL_BITF COMM BIT  4
     .
     .
     .
     MOV      ER0,GL_BUF1
     SB      GL_BITF+2
```

This example declares communal symbols GL_BUF1 and GL_BITF. These communal symbols are specified in operands of microcontroller instructions. GL_BUF1 reserves a 100H-byte area in DATA address space. GL_BITF reserves a 4-bit area in BIT address space.

4.12.13.4 Using Public, External, And Communal Symbols

To refer the same symbol from multiple source files, declare a public symbol, external symbol, or communal symbol in each source file. The following conditions must be satisfied. Programs that do not satisfy these conditions will cause errors during linking.

- Symbols are declared with the same name.
- Symbols are declared with the same usage type.

Only one symbol with the same name can be declared in each source file. For example, if a public symbol has been declared, then an external symbol or communal symbol with the same name cannot be declared in the same source file.

Below are representative examples of the use of symbols across multiple source files.

(1) Referring Public Symbols With External Symbols

The following example shows the use of PUBLIC directives and EXTRN directive for symbols that are defined in one source file and that are used in a different source file.

■ Example ■

```
/*
  Source File 1
*/
     TYPE(M66507)


     PUBLIC BUF_SIZE
     PUBLIC DAT_BUFF

BUF_SIZE EQU   100H
;
     DSEG  AT 200H
DAT_BUFF:
     DS    10H
```

```
/*
  Source File 2
*/
     TYPE(M66507)

EXTRN    NUMBER:BUF_SIZE
EXTRN    DATA FIX:DAT_BUFF


     CSEG
     MOV     X1,#BUF_SIZE
     ADDB    R0,DAT_BUFF
```

In this example, BUF_SIZE and DAT_BUFF are defined in source file 1 but are also used in source file 2. To do this, both symbols are declared public in source file 1 with the PUBLIC directive, and both are declared external in source file 2 with the EXTRN directive.

(2) Using Communal Symbols In Multiple Source Files

The following example shows the use of COMM directives for communal symbols that are defined in multiple source file but that utilize a common data area.

■ Example ■

```
/*
  Source File 1
*/
    TYPE(M66507)


GL_BUF1 COMM DATA 2 FIX
GL_BUF2 COMM DATA 2 FIX
GL_BUF3 COMM DATA 4 FIX

    CSEG
    MOVB    R0,GL_BUF1
    ADDB    R1,GL_BUF2
    XORB    R2,GL_BUF3
```

```
/*
  Source File 2
*/
    TYPE(M66507)

GL_BUF1 COMM DATA 2 FIX
GL_BUF2 COMM DATA 2 FIX
GL_BUF3 COMM DATA 6 FIX        ;Size of GL_BUF3 differs in source file 1.

    CSEG
    MOV     ER0,GL_BUF1
    L       A,GL_BUF2
    OR      A,GL_BUF3
```

In this example, three communal symbols (GL_BUF1, GL_BUF2, GL_BUF3) are defined in both source files. Common data areas for these symbols will be reserved in both source files. GL_BUF1 and GL_BUF2 will each be allocated 2 bytes. The size specified for GL_BUF3 differs in source file 1 and source file 2, so it will be allocated the larger size of 6 bytes.

(3) Referring Communal Symbols With External Symbols

The following example shows the use of EXTRN directive for communal symbols that are defined in one source file and that are used in a different source file.

■ Example ■

```
/*
  Source File 1
*/
    TYPE(M66507)


GL_BUF1 COMM DATA 2 FIX
GL_BUF1 COMM DATA 2 FIX
GL_BUF1 COMM DATA 4 FIX

    CSEG
    MOVB    R0,GL_BUF1
    ADDB    R1,GL_BUF2
    XORB    R2,GL_BUF3
```

```
/*
  Source File 2
*/
    TYPE(M66507)

EXTRN   DATA FIX:GL_BUF1 GL_BUF2 GL_BUF3

    CSEG
    MOV     ER0,GL_BUF1
    L      A,GL_BUF2
    OR      A,GL_BUF3
```

In this example, three communal symbols (GL_BUF1, GL_BUF2, GL_BUF3) are defined in source file 1. To use these symbols in source file 2, they are declared external using the EXTRN directive.

4.12.13.5 Using Partial Segments

To use the same segment symbol from multiple source files, a segment symbol with the same name must be defined in each source file with the SEGMENT directive. Relocatable segments used in multiple source files this way are called partial segments.

Below are examples showing the use of partial segments.

■ Example1 ■

In the following example, partial segments are defined in source file 1 and source file 2.

```
/*
  Source File 1
*/


PDATSEG SEGMENT DATA
    RSEG    PDATSEG
LABEL:  DS     2
```

```
/*
  Source File 2
*/

PDATSEG SEGMENT DATA
    RSEG    PDATSEG
LABEL:  DS     2
```

When RL66K links these two source files are linked using RL66K, it will handle the logical segment PDATSEG in both as the same segment, allocating them contiguously in DATA address space. The positional relationship of the two segments will be the order of the object files specified in the RL66K command line.

The labels LABEL in each source file are local symbols, so they will have different address values.

■ Example2 ■

```
/*
  Source File 1
*/

PDATSEG     SEGMENT DATA
       RSEG    PDATSEG
PDAT1L:    DS     2
```

```
/*
  Source File 2
*/

PDATSEG     SEGMENT DATA
       RSEG    PDATSEG
PDAT2L:    DS     2
```

```
/*
  Source File 3
*/

PDATSEG     SEGMENT DATA   ;Only definition is performed in this file


SUBCODE     SEGMENT CODE
       RSEG    SUBCODE
       MOV     ER0,#0
       MOV     PDATSEG,ER0
       MOV     PDATSEG+2,ER1
```

In this example, the partial segments PDATSEG defined in source file 1 and source file 2 will be allocated contiguously in DATA address space. In order to refer the segment symbol PDATSEG in source file 3, PDATSEG must be defined with the SEGMENT directive. PDATSEG will represent the first address of the singled linked segment.

4.12.14 Assumptions And Checks Of Program State (USING)

The USING directive is used to manage the states of hardware registers and flags affecting program operation. RAS66K looks at register and flag states specified with the USING directive and checks whether or not the use of instructions and addressing is appropriate for those states. The basic syntax of the USING directive is as follows.

■ Syntax ■

```
USING register_name status
```

The role of the USING directive is to inform RAS66K that the state of *register_name* is *status*. The *register_name* can be one of the following.

register_name	**Meaning**
DSREG	Specifies value of physical segment address of data memory space. It is the state of DSR for nX-8/500 and bits 13-15 of LRB for nX-8/300.
TSREG	Specifies value of physical segment address of program memory space for table refer ences. It is the state of TSR for nX-8/500.
PAGE 12	Specifies page number of current page. It is the state of LRBH for nX-8/500 and bits 5- of LRB for nX-8/100~400.
DATA	Specifies state of data descriptor (DD).
OPRT	Specifies state of stack flag (SF). Used only with nX-8/300.
PREG	Specifies bank number of pointing register set. It is the state of SCB3 bit of PSW.
LREG	Specifies bank number of local register set. It is the state of LRBL for nX-8/500.

Of these, DSREG, TSREG, and PAGE information will be passed to RL66K through the object file, so RL66K will check addressing that RAS66K could not check. PREG and LREG information is also passed to RL66K, which uses it to control relocatable segment allocation.

■ Attention ■

The effective range of an assumption of a USING directive is from the next source statement until a new specification with the same *register_name*. Note that this has no relation with actual program flow.

USING directives cause RAS66K and RL66K to assume register states, but do not generate object code that sets registers. Use microcontroller instructions to set the actual hardware.

4.12.14.1 Assumption Of Physical Segment Address In Data Memory Space (USING DSREG)

■ Syntax ■

```
USING DSREG status
```

■ Description ■

The USING DSREG directive informs RAS66K and RL66K of the value of the physical segment address in data memory space. The value of the physical segment address of data memory space is stored in DSR for nX-8/500 and bits 13-15 or LRB for nX-3/100.

RAS66K and RL66K check whether or not the physical segment address assumption of the USING DSREG directive matches the physical segment address in data memory specified with RAM addressing. If they do not match, then a warning will occur.

Specify one of the following for status.

status	Description
address	A general expression that represents an address in data memory space.
#phy_seg_address	A general expression that represents a physical segment address.
ANY	Do not make an assumption of the physical segment address. (Default)

An *address* is a general expression that represents an address in data memory space. RAS66K calculates the physical segment address from *address*. A *phy_seg_address* is a general expression that represents a physical segment address in data memory space. Both *address* and *phy_seg_address* can contain forward references.

RAS66K and RL66K will perform DSR checks if *address* or *phy_seg_address* is specified. They will not perform DSR checks if ANY is specified. ANY will be set until the first USING DSREG directive is encountered in the program.

■ Additional Information ■

The *address* must be an expression that represents an address in data memory space. It must not be an expression that represents an address in the COMMON area.

RAS66K will perform a DSR check when it can determine the physical segment address value set by the USING DSREG directive and the physical segment address of the instruction operand. Otherwise, RL66K will perform the DSR check.

■ Example 1 ■

```
        DSEG    #3
PHY3TBL: DS      100H

        DSEG    #4
PHY4TBL: DS      100H
        .
        .
        .


        CSEG
        USING   DSREG PHY3TBL               ;Assume DSR is 3.
        MOVB    DSR,#SEG PHY3TBL

        MOV     ER0,PHY3TBL         ;OK
        MOV     ER1,PHY4TBL         ;Warning
        .
        .
        .
        CSEG
        USING   DSREG PHY4TBL               ;Assume DSR is 4.
        MOVB    DSR,#SEG PHY4TBL

        MOV     ER0,PHY3TBL         ;Warning
        MOV     ER1,PHY4TBL         ;OK
        .
        .
        .
```

The first USING DSREG directive specifies PHY3TBL as its operand. PHY3TBL resides in physical segment 3, so RAS66K will assume a physical segment address of 3. Of the operands of the MOV instructions immediately following, PHY4TBL resides in physical segment 4, which does not match the USING DSREG assumption. Accordingly, RAS66K will issue a warning for this MOV instruction.

The second USING DSREG directive specifies PHY4TBL as its operand, so now RAS66K will assume a physical segment address of 4. Accordingly, of the MOV instructions immediately following, the one that accesses PHY3TBL will cause a warning.

■ **Example 2** ■

The following example does not check during assembly.

```
EXTRN   DATA:GL_TBL1 GL_TBL2
DAT_SEG SEGMENT DATA

    CSEG
    USING   DSREG #SEG GL_TBL1
    MOVB    DSR,#SEG GL_TBL1

    MOV     ER0,GL_TBL2         ;RL66K will check
    MOV     ER1,DAT_SEG         ;RL66K will check
```

In this example, RAS66K cannot determine the physical address of GL_TBL1 specified in the USING DSREG directive. Accordingly, RL66K will perform the DSR checks of the two MOV instructions. If the physical segment addresses do not match, then a warning will occur during linking.

4.12.14.2 Assumption Of Physical Segment Address In Program Memory Space (USING TSREG)

■ **Syntax** ■

```
USING TSREG status
```

■ **Description** ■

The USING TSREG directive informs RAS66K and RL66K of the value of the physical segment address in program memory space for referring tables. The value of the physical segment address of program memory space is stored in TSR.

RAS66K and RL66K check whether or not the physical segment address assumption of the USING TSREG directive matches the physical segment address in program memory specified with table data addressing or ROM window addressing. If they do not match, then a warning will occur.

Specify one of the following for status.

status	**Description**
address	A general expression that represents an address in program memory space.
#phy_seg_address	A general expression that represents a physical segment address.
ANY	Do not make an assumption of the physical segment address. (Default)

An *address* is a general expression that represents an address in program memory space. RAS66K calculates the physical segment address from *address*. A *phy_seg_address* is a general expression that represents a physical segment address in program memory space. Both *address* and *phy_seg_address* can contain forward references.

RAS66K and RL66K will perform TSR checks if *address* or *phy_seg_address* is specified. They will not perform TSR checks if ANY is specified. ANY will be set until the first USING TSREG directive is encountered in the program.

■ **Additional Information** ■

The address must be an expression that represents an address in program memory space.

RAS66K will perform a TSR check when it can determine the physical segment address value set by the USING TSREG directive and the physical segment address of the instruction operand. Otherwise, RL66K will perform the TSR check.

■ **Example 1** ■

```
     CSEG   #3
PHY3TBL: DW     10H,20H,30H
     CSEG   #4
PHY4TBL: DW     40H,50H,60H
     .
     .
     .
     CSEG
     USING  TSREG PHY3TBL       ;Assume TSR is 3.
     MOVB   TSR,#SEG PHY3TBL

     LC     A,PHY3TBL           ;OK
     LC     A,PHY4TBL           ;Warning
     .
     .
     .
     CSEG
     USING  TSREG PHY4TBL       ;Assume TSR is 4.
     MOVB   TSR,#SEG PHY4TBL

     LC     A,PHY3TBL           ;Warning
     LC     A,PHY4TBL           ;OK
     .
     .
     .
```

The first USING TSREG directive specifies PHY3TBL as its operand. PHY3TBL resides in physical segment 3, so RAS66K will assume a physical segment address of 3. Of the operands of the LC instructions immediately following, PHY4TBL resides in physical segment 4, which does not match the USING TSREG assumption. Accordingly, RAS66K will issue a warning for this LC instruction.

The second USING TSREG directive specifies PHY4TBL as its operand, so now RAS66K will assume a physical segment address of 4. Accordingly, of the LC instructions immediately following, the one that accesses PHY3TBL will cause a warning.

■ **Example 2** ■

The following example does not check during assembly.

```
EXTRN   CODE:GL_TBL1 GL_TBL2
ROM_SEG SEGMENT CODE

     CSEG
     USING   TSREG #REG GL_TBL1
     MOVB    TSR,#SEG GL_TBL1

     LC      A,GL_TBL2              ;RL66K will check
     LC      A,ROM_SEG              ;RL66K will check
```

In this example, RAS66K cannot determine the physical address of GL_TBL1 specified in the USING TSREG directive. Accordingly, RL66K will perform the TSR checks of the two LC instructions. If the physical segment addresses do not match, then a warning will occur during linking.

4.12.14.3 Assumption Of Current Page (USING PAGE)

■ **Syntax** ■

```
    USING PAGE status
```

■ **Description** ■

The USING PAGE directive informs RAS66K and RL66K of the page number of the current page. The page number of the current page is stored in LRBH for nX-8/500 and in bits 5-12 of LRB for nX-8/100~400.

RAS66K and RL66K check whether or not the page number assumption of the USING PAGE directive matches the page number specified current page addressing or current page SBA addressing. If they do not match, then a warning will occur.

Specify one of the following for status.

status	**Description**
address	A general expression that represents an address in data memory space.
ANY	Do not set the current page. (Default)

An address is a general expression that represents an address in data memory space. It can also be an address in program memory space when using the ROM window function. RAS66K calculates the page number from address. The address can contain forward references.

RAS66K and RL66K will perform current page checks if address is specified. They will not perform TSR checks if ANY is specified. ANY will be set until the first USING PAGE directive is encountered in the program.

■ **Additional Information** ■

RAS66K will perform a current page check when it can determine the page number set by the USING PAGE directive and the page number of the instruction operand. Otherwise, RL66K will perform the current page check.

■ **Example 1** ■

```
     DSEG   AT 2000H
P20DATA: DS     100H                ;Data area in page 20H

     DSEG   AT 3000H
P30DATA: DS     100H                ;Data area in page 30H
      .
      .
      .
```

```
    CSEG
    USING   PAGE P20DATA         ; Assume current page is page 20H
    MOVB    ALRBH,#PAGE P20DATA

    MOV     ER0,OFF P20DATA      ;OK
    MOV     ER1,OFF P30DATA      ;Warning
    .
    .
    .
    CSEG
    USING   PAGE P30DATA                   ; Assume current page is page 30H
    MOVB    ALRBH,#PAGE P30DATA

    MOV     ER1,OFF P20DATA      ;Warning
    MOV     ER0,OFF P30DATA      ;OK
    .
    .
    .
```

The first USING PAGE directive specifies P20DATA as its operand. P20DATA resides in page 20H, so RAS66K will assume a current page of 20H. Of the operands of the MOV instructions immediately following, P30DATA resides in page 30H, which does not match the USING PAGE assumption. Accordingly, RAS66K will issue a warning for this MOV instruction.

The second USING PAGE directive specifies P30DATA as its operand, so now RAS66K will assume a current page of 30H. Accordingly, of the MOV instructions immediately following, the one that accesses P20DATA will cause a warning.

■ Example 2 ■

The following example does not check during assembly.

```
EXTRN   DATA:GL_TBL1 GL_TBL2
DAT_SEG SEGMENT DATA

    CSEG
    USING   PAGE GL_TBL1
    MOVB    ALRBH,#PAGE GL_TBL1

    MOV     ER0,OFF GL_TBL2     ;RL66K will check
    MOV     ER1,OFF DAT_SEG     ;RL66K will check
```

In this example, RAS66K cannot determine the physical address of GL_TBL1 specified in the USING PAGE directive. Accordingly, RL66K will perform the current page checks of the two MOV instructions. If the physical segment addresses do not match, then a warning will occur during linking.

4.12.14.4 Assumption Of Data Descriptor (USING DATA)

■ Syntax ■

```
USING DATA status
```

■ Description ■

Many word instructions of the OLMS-66K Series will not operate correctly if the state of the data descriptor (DD) is not word (1). Many byte instructions will not operate correctly if the state of DD is not byte (0).

The USING DATA directive informs RAS66K of the state of DD. RAS66K checks if instructions in the program are affected by DD, and for those that are, checks if they are being used with the appropriate state of DD. If they are not, then RAS66K will issue a warning. These are called flag attribute checks of instructions affected by DD.

Specify one of the following for status.

status	**Description**
WORD	DD state is word (1).
BYTE	DD state is byte (0).
ANY	Do not perform DD flag attribute checks. (Default)

RAS66K will perform DD flag attribute checks if WORD or BYTE is specified. It will not perform DD flag attribute checks if ANY is specified. ANY will be set until the first USING DATA directive is encountered in the program.

■ Additional Information ■

Refer to the instruction manual of the target microcontroller for instructions affected by DD and instructions that change DD.

The DD assumption of USING DATA directives will affect flag attribute checks of branch instructions. For details refer to Section 4.10.9.2, “Flag Attribute Checks of Branch Instructions,” and Section 4.12.14.6, “Flag Attribute Checks Of Branch Instructions (CHK).”

■ Example ■

```
    TYPE    (M66507)

    USING   DATA  WORD              ;Assume DD state is word
    SDD                             ;Actually set DD
    AND     A,#0F800H               ;Word instruction is OK
    ANDB    A,#0F8H         ;Byte instruction causes warning
;
;
    USING   DATA  BYTE              ;Assume DD state is byte
    RDD                             ;Actually reset DD
    AND     A,#0F800H               ;Word instruction causes warning
    ANDB    A,#0F8H         ;Byte instruction is OK
;
;
    USING   DATA  ANY       ;Clear assumption of DD state
    AND     A,#0F800H               ;Check not performed
    ANDB    A,#0F8H         ;Check not performed
```

In this example, the first USING DATA directive assumes the state of DD is word. In this state, only word instructions can be used from instructions affected by DD. Accordingly, RAS66K will issue a warning for the ANDB instruction.

The second USING DATA directive assumes the state of DD is byte. In this state, only byte instructions can be used from instructions affected by DD. Accordingly, RAS66K will issue a warning for the AND instruction.

Finally, USING DATA ANY is specified. This declares that no DD assumption should be made. In this state, checks of instructions affected by DD are not performed. Accordingly, RAS66K will not issue warnings for the AND and ANDB instructions.

4.12.14.5 Assumption Of Stack Flag (USING OPRT)

■ Syntax ■

```
USING OPRT status
```

■ Description ■

Many instructions for manipulating the user stack of nX-8/300-based microcontrollers will not operate correctly if the state of the stack flag (SF) is not 1. Many instructions for manipulating the accumulator will not operate correctly if the state of SF is not 0.

The USING OPRT directive informs RAS66K of the state of SF. RAS66K checks if instructions in the program are affected by SF, and for those that are, checks if they are being used with the appropriate state of SF. If they are not, then RAS66K will issue a warning. These are called flag attribute checks of instructions affected by SF.

Specify one of the following for status.

status	**Description**
STACK	SF state is 1.
A	SF state is 0.
ANY	Do not perform SF flag attribute checks. (Default)

RAS66K will perform SF flag attribute checks if WORD or BYTE is specified. It will not perform SF flag attribute checks if ANY is specified. ANY will be set until the first USING OPRT directive is encountered in the program.

■ Additional Information ■

The USING OPRT directive can be used only with nX-8/300.

Refer to the instruction manual of the target microcontroller for instructions affected by SF and instructions that change SF.

The SF assumption of USING OPRT directives will affect flag attribute checks of branch instructions. For details refer to Section 4.10.9.2, “Flag Attribute Checks of Branch Instructions,” and Section 4.12.14.6, “Flag Attribute Checks Of Branch Instructions (CHK).”

■ **Example** ■

```
    TYPE   (M66301)

    USING  OPRT  STACK      ;Assume SF state is 1
    SB     SF               ;Actually set SF
    PUSHU  X1                      ;OK
    L      A,X1        ;Warning
;
;
    USING  OPRT  A          ;Assume SF state is 0
    RB     SF               ;Actually reset SF
    PUSHU  X1                      ;Warning
    L      A,X1        ;OK
;
;
    USING  OPRT  ANY               ;Clear assumption of SF state
    PUSHU  X1                      ;Check is not performed
    L      A,X1        ;Check is not performed
```

In this example, the first USING OPRT directive assumes the state of SF is STACK. In this state, only user stack manipulation instructions can be used from instructions affected by SF. Accordingly, RAS66K will issue a warning for the L A,X1 instruction.

The second USING OPRT directive assumes the state of SF is A. In this state, only accumulator manipulation instructions can be used from instructions affected by SF. Accordingly, RAS66K will issue a warning for the PUSHU X1 instruction.

Finally, USING OPRT ANY is specified. This declares that no SF assumption should be made. In this state, checks of instructions affected by SF are not performed. Accordingly, RAS66K will not issue warnings for the PUSHU and L instructions.

4.12.14.6 Flag Attribute Checks Of Branch Instruction (CHK)

■ **Syntax** ■

```
   CHK
```

■ **Description** ■

The CHK directive instructs RAS66K to perform flag attribute checks of branch instructions.

When the CHK directive is specified, RAS66K will check if the flag attributes (DD and SF states) of branch destinations match those of branch sources when it encounters branch instructions. If they do not match, RAS66K will issue a warning.

The flag attributes of the branch source will be the flag value assumptions from the USING DATA directive and USING OPRT directive at the point where the branch instruction is coded.

The flag attributes of the branch destination will be the flag attributes held by the symbol specified as the operand of the branch instruction (the label or subroutine name representing the branch destination).

■ **Example** ■

```
    CHK                        ;Check flag attributes of branch instructions.
    USING  DTA WORD            ;Assume state of DD is word.
    CAL    PROC1       ;DD states match, so OK.
    CAL    PROC2       ;DD state differs, so warning occurs.
    CAL    PROC3       ;Check not performed.
    .
    .
    .
    USING  DATA  WORD  ;Assume state of DD is word.
PROC1:
    .
    .
    .
    USING  DATA  BYTE  ;Assume state of DD is byte.
PROC2:
    .
    .
    .
    USING  DATA  ANY   ;Clear assumption of DD state.
PROC3:
    .
    .
    .
```

In this example, three subroutines are called while the state of DD is assumed to be word. In the first CAL instruction, the DD state of branch destination PROC1 is word, which that of the branch source, so no warning will occur. In the second CAL instruction, the DD state of branch destination PROC2 is byte, which does not match that of the branch source, so a warning will occur. In the third CAL instruction, there is no assumption of the DD state of branch destination PROC3, so no warning will occur.

4.12.14.7 Assumption Of Pointing Register Set Bank Number (USING PREG)

■ **Syntax** ■

```
USING PREG bank_no
```

■ **Description** ■

The USING PREG directive informs RAS66K of the bank number of the pointing register set being used. The bank number of the pointing register set is stored in the SCB bit of PSW.

The *bank_no* is a constant expression that represents the bank number of the pointing register set. It must not include forward references. Its value must be 0 to 7.

When the USING PREG directive is specified, RAS66K resets the address value of each symbol representing a pointing register address to a value that corresponds to the specified bank number. More specifically, symbol values are determined as follows.

Symbol	**Assumed Value**
AX1	*preg_base* + 8**bank_no* +0
AX2	*preg_base* + 8**bank_no* +2
ADP	*preg_base* + 8**bank_no* +4
AUSP	*preg_base* + 8**bank_no* +6

The *preg_base* is the base address of the pointing register area. It is 80H for nX-8/100~400, and 200H for nX-8/500.

USING PREG 0 will be assumed until the first USING PREG directive is encountered.

The assumed value of the USING PREG directive will also affect RL66K link processing. Information about bank numbers of the pointing register set specified by USING PREG directives is passed to RL66K through the object file. RL66K will not allocate segments over the areas corresponding to those bank numbers.

■ **Additional Information** ■

The PRBANK and NOPRBANK directives are pointing register directives. They inform RL66K about the pointing register bank used in the program. RAS66K passes a combination of information from these directives and the USING PREG directive to RL66K.

■ Example ■

The example below is for nX-8/500.

```
TYPE  (M66507)

USING PREG  0                   ;Select bank 0.
ANDB  PSWL,#1111_1000B          ;Set SCB to 0.
L     A,AX1                     ;200H
L     A,AX2                     ;202H
L     A,ADP                     ;204H
L     A,AUSP                    ;206H

USING PREG  3                   ;Select bank 3.
ANDB  PSWL,#1111_1000B
ORB   POWL,#0000_0011B          ;Set SCB to 3.
L     A,AX1                     ;218H
L     A,AX2                     ;21AH
L     A,ADP                     ;21CH
L     A,AUSP                    ;21EH
```

The first USING PREG directive selects bank 0, so the addresses of AX1, AX2, ADP, and AUSP will be assumed to be 200H, 202H, 204H, and 206H, respectively. The second USING PREG directive selects bank 3, so the addresses of AX1, AX2, ADP, and AUSP will be assumed to be 218H, 21AH, 21CH, and 21EH, respectively.

4.12.14.8 Assumption of Local Register Set Bank Number (USING LREG)

■ **Syntax** ■

```
USING LREG bank_no
```

■ **Description** ■

The USING LREG directive manages the bank number of the nX-8/500 local register set being used. This directive cannot be used with nX-8/100~400.

The USING PREG directive informs RAS66K of the bank number of the pointing register set being used. The bank number of the local register set is stored in LRBL.

The bank_no is a constant expression that represents the bank number of the local register set. It must not include forward references. Its value must be 0 to 255.

When the USING LREG directive is specified, RAS66K resets the address value of each symbol representing a local register address to a value that corresponds to the specified bank number. More specifically, symbol values are determined as follows.

Symbol	Assumed Value
AER0	*lreg_base* + 8**bank_no* +0
AER1	*lreg_base* + 8**bank_no* +2
AER2	*lreg_base* + 8**bank_no* +4
AER3	*lreg_base* + 8**bank_no* +6
AR0	*lreg_base* + 8**bank_no* +0
AR1	*lreg_base* + 8**bank_no* +1
AR2	*lreg_base* + 8**bank_no* +2
AR3	*lreg_base* + 8**bank_no* +3
AR4	*lreg_base* + 8**bank_no* +4
AR5	*lreg_base* + 8**bank_no* +5
AR6	*lreg_base* + 8**bank_no* +6
AR7	*lreg_base* + 8**bank_no* +7

The *lreg_base* is the base address of the local register area. It is 200H.

RAS66K will not make a local register bank assumption until the first USING LREG directive is encountered. Accordingly, the above symbols representing local register addresses cannot be used unless a USING LREG directive is specified.

The assumed value of the USING LREG directive will also affect RL66K link processing. Information about bank numbers of the local register set specified by USING LREG directives is passed to RL66K through the object file. RL66K will not allocate segments over the areas corresponding to those bank numbers.

■ Additional Information ■

The LRBANK and NOLRBANK directives are local register directives. They inform RL66K about the local register bank used in the program. RAS66K passes a combination of information from these directives and the USING LREG directive to RL66K.

■ Example ■

```
 TYPE  (M66507)

USING LREG  10H       ;Select bank 10H.
MOVB  ALRBL,#10H             ;Set LRBL to 10H.
L    A,AER0           ;280H
L    A,AER1           ;282H
L    A,AER2           ;284H
L    A,AER3           ;286H

USING LREG  20H       ;Select bank 20H.
MOVB  ALRBL,#20H             ;Set LRBL to 20H.
L    A,AER0           ;300H
L    A,AER1           ;302H
L    A,AER2           ;304H
L    A,AER3           ;306H
```

The first USING LREG directive selects bank 10H, so the addresses of AER0, AER1, AER2, and AER3 will be assumed to be 280H, 282H, 284H, and 286H, respectively. The second USING LREG directive selects bank 20H, so the addresses of AER0, AER1, AER2, and AER3 will be assumed to be 300H, 302H, 304H, and 306H, respectively.

4.12.15 Include File (INCLUDE)

■ Syntax ■

```
INCLUDE (include_file)
```

■ Description ■

The contents of an include file will be inserted at the position of the INCLUDE directive. The *include_file* specifies the include file to be inserted.

The inserted file is placed from its start at the directive's location. Additional files can be inserted by using further INCLUDE directives within an include file. INCLUDE directives can be nested up to 8 levels.

If an END directive is specified in an include file, then all further contents in that include file will be ignored.

■ Additional Information ■

The directory storing include files can be specified using the /I option.

■ Example ■

```
/*
  Source File
*/
  INCLUDE(DEFINE.H)

  CSEG
  ORG     RESET
  DW      START
  .
  .
```

```
/*
  Include File(DEFINE.H)
*/

  TYPE(M66507)

  PAGE  60,100
  SYM
  REF
  ERR(ERR.MES)
```

In this example, print file control and error message output control are coded within the include file.

4.12.16 Program Termination (END)

■ **Syntax** ■

```
   END
```

■ **Description** ■

The END directive indicates the end of the program. RAS66K assembles until it encounters an END directive. RAS66K will ignore any source statements after an END directive.

However, if there is an END directive within an include file, then all further contents of that file will be ignored. Contents coded after an END directive will not be output to the print file.

■ **Example** ■

```
  TYPE(M66507)
  .
  .
  .
  END
This part isn't read.
```

In this example, code with incorrect syntax for assembly language follows the END directive. RAS66K ignores this portion, so no error will occur.

4.12.17 Module Name Setting (NAME)

■ **Syntax** ■

```
   NAME (module_name)
```

■ **Description** ■

A module name is the name of one object file.

In older versions of RAS66K, module names could be changed using the NAME directive. However, in the current RAS66K module names are the base names of source files, so they cannot be changed.

The NAME directive exists only for compatibility with previously created source program. The NAME directive itself has no function. Therefore, if you are creating a new OLMS-66K program, there is no need to code a NAME directive.

4.12.18 Register Bank Declarations

This section explains the directives that declare the bank numbers of pointing register sets and local register sets used in programs.

The information from these directives is passed to RL66K through object files. RL66K will not allocate relocatable segments over pointing register set and local register set areas used by programs. This prevents incorrect operation where register contents are corrupted from overlapping relocatable segments and register areas. Refer to Section 5.5.4.2, “Quasi-Segments,” to see how RL66K handles pointing register areas and local register areas.

4.12.18.1 Pointing Register Bank Declaration (PRBANK, NOPRBANK)

■ Syntax ■

```
PRBANK bank_no [,bank_no … ]
NOPRBANK
```

■ Description ■

These directives declare the bank numbers of pointing register sets used in the program.

The PRBANK directive declares that pointing register sets of the bank indicated by bank_no are used in the program. Each bank_no must be a constant expression that represents a bank number 0 to 7, and that does not include forward references. Multiple bank_no parameters may be specified in a single PRBANK directive, and multiple PRBANK directives may be specified within one program.

The NOPRBANK directive declares that there is no available information about bank numbers of pointing register sets used in the program. In other words, RL66K will not control allocation of logical segments with regard to pointing register areas.

■ Additional Information ■

Both the PRBANK and NOPRBANK directives cannot be specified within one program. If the PRBANK, NOPRBANK, and USING PREG directives are not specified, then RL66K will be passed information that only bank 0 was used.

Information about the settings of the PRBANK, NOPRBANK, and USING PREG directives is passed to RL66K. If a USING PREG directive is specified in the program, then the NOPRBANK directive will be ignored even when specified.

4.12.18.2 Local Register Bank Declaration (LRBANK, NOLRBANK)

■ Syntax ■

```
LRBANK bank_no [,bank_no …]
NOLRBANK
```

■ Description ■

These directives are used to control allocation of logical segments in the presence of nX-8/500 local registers. These directives cannot be used with nX-8/100~400.

The LRBANK directive declares that local register sets of the bank indicated by *bank_no* are used in the program. Each *bank_no* must be a constant expression that represents a bank number 0 to 255, and that does not include forward references. Multiple *bank_no* parameters may be specified in a single LRBANK directive, and multiple LRBANK directives may be specified within one program.

The NOLRBANK directive declares that there is no available information about bank numbers of local register sets used in the program. In other words, RL66K will not control allocation of logical segments with regard to local register areas.

■ Additional Information ■

Both the LRBANK and NOLRBANK directives cannot be specified within one program. If the LRBANK, NOLRBANK, and USING PREG directives are not specified, then RL66K will not control allocation of logical segments with regard to local register areas.

Information about the settings of the LRBANK, NOLRBANK, and USING PREG directives is passed to RL66K. If a USING PREG directive is specified in the program, then the NOLRBANK directive will be ignored even when specified.

■ Example ■

```
TYPE  (M66507)

PRBANK   0,1,2
LRBANK    10H,11H,12H
```

This example declares the use of pointing register banks 0-2 and local register banks 10H-12H. As a result, RL66K will not allocate segments in 200H-217H and 280H-297H of data memory space.

4.12.19 Conditional Assembly (IF, IFDEF, IFNDEF, ELSE, ENDIF)

By using conditional assembly functions, the programmer can control assembly such that particular program blocks are assembled only when certain conditions are met. This allows a single source program to be used for multiple purposes.

Conditional assembly is implemented by coding conditional assembly directives. The syntax of conditional assembly directives is as follows.

```
IFxxx  conditional_operand
      true_conditional_body
ENDIF
```

It can also be as below.

```
IFxxx  conditional_operand
      true_conditional_body
ELSE
      false_conditional_body
ENDIF
```

Here "IFxxx" indicates one of the following conditional assembly directives.

```
      IF      IFDEF      IFNDEF
```

The *conditional_operand* is a symbol or expression giving a true/false condition for conditional assembly. The contents specified for *conditional_operand* differ depending on the conditional assembly directive.

The *true_conditional_body* and *false_conditional_body* represent blocks of source statements. If the condition is true, then the source statement block of the *true_conditional_body* will be assembled. If the condition is false, then the source statement block of the *true_conditional_body* will be skipped. When there is an ELSE directive, the *false_conditional_body* will be assembled.

Further conditional assembly directives may be coded within the *true_conditional_body* and *false_conditional_body*. Conditional assembly directives can be nested up to 15 levels deep.

4.12.19.1 Conditional Assembly On Expression Value (IF)

■ Syntax ■

```
    IF expression
```

■ Description ■

The *expression* is a constant expression that does not include forward references.

If the value of *expression* is non-zero, then it will be considered true. If the value of *expression* is zero, then it will be considered false. The *expression* will also be considered false if it includes forward references or contains syntax errors.

■ Example ■

```
;PROG_SW  EQU   0
;PROG_SW  EQU   1
 PROG_SW  EQU   2
          .
          .
          .
IF   PROG_SW==2

BUF_SIZE1 EQU   100H
BUF_SIZE2 EQU   200H

ELSE

BUF_SIZE1 EQU   200H
BUF_SIZE2 EQU   400H

ENDIF

     DSEG  AT 1000H
BUF1:     DS    BUF_SIZE1
BUF2:     DS    BUF_SIZE2
```

In this example, “PROG_SW==2” is specified as the conditional expression. PROG_SW is assigned the value 2 at the start of the program, so the conditional will be true. Accordingly, BUF_SIZE1 will be set to 100H and BUF_SIZE2 will be set to 200H.

4.12.19.2 Conditional Assembly On Symbol Definition Or Non-Definition (IFDEF, IFNDEF)

■ **Syntax** ■

```
   IFDEF symbol
   IFNDEF symbol
```

■ **Description** ■

The *symbol* is any symbol other than reserved words.

For the IFDEF directive, the condition will be true if *symbol* has been defined in a previous source statement. The condition will be false if *symbol* is not defined in the program, or if it is defined in a later source statement.

The IFNDEF directive is the exact opposite of the IFDEF directive. The condition will be false if *symbol* has been defined in a previous source statement. The condition will be true if *symbol* is not defined in the program, or if it is defined in a later source statement.

■ **Example** ■

```
PROG_SW2  EQU   0
     .
     .
     .
IFDEF PROG_SW1

    INCLUDE  (INIT1.INC)
ELSE

 IFNDEF PROG_SW2

    INCLUDE  (INIT2.INC)
 ELSE
    INCLUDE  (INIT3.INC)
 ENDIF

ENDIF
```

In this example, an IFNDEF directive is nested within the false conditional block of the IFDEF directive. The PROG_SW1 operand of the IFDEF directive is not defined, so this condition will be false. The PROG_SW2 operand of the next IFNDEF directive is defined, so this condition will be false. Therefore, INIT3.INC will be included.

4.12.20 Macro Definition (DEFINE)

■ Syntax ■

```
DEFINE symbol "macro_body"
```

■ Description ■

The DEFINE directive defines a macro symbol.

The DEFINE directive assigns *macro_body* to *symbol*. After this definition, RAS66K will replace *symbol* with *macro_body* whenever it appears in a source statement before assembling it.

Other macro symbols may be coded within *macro_body*. In this case, in the process of replacing the first macro, RAS66K will replace the nested macro. Macros are permitted to be nested 8 levels deep.

■ Additional Information ■

Macro symbols can be referred only after they are defined with the DEFINE symbol.

■ Example ■

```
DEFINE  LA     "L   A,"
DEFINE  LX1    "MOV X1,"
DEFINE  ROMSEG "SEGMENT CODE WINDOW"
;
    LA     ER0    ;L   A,ER0
    LA     #100H  ;L   A,100H
    LX1    [DP]   ;MOV X1,[DP]
    LX1    LRB    ;MOV X1,LRB
;
ROMSEG1 ROMSEG     ;ROMSEG1 SEGMENT CODE WINDOW
ROMSEG2 ROMSEG     ;ROMSEG2 SEGMENT CODE WINDOW
```

In this example, LA is replaced by "L A," LX1 by "MOV X1," and ROMSEG by "SEGMENT CODE WINDOW." RAS66K interprets the code using macros as shown in the comments.

■ Attention ■

Care is required when using macro symbols and conditional assembly directives together. Examine the example program below.

```
DEFINE SW       "SYM1"

IFDEF  SW
   INCLUDE  (FILE1)
ELSE
   INCLUDE  (FILE2)
ENDIF
```

In this program, “IFDEF SW” is written to mean, “if symbol SW is defined.” However, SW is actually a macro symbol, so RAS66K will judge whether SYM1, which is the macro body of SW, is defined or not as the condition. SYM1 is not defined, so the condition will be false and FILE2 will be included.

Thus, even if a macro symbol is specified as the operand of an IFDEF or IFNDEF directive, RAS66K will interpret the operand as the string assigned to the macro symbol.

4.12.21 C Source Level Debug Information (CFILE, CFUNCTION, CLINE)

■ Syntax ■

```
CFILE expression
CFUNCTION expression
CLINE expression
```

■ Description ■

These directives are automatically generated by the C compiler CC66K. They are not used by the programmer. The information given by these directives is used for C source level debugging by CDB66K.

The CFILE directive gives information about C source program files.

The CFUNCTION directive gives information about C source program functions.

The CLINE directive gives information about C source program line numbers.

If these directives are written in normal assembly source, or if they are included in a source program generated by CC66K and replaced, then there is no guarantee that correct assembly results will be obtained or that following debugging operations will be correct.

4.12.22 Optimization Of Branch Instructions

OLMS-66K provides several jump instructions and subroutine call instructions. If GJMP or GCAL directives are used instead of directly coding the microcontroller instructions, then RAS66K will convert them to the optimal instructions corresponding to the address value of the branch destination or distance to the branch destination.

4.12.22.1 Optimization Of Jump Instructions (GJMP)

■ **Syntax** ■

```
    GJMP symbol
```

■ **Description** ■

The GJMP directive converts to an optimal jump instruction.

The *symbol* is a symbol representing the jump destination. RAS66K converts the GJMP directive to the optimal jump instruction corresponding to the jump destination represented by *symbol*. However, if *symbol* is a forward reference, then it will not be optimized, but instead will be converted to the jump instruction with the longest machine code.

For nX-8/100~400, each GJMP directive will be converted to either an SJ instruction or J instruction.

For nX-8/500, each GJMP directive will be converted to either an SJ instruction, J instruction, or FJ instruction. However, if there is only one physical segment in program memory space, or if the SMALL or COMPACT memory model is being used, then there will be no conversions to the FJ instruction.

■ **Example** ■

```
     CSEG   AT  1000H
LABEL1:
     .
     .
     .
     CSEG   AT  1040H
     GJMP    LABEL1          ;Converts to SJ instruction.
     .
     .
     .
     CSEG   AT  2000H
     GJMP    LABEL1          ;Converts to J instruction.
```

In this example, the distance from the first GJMP instruction to LABEL1 is in the range -128~+127, so it will be converted to an SJ instruction. The distance from the second GJMP instruction to LABEL1 is not in the range -128~+127, so it will be converted to a J instruction.

4.12.22.2 Optimization Of Call Instructions (GCAL)

■ **Syntax** ■

```
    GCAL symbol
```

■ **Description** ■

The GCAL directive converts to an optimal subroutine call instruction.

The *symbol* is a symbol representing the subroutine. RAS66K converts the GCAL directive to the optimal call instruction corresponding to the call destination represented by *symbol.* However, if *symbol* is a forward reference, then it will not be optimized, but instead will be converted to the call instruction with the longest machine code.

For nX-8/100~400, each GCAL directive will be converted to either an SCAL instruction or CAL instruction.

For nX-8/500, each GCAL directive will be converted to either an ACAL instruction, CAL instruction, or FCAL instruction. However, if there is only one physical segment in program memory space, or if the SMALL or COMPACT memory model is being used, then there will be no conversions to the FCAL instruction.

■ **Example** ■

```
     TYPE (M66507)

     CSEG   AT   1000H        ;ACAL area
SUB1:
     .
     .
     .
     CSEG   AT   2000H
SUB2:
     .
     .
     .
     CSEG   AT   3000H
;
     GCAL   SUB1              ;Converts to ACAL instruction.
     GCAL   SUB2              ;Converts to CAL instruction.
```

In this example, the call destination SUB1 of the first GCAL instruction is in the ACAL area, so it will be converted to an ACAL instruction. The call destination SUB2 of the second GCAL instruction is not in the ACAL area, so it will be converted to an CAL instruction.

4.12.23 Print File Control

The print file consists of three lists.

- Assembly list
- Symbol list
- Cross-reference list

Refer to Section 4.13, “Print Files,” for details about print files. The following directives can be used to control the creation and format of print files.

4.12.23.1 Print File Output Control (PRN, NOPRN)

■ Syntax ■

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

■ Description ■

The PRN directive generates a print file. The *print_file* specifies the print file name. If the operand is omitted or part of the file specification is omitted, then refer to Section 4.2, “File Specification Defaults,” for defaults.

The NOPRN directive suppresses generation of a print file.

If both the PRN and NOPRN directives are omitted, then RAS66K will generate a print file. In this case the print file name will be the same as the source file name with extension changed to “.PRN.” Multiple PRN and NOPRN directives can be used, but only the first will be valid.

■ Example ■

```
PRN(OUTPUT.LST)
```

This example specifies the generation of a print file OUTPUT.LST.

4.12.23.2 Force Page Break (PAGE without operand)

■ Syntax ■

```
   PAGE
```

■ Description ■

A PAGE directive without an operand forces a page break in the print file. It will move output from the line containing the PAGE directive to the next page. However, PAGE directives will be ignored in ranges where NOLIST directives suppress output of the assembly list.

■ Additional Information ■

The PAGE directive has two functions. A page break is performed when the PAGE directive is specified without an operand. If an operand is specified with the PAGE directive, then the directive has the function of specifying the characters per line and lines per page in the print file.

■ Example ■

```
  CSEG
  .
  .
  .
  RT
;
  PAGE                ;Page break here.
  CSEG  AT  2000H
```

4.12.23.3 Lines Per Page and Characters Per Line Specification (PAGE with operands)

■ **Syntax** ■

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

■ **Description** ■

A PAGE directive with operands specifies the characters per line and lines per page in the print file. The *page_length* specifies the lines per page, and the *page_width* specifies the characters per line. Both *page_length* and *page_width* are constant expressions that do not include forward references.

Just one of *page_length* or *page_width* may be specified, but if both are omitted and only PAGE is specified, then it will force a page break.

The range of values of *page_length* is 10 to 65535. Values specified as less than 10 will be recognized as 10, while values specified as greater than 65535 will be recognized as 65535. The initial value of *page_length* is 60.

The range of values of *page_width* is 79 to 132. Values specified as less than 79 will be recognized as 79, while values specified as greater than 132 will be recognized as 132. The initial value of *page_width* is 79.

If multiple PAGE directives with operands are used in a program, then only the settings of the first directive will be valid.

■ **Example** ■

```
PAGE  70 , 130
```

This example sets the print file lines per page to 70, and the characters per line to 130.

4.12.23.4 Title Specification (TITLE)

■ **Syntax** ■

```
    TITLE "character_string"
```

■ **Description** ■

The TITLE directive specifies the print file title. The title is output in the header of each page of the print file.

The *character_string* specifies the string for the title. The title can be up to 70 characters. If more than 70 characters are specified in the *character_string*, then the excess characters will be ignored.

Multiple TITLE directives can be used in a program, but only the last one used will be valid. If the TITLE directive is omitted, then nothing will be output as the title in the print file header.

■ **Example** ■

```
  TITLE   "Communication Program"
```

This example specifies "Communication Program" as the title.

4.12.23.5 Date Specification (DATE)

■ **Syntax** ■

```
    DATE "character_string"
```

■ **Description** ■

The DATE directive specifies the string to be output in the date field of the print file.

The *character_string* can be up to 25 characters. If more than 25 characters are specified in the *character_string*, then the excess characters will be ignored.

Multiple DATE directives can be used in a program, but only the last one used will be valid. If the DATE directive is omitted, then the date when RAS66K was invoked will be output in the print file's date field.

■ **Example** ■

```
  DATE   "Jun 1, 1993"
```

This example outputs "Jun 1, 1993" in the print file's date field.

4.12.23.6 Assembly List Output Control (LIST, NOLIST)

■ Syntax ■

```
LIST
NOLIST
```

■ Description ■

The assembly list is a list of the program and its corresponding object code. Which ranges of a program to be output to the assembly list is specified using LIST and NOLIST directives.

Lines following a LIST directive in the program will be output to the assembly list. Lines will be output to the assembly list until RAS66K encounters a NOLIST directive.

Lines following a NOLIST directive in the program until the next LIST directive will not be output to the assembly list. However, if a source statement generates an error or warning even in a range which is not being output to the assembly list, then it will be output to the assembly list. If there is a PAGE directive for a page break in a range which is not being output to the assembly list, then that PAGE directive will be ignored.

RAS66K assembles as if a LIST directive is specified at the start of the program. Therefore, if no LIST or NOLIST directive is used, then the entire program will be output to the assembly list.

■ Example ■

```
TYPE(M66507)
NOLIST              ;Do not output following lines to assembly list.
.
.
.
MOV    R0,R1        ;Error, so this will be output to assembly list.
.
.
.
LIST         ;Output following lines to assembly list.
.
.
.
END
```

4.12.23.7 Symbol List Output Control (SYM, NOSYM)

■ Syntax ■

```
SYM
NOSYM
```

■ Description ■

The symbol list is a list of user symbols used in the program and their contents. The SYM and NOSYM directives specify whether or not to output to the symbol list.

When the SYM directive is specified, information about all user symbols will be output to the symbol list. When the NOSYM directive is specified, a symbol list will not be generated.

If neither the SYM nor NOSYM directive specified, then the symbol list will not be output. Multiple SYM and NOSYM directives may be used in a program, but only the first specification will be valid.

■ Example ■

```
TYPE(M66507)

SYM
```

This example will output a symbol list.

4.12.23.8 Cross-Reference List Output Control (REF, NOREF)

■ **Syntax** ■

```
   REF
   NOREF
```

■ **Description** ■

The cross-reference list is a list of the lines where each user symbol is defined and used in the program. Which user symbols are to be output to the cross-reference list is specified using REF and NOREF directives.

User symbols defined or referred following a REF directive in the program until the next NOREF directive will generate a cross-reference list. User symbols defined or referred following a NOREF directive until the next REF directive will not generate a cross-reference list.

RAS66K assembles as if a NOREF directive is specified at the start of the program. Therefore, if no REF directive is used, then no cross-reference list will be generated.

■ **Example** ■

```
     TYPE(M66507)

     REF              ;Output following symbol line numbers to cross-reference list.

SYM3    EQU     1
     CSEG
LABEL3: DW      SYM4

     NOREF     ;Do not output following symbol line numbers to cross-reference list.

SYM4    EQU     2
LABEL4: NOP
```

In this example, a cross-reference list will be generated for SYM3, SYM4, and LABEL3.

4.12.23.9 Tab Code Replacement (TAB)

■ Syntax ■

```
TAB [ tab_width ]
```

■ Description ■

The TAB directive will replace tab codes used in the program with the appropriate number of space characters and output them to the assembly list. Specify the TAB directive when using a printer that does not recognize tab codes.

The *tab_width* is the number of spaces corresponding to a single tab code. It can be in the range 1 to 16. If the *tab_width* is omitted, then RAS66K will assume 8.

Multiple TAB directives can be used in a program, but only the first will be valid. If the TAB directive is omitted, then tab codes will be output as is to the print file.

■ Example ■

```
TAB 4
```

In this example, tab codes used in the program will be replaced by up to 4 bytes of spaces and output to the assembly list.

4.12.24 Object File Control

RAS66K stores program code in binary format into the object file that it generates. When debugging information is also stored in the object file, symbolic debugging of the program is possible.

The following directives can be used to control object file generation and output of debugging information to the object file.

4.12.24.1 Object File Output Control (OBJ, NOOBJ)

■ Syntax ■

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

■ Description ■

The OBJ directive generates an object file. The *object_file* specifies the object file name. If the operand is omitted or if part of the file specification is omitted, then refer to Section 4.2, “File Specification Defaults,” for the defaults.

The NOOBJ directive suppresses generation of an object file.

If both OBJ and NOOBJ directives are omitted, then an object file will be generated. In this case the object file name will be the same as the source file name with the extension changed to “.OBJ.” Multiple OBJ and NOOBJ directives can be used in a program, but only the first will be valid.

■ Example ■

```
OBJ(OUTPUT.OBJ)
```

This examples specifies generation of an object file (OUTPUT.OBJ).

4.12.24.2 Assembly Level Debugging Information Output Control (DEBUG, NODEBUG)

■ Syntax ■

```
DEBUG
NODEBUG
```

■ Description ■

The DEBUG directive outputs assembly level debugging information to the object file. Symbolic debugging of the program is possible when the object file includes debugging information.

The NODEBUG directive suppresses output of assembly level debugging information to the object file.

If no DEBUG directive is specified, then debugging information will not be output to the object file. Multiple DEBUG and NODEBUG directives can be used, but only the first will be valid.

■ Example ■

```
TYPE (M66507)

DEBUG
```

This example outputs debugging information.

4.12.25 Error Message Output Control (ERR, NOERR)

Error messages are output to the screen or to an error file. The error message output destination is controlled with the following directives.

■ **Syntax** ■

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

■ **Description** ■

The ERR directive tells RAS66K the output destination for error messages. The *error_file* specifies the error file to which error messages will be output. If the operand is omitted or if part of the file specification is omitted, then refer to Section 4.2, “File Specification Defaults,” for the defaults.

The NOERR directive tells RAS66K to not display error messages on the screen (standard output).

If both ERR and NOERR directives are omitted, then error messages will be output to the screen. Multiple ERR and NOERR directives can be used in a program, but only the first will be valid.

■ **Additional Information** ■

By using the ERR and NOERR directives, output destination can be controlled only for assembler error messages and warnings. To output fatal error messages and internal processing error messages to a file, use the DOS redirection function.

■ **Example** ■

```
ERR(ERROR.LST)
```

This example specifies the generation of an error file ERROR.LST.

4.13 Print Files

This section explains the format of print files generated by RAS66K and how to read them. A print file consists of the following lists.

- Assembly list

  The assembly list is a list of the program and its corresponding object code.

- Cross-reference list

  The cross-reference list shows the lines where each user symbol was defined and referred.

- Symbol list

  The symbol is a list of information about the user symbols used in the program.

- Termination message

  The termination message shows the total number of errors and warnings detected during assembly and information about address space.

Print file output can be controlled by using the options and directives below.

Table 4-37. Directives And Options For Print File Output

	Directives	Options
Print file	PRN, NOPRN	/PR, /NPR
Assembly list	LIST, NOLIST	/L, /NL
Cross-reference list	REF, NOREF	/R, /NR
Symbol list	SYM, NOSYM	/S, /NS

4.13.1 How To Read Assembly Lists

An example of an assembly list is shown below. To simplify explanations, numbers are shown to the left of the assembly list.

```
1)   RAS66K (MSM66507) Relocatable Assembler, Ver.4.21 assemble list. page:   1
2)    Source File: SAMPLE.asm
3)    Object File: SAMPLE.obj
4)    Data  : '93 05/13 Thu.[16;53]
5)    Title :
6)   ## Loc.Object                  Line   Source Statements

                                    1   ;********************************
                                    2   ; sample program
                                    3   ;********************************
                                    4
                                    5          TYPE (M66507)
                                    6
                                    7  ;--------------------------------
7)   -------- I N C L U D E ---------    8         INCLUDE (DEFINE.H)
                                    9  ;  include
                                   10        TAB    8
                                   11         PAGE    60, 80
                                   12        SYM
                                   13        REF
                                   14        ERR
                                   15           EXTRN   NUMBER: EXT_NUM
                                   16  ;   end of include
                                   17
8)   ------  END OF INCLUDE  --------
9)     --------------------------      18         CSEG    AT 2000H
10)    -------- U S I N G -------      19         USING   PAGE    ANY
       -------- U S I N G -------      20         USING   DATA    ANY
                                   21  ;--------------------------------
┌    00:2000                            22  LABEL:
│      -------------------------     23        DSEG
│    00:0200                           24         DS      100H
11)    -------------------------     25        BSEG
│    00:01000 (0200.0)                  26          DBIT    200H
└    00:01000 (0200.0)                  27          ORG     200H.0
                                   28  ;--------------------------------
┌      -------------------------     29        CSEG
│    00:2000 F8 FF-FF                  30         L       A,#0FFFFH
12)  00:2003 F9 FF                     31         LB      A,#0FFH
│    00:2005 F8 00-00'                  32         L       A,#EXT_NUM
└    00:2008 F9 00'                    33         LB      A,#EXT_NUM
                                   34  ;--------------------------------
┌       = 0000FFFFH                   35  NUM_SYM        EQU     0FFFFH
13)     = 00:FFFFH                    36  DATA_SYM       DATA    0FFFFH
└       = 00:7FFFFH (FFFFH.7)          37  BIT_SYM        BIT     0FFFFH.7
                                   38  ;--------------------------------
┌      -------------------------     39        CSEG
│    00:200A 01 02 03 04 05 06         40          DB      1,2,3,4,5,6,
│    00:2010 07 08 09 0A              >>>   7,8,9,10
14)  00:2014 01-00 02-00 03-00         41          DW      1,2,3,
│    00:201A 04-00 05-00 06-00          >>>   4,5,6,
└    00:2020 07-00 08-00              >>>   7,8
                                   42  ;--------------------------------
                                   43        L      A, A
┌     ** Error 00: bad operand
│    00:2024 87 FF-FF                  44         L       A, BIT_SYM
15)   ** Warning 12: usage type mismatch
└     ** Error 29: out of range
                                   45
```

Each of the numbers to the left of the assembly list explained next.

1) The target microcontroller type, assembler version, and page number are shown at the top of each page.

2) This is the source file name.

3) This is the object file name.

4) This is the date of assembly.

5) This is the title specified by the TITLE directive. If the TITLE directive is omitted, then nothing will be output on the title line.

6) This line shows the names of the fields in the assembly list. The description of each field is given below.

Table 4-38. Assembly List Fields

Field	**Description**
##	Displays the physical segment address. The physical segment address is shown in hexadecimal when it can be determined. It is shown as "##" when the physi cal segment attribute is COMMON. It is shown as "??" when it cannot be determined.
Loc.	Displays the location counter as a hexadecimal number. Absolute addresses are shown for absolute segments, while relative positions from the first address of the segment are shown for relocatable segments. For BIT segments, the location counter will be shown as a bit address and as an expression that uses the dot operator. 01000 (0200.0)
Object	Displays object code as hexadecimal numbers. Object code is displayed in byte units. If the object code could not be fixed at assembly time, then the undetermined data will be shown with a single quotation mark (') appended.
Line	Displays line numbers in decimal assuming include file contents are inserted in the source file.
Source statements	Displays the contents of the source file or include file.

• Error messages or special messages other than those given above may be displayed in the ## field, Loc field, and Object field.

7) This is displayed when an INCLUDE directive inserts an include file. From the next line on, the include file contents will be shown in the Source Statements field.

8) This is displayed after the entire include file has been displayed.

9) This is displayed when the segment is changed with a CSEG, DSEG, or RSEG directive.

10) This is output on lines where a USING directive is specified.

11) This displays the location of each statement where a label, DS directive, DBIT directive, or ORG directive is coded.

12) This shows the object code. Two bytes of object code are connected by a hyphen (-). If the object code could not be determined by RAS66K, then a single quotation mark (') will be appended to the right of the undetermined object code.

13) This shows the value (or address) assigned to the symbol when a local symbol definition directive (EQU, SET, CODE, DATA, EDATA, CBIT, BIT, EBIT) is coded.

14) This displays 6 bytes of code per line where DB directives or DW directives are coded. A statement that contains 7 bytes or more of code will be displayed in multiple lines with >>> shown on the Line field for the second line and after.

15) When a source statement has an error or warning, this shows the error contents immediately after that statement. When one line generates many errors, up to the first 5 errors generated will be displayed.

4.13.2 How To Read Cross-Reference Lists

The cross-reference list shows the line numbers of symbols that appear in the program.

An example of a cross-reference list is shown below.

```
RAS66K (MSM66507) Relocatable Assembler, Ver.4.21 C-Ref list.    page:   2

symbol       lines ( #:definition line)
------------ ------------------------------------------------------------------
APSW ....... (SFR) 43
ASSP ....... (SFR) 42
BITSYM ..... 26#
CODESYM .... 25#
COMSYM ..... 27#
EXTSYM ..... 22#
MACROSYM ... 29#
NUMSYM ..... 23#
PUBSYM ..... 24# 30
SEG000 ..... 11# 18
SEG001 ..... 12# 18
SEG010 ..... 13# 19
SEG011 ..... 14# 19 32
SEG020 ..... 15# 20 35
SEG021 ..... 16# 20 38
UNDEF ...... 30
```

As shown above, the cross-reference list consists of two fields. Each field is explained below.

The symbol field displays the names of user symbols.

The lines field shows the line numbers where the symbols appeared. Line numbers with # appended indicate the symbol's definition line. If the symbol was defined with SET directives, then the last definition line will be appended with #. (SFR) will be prefixed before the line numbers for SFR address symbols defined in the DCL file.

4.13.3 How To Read Symbol Lists

The symbol list gives detailed information about symbols that appear in the program. It is configured from symbol information, segment information, and segment group information.

4.13.3.1 Symbol Information

The symbol information display shows detailed information about all symbols defined in the program and SFR symbols that are referred at least once in the program.

Below is an example of symbol information.

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

symbol       type usgtyp physeg     value    ID
------------ ---- ------ ------ ----------- ----
APSW ....... sfr  DATA   #00          4H     0
ASSP ....... sfr  DATA   #00          0H     0
BITSYM ..... loc  BIT    #00        200H.0   0
CODESYM .... loc  CODE   #00       1000H     0
COMSYM ..... com  DATA   ANY          0H     1
EXTSYM ..... ext  CODE   ANY          0H     1
MACROSYM ... mcr  Macro body
NUMSYM ..... loc  NUMBER ---    10000000H    0
PUBSYM ..... pub  DATA   #00       1000H     0
SEG000 ..... seg  CODE   ANY          0H     1
SEG001 ..... seg  CODE   ANY          0H     2
SEG010 ..... seg  CODE   #00          0H     3
SEG011 ..... seg  CODE   #00          0H     4
SEG020 ..... seg  DATA   COMMON       0H     5
SEG021 ..... seg  BIT    ANY          0H.0   6
UNDEF ...... ***
```

Each of these fields are explained next.

The symbol field shows the names of user symbols.

The type field displays the symbol types as follows.

Display	Description
***	Undefined symbol
sfr	SFR symbol
loc	Local symbol
pub	Public symbol
seg	Segment symbol
com	Communal symbol
ext	External symbol
mcr	Macro symbol (the macro body string will be shown after “mcr”)

The usgtyp field shows the usage types as follows.

Display	Description
NUMBER	Usage type NUMBER
CODE	Usage type CODE
DATA	Usage type DATA
EDATA	Usage type EDATA
CBIT	Usage type CBIT
BIT	Usage type BIT
EBIT	Usage type EBIT

The physeg field shows the physical segment attributes as follows.

Display	Description
—-	Numeric value
#XX	Physical segment address *XX* (hexadecimal)
COMMON	Physical segment attribute COMMON
ANY	Undetermined physical segment address

The value field shows the symbol’s value in hexadecimal.

For segment symbols, communal symbols, and external symbols, the ID field shows the order of definition . For simple relocatable symbols, it shows the ID of the segment in which each symbol resides. Otherwise it shows nothing.

4.13.3.2 Segment Information

The segment information display shows detailed information about segment symbols defined in the program.

Below is an example of segment information.

```
----- segment information -----

S-ID symbol       segtyp physeg  size  bound G-ID reltype
---- ------------ ------ ------ ------ ----- ---- ----------------------------
  1 SEG000 ..... CODE   ANY       0H PAGE    1
  2 SEG001 ..... CODE   ANY       0H OCT     1
  3 SEG010 ..... CODE   #00       0H WORD    2
  4 SEG011 ..... CODE   #00      10H UNIT    2
  5 SEG020 ..... DATA   COMMON   100H UNIT    3
  6 SEG021 ..... BIT    ANY     200H UNIT    3 SBA
```

Each of these fields are explained next.

The S-ID field shows the definition order of the segment symbols.

The symbol field the name of each segment symbol.

The segtyp field shows the segment type as follows.

Display	Description
CODE	Segment type CODE
DATA	Segment type DATA
EDATA	Segment type EDATA
BIT	Segment type BIT
EBIT	Segment type EBIT

The physeg field show the physical segment attribute.

The size field shows the segment size in hexadecimal. If the units of size is bits, then the display will be bits. Otherwise, the display will be bytes.

The bound field shows the boundary value attribute. Boundary value attributes are as follows.

Display	Description
UNIT	1-bit boundary for BIT and EBIT segments. 1-byte boundary for all other segments.
WORD	2-byte boundary.
OCT	8-byte boundary.
PAGE	256-byte boundary.
integer constant (hexadecimal)	If the integer constant value is n: n-bit boundary for BIT and EBIT segments. n-byte boundary for all other segments.

The G-ID field shows the group ID of the segment group containing the segment. If no GROUP directive was defined, then nothing will be displayed in this field.

The reltyp field shows the specified special area attribute. If no special area attribute was specified, then nothing will be displayed in this field.

4.13.3.3 Segment Group Information

The segment group information display shows the relocatable segments residing in each segment group.

Below is an example of segment group information.

```
----- segment group information -----

G-ID  memory  physeg group-members-list
---- -------- ------ -----------------------------------------------------------
  1 CODE     ANY    SEG000 SEG001
  2 CODE     #00    SEG010 SEG011
  3 DATA/BIT ANY     SEG020 SEG021
```

Each of these fields are explained next.

The G-ID field shows the definition order of the groups in decimal.

The memory field shows the memory space that the group is allocated to. One of the following will be displayed.

Display	Description
CODE	Group is allocated to program memory space.
DATA/BIT	Group is allocated to data memory space.

The physeg field shows the physical segment attribute.

The group-members-list field lists the names of the relocatable segments that reside in each group.

4.13.4 How To Read Termination Messages

Display of termination messages differs depending on the CPU core type. Below are some examples.

■ Example1 ■

```
Target       : MSM66201 (nX-8/200)

Errors   : 0
Warnings : 0 (/Wrpeaus)
Lines    : 62
```

This is the termination message for CPU cores nX-8/100, 200, and 400.

The microcontroller device type is shown after “Target.” The total number of errors is shown after “Errors.” The total number of warnings is shown after “Warnings.” The number of lines process is shown after “Lines.”

■ Example2 ■

```
Target       : MSM66301 (nX-8/300)
Common Top   :   FFH

Errors   : 0
Warnings : 0 (/Wrpeaus)
Lines    : 62
```

This is the termination message for CPU core nX-8/300.

In addition to the contents of the termination message of Example 1, the ending address of the COMMON area is displayed after the microcontroller device type.

■ Example3 ■

```
Target       : MSM66507 (nX-8/500)
Memory Model : LARGE ................................... 1)
Common Top   :  3FFH ................................... 2)
ROM WINDOW    : 3000H to 7FFFH .............. 3)
Internal RAM :  200H to  7FFH .............. 4)
EEPROM       : 4000H to 5FFFH .............. 5)
DUAL-PORT    : 6000H to 7FFFH .............. 6)

Errors   : 0
Warnings : 0 (/Wrpeaus)
Lines    : 62
```

This is the termination message for CPU core nX-8/500.

1) This shows the memory model.

2) This shows the ending address of the COMMON area.

3) This shows the range of the ROM window area specified with the WINDOW directive. It will display “None” if on ROM window area was specified.

4) This shows the range of the internal RAM area.

5) If an EEPROM area is defined in the DCL file, then this shows its range. If not defined, then this line will not be displayed.

6) If a dual port RAM area is defined in the DCL file, then this shows its range. If not defined, then this line will not be displayed.

4.14 EXTRN Declaration Files

This section explains the use of EXTRN declaration files generated by RAS66K

4.14.1 Purpose Of EXTRN Declaration Files

An EXTRN declaration file's contents are external declarations corresponding to the public declarations in the program. An EXTRN declaration file is generated when the /X option is specified.

When a program is divided into modules, the module that defines a symbol needs to declare it public, and modules that refer that symbol need to declare it external. Usually the programmer makes these declarations, but when the number of symbols or modules is large, it becomes easy to make mistakes in the management of symbol declarations.

The EXTRN declaration file generation function is provided as a way to solve this problem.

4.14.2 Use Of EXTRN Declaration Files

As a simple example, suppose subroutines SUB00 and SUB01 and RAM addresses FIX_TBL and SBA_BIT_TBL are defined in module F001.ASM and referred in modules F002.ASM and F003.ASM.

F001.ASM

```
TYPE (M66507)

   PUBLIC  SUB00 SUB01 FIX_TBL SBA_BIT_TBL
    CSEG    AT 1000H
SUB00:
    ;sub routine SUB00
    RT
SUB01:
    ;sub routine SUB01
    RT

    DSEG    AT 280H
FIX_TBL:
    DS     10H

    BSEG    AT 4C0H.0
SBA_BIT_TBL:
    DBIT    8*8
```

Assemble this file as follows.

```
RAS66K F001 /X
```

This will automatically generate the EXTRN declaration file F001.EXT.

F001.EXT

```
;; External symbol declaration file.

    EXTRN   BIT SBA : SBA_BIT_TBL
    EXTRN   DATA FIX        : FIX_TBL
    EXTRN   CODE    : SUB00
    EXTRN   CODE    : SUB01
;; End of listing.
```

Then specify the following at the start of the programs in F002.ASM and F003.ASM that refer these subroutines and symbols.

```
    INCLUDE (F001.EXT)
```

By coding this one line in the program, all public symbols defined in F001.ASM can be referred.

Thus, use of EXTRN declaration files releases the programmer from having to manage external symbols and makes programs easier to read and maintain. Also, as can be seen with FIX_TBL and SBA_BIT_TBL of F001.EXT in the example, the FIX, SBA, and SBAFIX attributes are automatically added, so RAS66K can perform correct address optimization.

4.15 Error Messages

RAS66K reports errors in assembly processing. The errors are reported in the following ways.

- Error messages are output to the screen or an error file.
- Error numbers are output to the print file.

RAS66K has the following types of errors.

- Fatal errors
- Assembler errors
- Warnings
- Internal processing errors

Fatal errors are errors so severe that RAS66K cannot continue processing. When a fatal error occurs, RAS66K suspends assembler processing.

Assembler errors are errors that occur during interpretation of the source file. RAS66K will continue assembler processing and generate a print file and object file even if assembler errors occur.

Warnings indicate that there may be problems in the program. RAS66K will continue assembler processing and generate a print file and object file even if warnings occur.

Internal processing errors occur when RAS66K detects an error in its internal processing. When an internal processing error occurs, RAS66K stops assembler processing.

These error messages are normally displayed to the screen. Use the DOS redirection function to output error messages to a file. To output only assembler errors and warnings to a file, use the /E option or ERR directive.

4.15.1 Format Of Error Messages

The format of error messages output to the screen or error file is as shown below.

■ Syntax ■

```
filename(line1) :line2 :type number :message
```

The *filename* will be the name of the file that occurred the error. The *line1* will be the line number in the source file where the error was occurred. The *line2* will be the value of the Line field in the print file where the error is shown to have occurred.

The type is the type of error. It will be one of the following.

type	**Error type**
Error	Indicates an assembler error.
Warning	Indicates a warning.

The number is the error number, and the message is the error message that is displayed. Section 4.15.2, "List Of Error Messages," gives a list of error numbers and error messages.

4.15.2 List Of Error Messages

Below is a list of the error messages displayed by RAS66K. The error message numbers are shown to the left of the error messages. A description is given below each error message.

4.15.2.1 Fatal Error Messages

F00 **insufficient memory**
There is not enough memory to continue processing. This error can occur when too many symbols are defined in the source program. If some TSR programs are resident in memory, then remove them. Also, if the /V or /R option (or REF directive) are specified, try assembling without them. If this error still occurs, then you need to split up the program or reduce the number of symbols.

F01 **file not found: *file_name***
The source file, include file, or DCL file indicated by *file_name* could not be found.

F02 **cannot open file: *file_name***
The file indicated by *file_name* cannot be generated. The *file_name* will be the object file, print file, or error file. Check that the specified name does not any invalid characters and that it exists in the specified directory.

F03 **cannot close file: *file_name***
The file cannot be closed. The most likely reason for this error is that disk capacity is insufficient.

F04 **error(s) found in DCL file**
At least one syntax error exists in the DCL file. Because assembler results cannot be guaranteed with this error, processing will terminate after the error message is displayed. As long as you use the original DCL file provided by Oki Electric, this error will not occur.

F05 **file seek error**
A file seek cannot be performed.

F06 **too many INCLUDE nesting levels**
Include file nesting levels exceeds 8.

F07 **line number exceeds 65535**
One source file has more than 65,535 lines (total lines including include files).

F08 **I/O error writing file**
The object file could not be written to.

F09 **TYPE directive missing**
There is no TYPE (device name) specification in the source file, or a different instruction is coded before the TYPE directive.

F10 **unclosed block comment**
A block comment /* . . . */ is not closed.

F11 **illegal reading binary file**

The contents of a C source level debugging information file or ABL file are not correct. If there is a problem with ABL file contents, then after the above error message, the display will continue with the following message.

ABL file : *message*

The *message* shows the contents of the error. When this error occurs, first check the following.

- Are no errors (except for warnings) generated with the first assembly?
- Do the memory model specification, case sensitivity specification, and include path specification completely match between the first assembly and re-assembly?
- Are no fatal errors related to addressing generated during linking?
- Is the /A option specified when linking?

If the error still occurs after all of these are confirmed, then please contact Oki Electric. The message types and their descriptions are shown next.

The messages below starting with "ABL file:" are displayed following the fatal error message "F11 illegal reading binary file" when there is a problem in the ABL file.

ABL file: module information is not found
Information about the program currently being assembled is not included in the ABL file.

ABL file: CORE ID mismatch
The module information defined in the ABL file and target CPU core type of the program currently being assembled do not match.

ABL file: Target machine mismatch
The module information defined in the ABL file and target microcontroller name of the program currently being assembled do not match.

ABL file: Memory Model mismatch
The module information defined in the ABL file and memory model of the program currently being assembled do not match.

ABL file: symbol is not entry
A symbol included in the ABL file is not defined in the program currently being assembled.

ABL file: symbol type mismatch
The symbol type (segment symbol, communal symbol, etc.) of the ABL file and of the program currently being assembled do not match.

ABL file: symbol usage type mismatch
The symbol usage type (CODE, DATA, BIT, etc.) of the ABL file and of the program currently being assembled do not match.

ABL file: local symbol value mismatch
A label or other local symbol stores an incorrect value.

ABL file: file format is illegal
The ABL file structure contains an error.

ABL file: absolute machine code mismatch.(line xxxx)
The machine code generated by re-assembly does not match the machine code fixed up by RL66K.

ABL file: location of absolute code mismatch.(line xxxx)
The machine code generated by re-assembly does not match the address of the machine code fixed up by RL66K.

ABL file: illegal machine code number information.(line xxxx)
The machine code information in the ABL file contains an error.

F12 **reading binary file check sum error**
The checksum of code in a C debugging information file or ABL file is not correct.

F13 **I/O error reading file**
A C debugging information file or ABL file could not be read.

F14 **old DCL file**
A DCL file for an old version of RAS66K is being used.

4.15.2.2 Assembler Error Messages

E00 **bad operand**
An operand is incorrect. For a microcontroller instruction, the addressing specification might be in error, or there might be too many or too few operands. For a directive, the code might not match the format of the directive.

E01 **bad syntax**
This is a basic syntax error found before the instruction was recognized.

E02 **USING LREG required**
Local register address symbols (AER0-AER3,AR0-AR7) must be used after the bank number is specified with a USING LREG directive.

E03 **physical segment address out of range**
The physical segment address exceeds the actual number of usable segments. If this error occurs while within the range of physical segments of the target microcontroller, then the memory model might be SMALL.

E04 **bad character: *c* (*XX*)**
The character *c* (ASCII code *XX*) cannot be used in a program.

E05 **illegal integer constant**
The integer constant or address constant is incorrectly coded.

E06 **illegal escape sequence format**
An incorrect escape sequence format is coded within a character constant or string constant.

E07 **unexpected EOL**
E08 **unexpected EOF**
A character constant (‘c’) or string constant (“...”) is not closed.

E09 **illegal string constant**
The string constant has an invalid coding.

E10 **string constant too long**
The number of characters in the string constant exceeds 256.

E11 **illegal option: *option***
The *option* is not recognized as an option. The specification will be ignored.

E12 **constant required**
An integer constant needs to be specified as an instruction operand or option.

E13 **declaration duplicated**
The same directive or option is specified two or more times, or a segment symbol previously registered in a group is specified in another GROUP directive.

E14 location out of range

The location exceeds the permitted range. This error occurs when the AT address of a segment start specification (CSEG directive, etc.) or start address of an ORG directive exceeds the upper or lower segment restriction, or when the location exceeds the upper restriction after being modified by an instruction or DS, DBIT, GJMP, GCAL, or DB directive.

E15 target microcontroller has no EEPROM

When the target microcontroller has no EEPROM, EDATA segment code is not permitted.

E16 #N not allowed in EDATA segment

The EEPROM area is always located in physical segment 0 or the COMMON area. Therefore a physical segment address cannot be specified for the EDATA segment.

E17 AT address must be NUMBER

If AT and # are specified when an absolute segment is started, then the AT specification will be interpreted as an offset address. In this case the AT specification cannot be an address expression. For example, the following will cause an error.

```
CSEG  AT  2:3000 #4
```

E18 segment type/usage type mismatch

The segment type or usage type required by an instruction does not match the specified type. This error occurs in the following cases.

- The usage type of the start address of a segment start specification (ORG directive, CSEG directive, etc.) does not match the segment type of the current segment.
- The usage type of an operand of a symbol definition directive (CODE directive, etc.) does not match the type of instruction.
- A DS directive is coded in a bit-oriented segment.
- A DBIT directive is coded in a byte-oriented segment.
- A DB or DW directive is coded in a segment other than CODE or EDATA.

E19 undefined symbol: *symbol*

The *symbol* is not defined.

E20 segment symbol required

A segment symbol is required as the right term of the SIZE operator and as the operand of RSEG and GROUP directives.

E21 forward reference not allowed

A forward reference was made where it is not permitted. Many directives do not allow forward references in their operands. Also segment names specified with the RSEG and GROUP directives must be previously defined.

E22 stack segment not allowed

The stack segment $STACK cannot be used.

E23 **symbol redefinition: *symbol***
The *symbol* has already been defined.

E24 **NONE type not allowed**
Address constants cannot be specified as operands of EQU and SET directives.

E25 **segment ID mismatch**
When a relocatable address is specified in the operand of an ORG directive in a relocatable segment, the expression must represent an address in the current segment.

E26 **address not allowed**
If the current segment is a relocatable segment of type ANY, then the address of an ORG directive must be a numeric value (NUMBER).

E27 **physical segment address mismatch**
The physical segment addresses do not match. This error occurs when an ORG directive is specified in a segment with a known physical segment and that physical segment address does not match the current segment. It also occurs when the physical segment addresses of segment symbols specified for a GROUP directive do not match.

E28 **local symbol required: *symbol***
The publicly declared *symbol* must be defined as a local symbol.

E29 **out of range: *message***
The operand value exceeds its permitted range. The *message* shows the specific name of the area.

E30 **illegal boundary**
E31 **illegal relocation type**
The boundary value specification or special area attribute specification of a SEGMENT or COMM directive is incorrect.

E32 **EDATA segment cannot belong to group**
An EDATA segment cannot be placed in a group.

E33 **entry overflow**
The number of segment symbols, communal symbols, external symbols, or segment groups exceeds 65,535.

E34 **string constant required**
The operand of a DATE or TITLE directive must be a string constant.

E35 **absolute expression required**
The operand must be a constant expression. This error occurs when the operand of many directives, the interrupt number of the SWI instruction, or the shift width of a rotate/shift instruction is not a constant expression.

E36 **simple relocatable expression required**
The operand of a symbol definition directive (EQU directive, etc.) or ORG directive must be a constant expression or simple relocatable expression.

E37 **expression unresolved**
An unresolved calculation cannot be evaluated, or the operand of a symbol definition directive (EQU directive, etc.) or ORG directive includes unresolved calculation.

E38 **illegal expression format**
The expression has a basic syntax error. For example, this error will occur if parentheses do not balance.

E39 **invalid relocatable expression**
An unpermitted operation on a relocatable symbol is being performed.

E40 **divide by zero**
A division or modulo operation by 0 is being performed.

E41 **illegal bit offset**
The bit offset that is the right term of a dot operator is not a constant, or the bit offset of bit addressing is specified as a value greater than 7 or as a non-constant value.

E42 **right expression of SEG operator must be address**
The operand of a SEG operator must be specified as an address.

E43 **nX-8/500 only**
The code is recognized only when the CPU core is nX-8/500.

E44 **illegal core name**
The CPU core name in the #CORE statement in the DCL file is incorrect.

E45 **undefined symbol or forward reference: *symbol***
The *symbol* used in the DCL file is an undefined symbol or has not yet been defined.

E46 **mnemonic required**
An instruction mnemonic is required after a #INSTRUCTION statement in the DCL file.

E47 **too many SWI address**
The maximum number of addresses that can be defined with the #SWI statement in the DCL file is 32.

E48 **#ENDCASE does not have a matching #CASE**
An #ENDCASE statement in the DCL file does not have a matching #CASE statement.

E49 **ROM-LARGE model only**
The FJ, FCAL, and FRT instructions can only be used with the MEDIUM or LARGE memory model.

E50 **cannot optimize RAM addressing**
Optimization of RAM addressing cannot be determined. 64K RAM direct addressing (dir N) is provided for nearly all instructions, but it is not supported for some. If an address is specified without an addressing operator for these instructions, or if “fix” and “sfr” optimization is impossible, then no appropriate addressing will exist. This error will occur in such cases.

E51 CODE segment only
Microcontroller instructions, GJMP directives, and GCAL directives can only be coded in CODE segments.

E52 GJMP/GCAL operand must be symbol
The operand of a GJMP or GCAL directive must be a specified as a symbol.

E53 VCAL address must be even
The operand of a VCAL instruction must be an even address.

E54 out of relative jump range
The branch destination address of a relative jump instruction is not in the range -128 to +127.

E55 LABEL or NAME format error
The syntax relationship between the instruction and a symbol or label is incorrect. For example, this error will occur in the following cases.

```
LABEL:    EQU   100H
NAMES     DS    100H
```

E56 invalid CPU instruction
An instruction used in the program was not defined with an #INSTRUCTION statement in the DCL file.

E57 invalid initialization directive
The position of a directive that initializes the assembler (WINDOW, COMMON, MODEL) is not correct. This error occurs when an instruction that cannot be coded before these directives is. For example, an error will occur with the following code.

```
EXTRN   NUMBER : MAXADDRESS    ;Cannot be coded before WINDOW and COMMON
WINDOW  8000H , 9FFFH
COMMON  2
```

E58 illegal SFR word/byte attribute
The format of the word/byte access attribute field of an SFR access attribute definition statement in the DCL file is incorrect.

E59 illegal SFR bit attribute
The format of the bit access attribute field of an SFR access attribute definition statement in the DCL file is incorrect.

E60 out of SFR address range
An SFR address of an SFR access attribute definition statement in the DCL file is not within the range of the SFR area defined by the SFR or XSFR keyword.

E61 misplaced ENDIF directive
There is no conditional assembly start directive (IF, IFDEF, IFNDEF) corresponding to an ENDIF directive.

E62 **misplaced ELSE directive**
There is no conditional assembly start directive (IF, IFDEF, IFNDEF) corresponding to an ELSE directive.

E63 **unexpected end of file in conditional directive**
There is no ENDIF directive corresponding to a conditional assembly start directive (IF, IFDEF, IFNDEF). This error will normally occur at the last line of the program.

E64 **too many conditional directive nesting levels**
The nesting of conditional assembly directives exceeds 15 levels.

E65 **too many macro nesting levels**
The nesting of macros exceeds 8 levels.

E66 **illegal relocation type combine**
An unpermitted combination of special area attribute is specified with a SEGMENT or COMM directive.

4.15.2.3 Warning Messages

There are six types of warnings. Warning checks can be disabled with the /NW option. Also, you can limit warning checks to particular types by specifying character that represent the warning types following /W. The characters that can be specified after /W and their meanings are shown below.

Character	Check
R	Checks relocatable segment definitions.
P	Checks directive coding.
E	Checks expression evaluations.
A	Performs addressing checks.
U	Perform checks based on USING directives.
S	Check SFR access attributes.

For example, to perform R, U, and S warning checks, specify /WRUS.

Warning check messages and their meanings are listed below. The character shown after the warning number represents the warning type.

W00(R) WINDOW is not set
If the ROM window area is not set by a WINDOW instruction, then the special area attribute WINDOW is ignored.

W01(R) stack size must be even
The stack segment size must be an even number. RL66K will reserve a stack area with a size one greater than the specified value.

W02(P) listing directive duplicated
A previously specified listing directive is being set again. This specification will be ignored.

W03(P) CPU type mismatch
The specification has no meaning for the target microcontroller CPU core. For example, this error will occur if the MODEL directive is specified with a CPU core other than nX-8/500.

W04(E) invalid attribute COMMON
If an address with physical attribute COMMON is specified as the right term of a SEG operator or as the operand of a DSREG or TSREG directive, then it has no meaning.

W05(E) address expression required
An address expression is needed. This warning occurs when a numeric expression is specified as the right term of an OFFSET or PAGE operator or as the operand of a USING PAGE directive.

W06(E) NUMBER expression required
A numeric expression is needed. This warning occurs when an address expression is specified as the right term of HIGH, LOW, or MID operator.

W07(P) directive will be supported in the future
The directive is not supported with the current version of RAS66K.

W08(E) segment address mismatch
The segment addresses of the left and right term of an address operator do not match.

W09(E) address attribute not inherited
The expression will be handled as a number. The address attribute will be lost.

W10(E) cannot check physical segment address matching
RAS66K cannot guarantee that the physical segment addresses match.

W11(E) right expression of operator must be NUMBER
The right term of a shift, divide or modulo operation must be numeric.

W12(A) usage type mismatch
The usage type of the addressing is incorrect, but machine code will be generated.

W13(E) left expression of bit operator must be byte address
The left term of a dot operator must be a byte expression.

W14(E) PAGE operator should be used only on byte/bit address
If the right term of a PAGE operator is a NONE expression, then RAS66K will perform the calculation assuming the value is a byte.

W15(E) DATA segment type only
The usage type of the right term of an LREG operator must be DATA.

W16(E) BPOS operator should be used only on bit address
The right term of a BPOS operator must be bit type.

W17(A) address out of segment range
The value of an expression specified for addressing does not actually exist in its address space. For example, this error will occur if a RAM address in the ROM window area is specified.

W18(U) (CHK) using data type mismatch
The DD state of the branch source and destination of the branch instruction do not match.

W19(U) (CHK) using operation type mismatch
The SF state of the branch source and destination of the branch instruction do not match.

W20(U) (USING DATA check) illegal data type instruction
The DD state specified with the USING DATA directive does not match the state required by the instruction.

W21(U) (USING OPRT check) illegal data type instruction
The SF state specified with the USING OPRT directive does not match the state required by the instruction.

W22(U) (USING DSREG check) out of RAM physical segment
The RAM physical segment address specified with the USING DSREG directive does not match the physical segment address of the operand.

W23(U) (USING TSREG check) out of ROM physical segment
The ROM physical segment address specified with the USING TSREG directive does not match the physical segment address of the operand.

W24(U) (USING PAGE check) out of current page
The current page number specified with the USING PAGE directive does not match the page number of the operand.

W25(S) illegal access to SFR
The access to the SFR area is invalid. This warning occurs for writes to an SFR with writes prohibited, and for word accesses to SFR with word accesses prohibited.

W26(S) cannot access to high byte of SFR word
A word access is being made to the high byte of an SFR that can only be accessed as a word.

W27(A) branch to different segment area
The physical segment addresses of the branch source and destination of a NEAR branch instruction do not match.

W28(A) cannot access to high byte
A word access is being made to an odd address in RAM.

W29(A) cannot write to ROM-WINDOW area
A write is being made to the ROM window area.

W30(A) VCAL address must be #0
The operand of a VCAL instruction must be an address in physical segment 0.

W31(E) reference before first definition
A symbol defined with the SET directive was referred before its first definition. It will be set to the last value defined.

4.15.2.4 Internal Processing Error Messages

**** RAS66K Internal Error : Process [*function*]****

This error occurs if RAS66K detects an internal processing error. The *function* is a string that represents the internal processing location. This error will normally not occur, but if it does, then please contact Oki Electric.

Chapter 5

RL66K

This chapter explains how to use the linker RL66K. It also explains the options and environment variables for controlling RL66K operation, and describes the format of the map file generated by RL66K.

5.1 Introduction

The linker RL66K links multiple object files generated by the relocatable assembler RAS66K, creating an absolute object file.

In addition to object files, library files created with LIB66K can also be specified as input to RL66K. When library files are specified, RL66K extracts object modules from the library files and links them. There are three ways to link object modules in a library file.

- Extract and link all object modules.
- Extract and link only the specified object modules.
- Extract and link only object modules for resolving unresolved external references.

In this chapter, object modules are simply called modules.

The absolute object file generated by RL66K includes object code in which all relocatable parts have been resolved. Debugging information can also be output to this file using options.

RL66K generates a map file. The map file is a list of segment allocation states and public symbols. It is used during program debugging to see segment start addresses.

In order to express relative positions within one physical segment, this chapter calls memory toward address 0 low memory, and memory toward address 0FFFFH high memory.

5.2 RL66K Memory Space Management

RL66K defines the following memory spaces using memory information obtained from input modules. It uses these spaces to allocate segments and communal symbols and to resolve values of relocatable symbols.

- Program memory space
- Data memory space
- EEPROM space
- Dual port RAM space

Program memory space and data memory space are defined for all OLMS-66K Series microcontrollers. EEPROM space and dual port RAM space are defined when the CPU core is nX-8/500 and the device contains EEPROM and dual port RAM.

5.2.1 Program Memory Space

Program memory space has up to 256 physical segments. The address range of one physical segment is 0 to 0FFFFH. CODE segments, communal symbols, and quasi-segments are allocated in program memory space. Refer to Section 5.5.4.2, "Quasi-Segments," regarding quasi-segments.

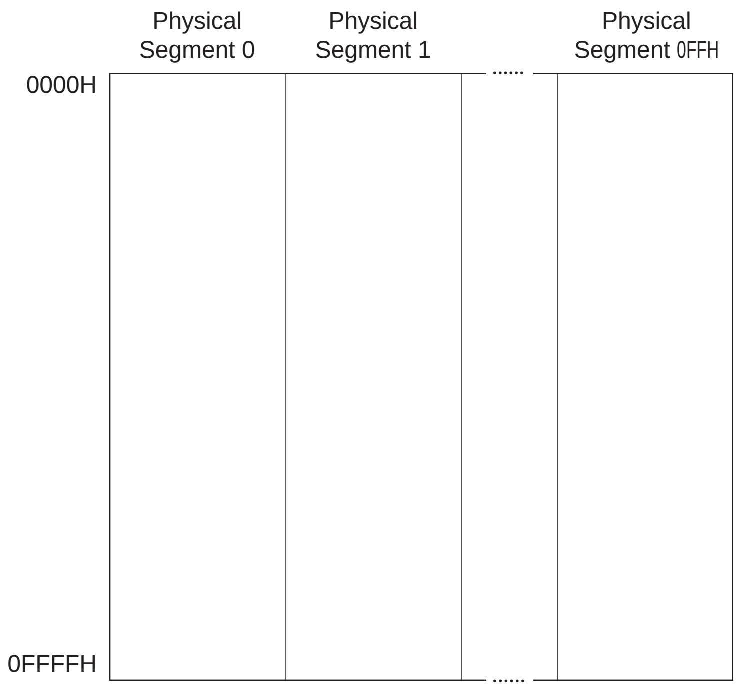


Figure 5-1. Program Memory Space Managed By RL66K

5.2.2 Data Memory Space

Data memory space has up to 256 physical segments. The address range of one physical segment is 0 to 0FFFFH. DATA segments, BIT segments, communal symbols, and quasi-segments are allocated in program memory space.

5.2.3 EEPROM Space

EEPROM space is the address range corresponding to the EEPROM area in data memory space. EDATA segments, EBIT segments, and communal symbols are allocated in EEPROM space.

5.2.4 Dual Port RAM Space

Dual port RAM space is the address range corresponding to the dual port RAM area in data memory space. DATA and BIT segments with the special area attribute DUAL and communal symbols are allocated in dual port RAM space.

Even DATA and BIT segments without the special area attribute DUAL will be allocated in dual port RAM space if they are given addresses in the dual port RAM area by the /DATA or /BIT option.

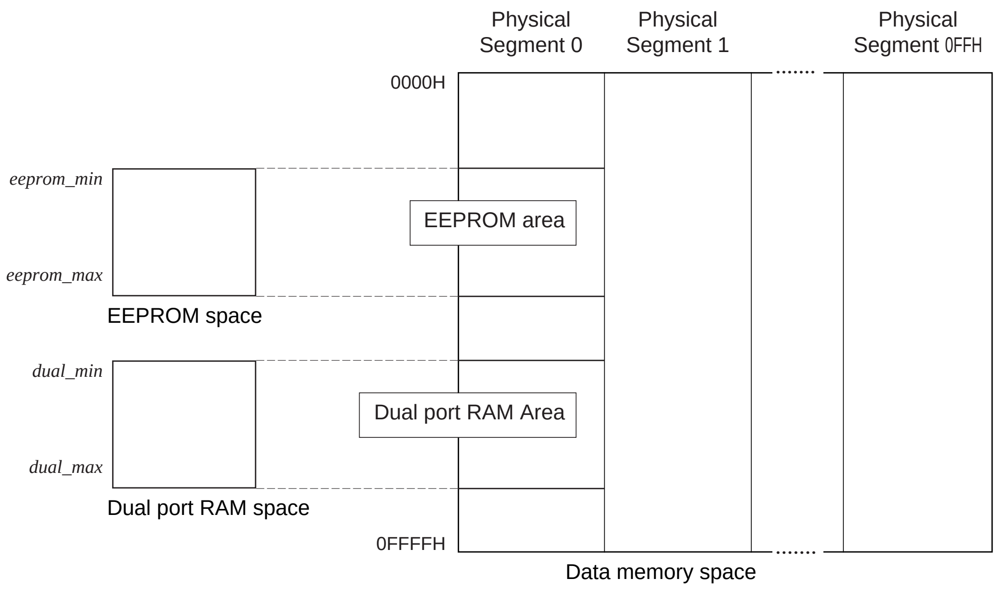


Figure 5-2. Data Memory Space, EEPROM Space, And Dual Port RAM Space Managed By RL66K

5.3 Using RL66K

This section explains the command line format of RL66K and how to start RL66K.

5.3.1 Command Line Format

RL66K command line format is as follows.

```
RL66K object_files [, [absolute_file] [, [map_file] [, [libraries] ] ]
] [;]
```

Each field is used as follows.

- The *object_files* field is used to specify the names of object files and library files to be linked.
- The *absolute_file* field is used to change the default output file name to another name.
- The *map_file* field is used to change the default map file name to another name.
- The *libraries* field is used to specify library files to be used to resolve unresolved external references.

Fields are delimited by commas. Input in fields is specified as needed. To specify nothing in a field, input only the comma that immediately follows the field. If only a return key is input instead of a comma, then RL66K will display a prompt for that field’s input.

RL66K provides several options for changing its default processing. Options can be specified in any field. Refer to Section 5.4, “RL66K Options, “ for details about options.

The semicolon (;) at the end of the RL66K command line indicates the end of the command. When a semicolon is specified, RL66K will not display prompts for input of the remaining fields. In this case RL66K will use the default values for the omitted fields. You can prevent unneeded prompts from being output by using the semicolon.

Unless a path is specified in an input file name, RL66K will assume the current directory in the current drive as the default path. For files not on the default path, you must give a path name in the file specification.

To specify a file name without an extension, append a period (.) immediately after the name. If no period is appended, then that field will be appended with its default extension.

In some fields, only a path name can be specified. In these cases, a backslash (\) must be added immediately after the path name. For example, specify “\USR\APDIR\.” If this is not done, RL66K will recognize the last directory of the path name as the file’s base name, and will append the default extension for that field.

The use of each field is explained next.

5.3.1.1 *object_files* Field

The *object_files* field is used to specify the object files to be linked. At least one file name must be specified. If no extension is specified, then RL66K will assume the default extension of ".OBJ."

When multiple files are specified, they should be delimited by a space or plus sign (+). To specify input in the *object_files* field that extends to the next line, type a plus sign (+) as the last character of the current line, press the return key, and continue the remaining input. However, a single name cannot be split in two. Refer to Example 4 of Section 5.3.1.5, "Command Examples."

Library files can also be specified in the *object_files* field. Only file names with the extension ".LIB" will be handled as library files. The default extension of this field is ".OBJ," so you must specify the extension ".LIB" to specify a library file. If no extension or an extension other than ".LIB" is specified, then RL66K will handle that file as an object file.

When a library file is specified, RL66K will extract and link all object modules in the library file regardless of whether or not they resolve unresolved external references. This is the same as if all object modules in the library file were specified in the *object_files* field. This method allows you to avoid typing many file names each time you invoke RL66K.

You can also link only particular modules in the library file. To do this, specify those module names after the library file name.

library file name (list of module names)

To specify multiple module names, delimit them with spaces in between. RL66K will extract and link only the specified modules from the library. The other modules in the library will not be linked.

■ **Example** ■

```
RL66K MAIN PROJECT.LIB(GETDATA CALC DISPLAY) ;
```

In this example, only the modules GETDATA, CALC, and DISPLAY in the library PROJECT.LIB will be extracted and linked as MAIN.OBJ.

File Searching Method

RL66K searches for object files and library files specified in the *object_files* field in the following places.

- If the file name includes a path specification, then RL66K will search for the file in that directory. If the file is not found, then the search terminates.
- If the file name does not include a path specification, then RL66K will search for the file in the current directory. If the file is not found, then the search terminates.

In either of the above cases, if the file is not found, then RL66K will display an error message and end processing.

Displaying An Explanation Of Command Line Format

If nothing is specified in the *object_files* field and only a return key is input, then RL66K will display a prompt. If only a return key is input to this prompt, then RL66K will display an explanation of command line format and terminate. This is convenient when you have forgotten how to invoke RL66K.

5.3.1.2 *absolute_file* Field

The *absolute_file* field is used to specify the name of the absolute object file that RL66K will output. If no extension is specified, then RL66K will assume the default of ".ABS."

If nothing is specified in the *absolute_file* field, then RL66K will use a default name. The default name will be the base name of the first file in the *object_files* field with the extension ".ABS" appended.

If only a path is specified in the *absolute_file* field, then RL66K will create an absolute object file with the default name in that directory. Unless the path is explicitly specified, RL66K will create the absolute object file in the current directory.

5.3.1.3 *map_file* Field

The *map_file* field is used to specify the name of the map file or to suppress its generation. The map file is a text file that shows the results of linking. Refer to Section 5.6, "Map File," for the format of the map file.

If nothing is specified in the *map_file* field, then RL66K will use a default name. The default name will be the base name of the first file in the *object_files* field with the extension ".M66" appended.

If only a path is specified in the *map_file* field, then RL66K will create a map file with the default name in that directory. Unless the path is explicitly specified, RL66K will create the map file in the current directory.

To suppress generation of a map file, specify "NUL" in this field.

5.2.1.4 *libraries* Field

The *libraries* field can be used to specify library files. When specifying multiple files, delimit the names with spaces or plus signs (+). When the input to the *libraries* field extends to the next line, enter a plus sign (+) as the last character of the current line, press the return key, and continue with the remaining input. However, one name cannot be split over two lines. If only a library's base name is specified with no extension, then RL66K will assume a default extension of ".LIB."

The library files specified in the libraries field are used to resolve unresolved external references. RL66K searches the library files in the order in which they are specified in the libraries field. If the path of a file is explicitly specified, then that directory will be searched. If the path is not specified, then RL66K will search the following directories in order.

- Current directory
- Directory defined in LIB66K environment variable

If there is an unresolved external reference even after using all specified library files, then it will remain unresolved by default. However, if the /CC option has been specified, then RL66K will search the emulation library for C language programs in order to resolve the remaining external references.

A path specification in the *libraries* field can inform RL66K of the directory in which to search for the emulation library. When specifying a path name, a backslash (\) must be appended to the end of the path name. If this is not done, RL66K will recognize the last name of the path as a library file's base name, and will search for that file with an extension ".LIB."

RL66K searches for emulation libraries in the following order.

- Current directory
- Directory specified in the *libraries* field
- All directories defined in the LIB66K environment variable

If RL66K cannot find an emulation library in one of these directories, then it will display a fatal error and terminate processing.

5.3.1.5 Command Examples

Use of the RL66K command line is shown here using examples.

■ Example 1 ■

```
RL66K MAIN CALC DISP,,MAINLIST,USER.LIB
RL66K MAIN+CALC+DISP,,MAINLIST,USER.LIB
```

These two commands indicate the same things to RL66K. RL66K will link MAIN.OBJ, CALC.OBJ, and DISP.OBJ, generating the absolute object file MAIN.ABS. The map file name will be MAINLIST.M66. RL66K will look into USER.LIB to resolve external references.

■ **Example 2** ■

```
RL66K MAIN CALC DISP,,NUL;
```

In this example, RL66K will link MAIN.OBJ, CALC.OBJ, and DISP.OBJ, generating the absolute object file MAIN.ABS. NUL is specified in the *map_file* field, so RL66K will not generate a map file.

■ **Example 3** ■

```
RL66K PROJECT1.LIB;
```

In this example, RL66K will link all modules in the library PROJECT1.LIB, generating the absolute object file PROJECT1.ABS. The map file name will be PROJECT1.M66.

■ **Example 4** ■

```
A>RL66K MAIN GETDATA +
INPUT FILES [.OBJ]: CALC ERRHDL +
INPUT FILES [.OBJ]: DISPLAY USER.LIB ;
```

In this example, the input to the *object_files* field is specified over three lines. In the first line (command line), MAIN.OBJ and GETDATA.OBJ are input. The last character of this line is a plus sign, so RL66K will display a prompt for required input to the *object_files* field. Here CALC.OBJ and ERRHDL.OBJ are input with a plus sign at the end. RL66K will again display a prompt for required input to the *object_files* field. Here DISPLAY.OBJ and USER.LIB are input, followed by a semicolon. Because a semicolon is specified, RL66K recognizes the end of the command, and will not display a prompt for the remaining fields.

5.3.2 Execution

RL66K begins processing if at least one object file is specified. There are three ways to specify the input needed by RL66K.

- Specify all needed input on the command line.
- Specify the input at prompts displayed by RL66K.
- Specify the input through a response file by specifying a response file name at a fixed position on the command line.

These methods can be used in combination. For direct command line specification, refer to Section 5.3.1, “Command Line Format.” This section describes the other two methods.

5.3.2.1 Prompt-Based Input

When some of the fields of the command line are omitted and the command line is not terminated with a semicolon, RL66K displays prompts for the omitted input. RL66K prompts for the input it needs by displaying the following lines one line at a time.

```
   INPUT FILES [.OBJ]:
   OUTPUT FILE [base_name.ABS]:
   MAP FILE [base_name.M66]:
   LIBRARIES [.LIB]:
```

RL66K will not display the next line until the current prompt is answered. Each prompt corresponds to the command line fields explained in Section 5.2.1, "Command Line Format."

Prompt	Command Line Field
`INPUT FILES`	`object_files`
`OUTPUT FILE`	`absolute_file`
`MAP FILE`	`map_file`
`LIBRARIES`	`libraries`

To specify all fields using prompts, input only "RL66K" at the DOS prompt.

Options can be specified anywhere in any field if they are input before a semicolon is input.

RL66K will display the default value for each field in right-angle brackets. If the default value is acceptable, simply input the return key. To change to some other value, type the file name. The *base_name* is the base name of the first file specified in the *object_files* field. To specify the defaults for remaining fields without displaying the prompts, input a semicolon and press the return key.

If a file name is specified without an extension, then RL66K will add the default extension. To specify a file name that has no extension, append a period (.) immediately after that name.

When multiple files or paths are specified in the *object_files* or *libraries* fields, delimit them with spaces or plus signs (+). If the response to *object_files* or *libraries* is long and will extend to the next line, then input a plus sign as the last character of the current line, press the return key, and continue the remaining input. If the same prompt is displayed on the new line, then the response input can be continued. A single file name or path name cannot be split over two lines.

5.3.2.2 Specifying Response File Input

RL66K can be given its inputs using response files. A response file is text file that includes input for the command line or prompts. By using response files, you can easily specify frequently used options or inputs and you can make specifications that exceed the 128-character restriction on DOS command lines.

Response File Usage

Specify a response file name to any prompt or at any position in the command line. Specify a response file name immediately after a "@" symbol. Response files do not have default extensions, so if a response file has an extension, then it must be specified. A path can be specified in the file name.

Response files can be specified in any field (any command line field or any prompt), and can be used as the specification for one field, multiple fields, or all remaining fields. RL66K does not particularly regulate the contents of response files. RL66K reads fields from a response file and assigns them in order to the fields that have not been input. RL66K will ignore field or command line specifications in the response file that come after all four fields have been satisfied or after it has encountered a semicolon.

■ Example 1 ■

In the example below, response file MYOBJ.LNK is specified after the object file MAIN.OBJ.

```
RL66K  MAIN @MYOBJ.LNK, MYLIB.LIB
```

■ Example 2 ■

In the example below, prompts are used to specify the same inputs as in Example 1.

```
A>RL66K MAIN
INPUT FILES [.OBJ]: @MYOBJ.LNK
LIBRARIES [.LIB]: MYLIB.LIB
```

Response File Contents

Input files may be input on separate lines or by delimiting them with commas on the same line. If a plus sign is added to the end of a line, then a field can be extended to the next line. Fields not to be input are expressed by blank lines or by commas.

Options can be input anywhere in any field if the come before a semicolon is specified.

Comments can be coded anywhere in response files. Comments have no effect on RL66K processing. Comments are coded following two consecutive slashes (//). RL66K handles all characters on a line after the slashes as a comment.

■ Example ■

An example of a response file is shown below. Line numbers are shown at the left to aid in explanations, but they would not be coded in an actual response file.

```
1:  // TM MODEL X1
2:  // RELEASE 2.3.1
3:  TMX1 GETDATA CALC+
4:  COMP DISPLAY+ //new module
5:  TABLE         //original date
6:  /S
7:  TMX1LIST
8:  // library
9:  MATH.LIB
```

The first, second, and eighth lines are only comments, so they are ignored. The third and fourth lines end with a plus sign, so they are continued to the following line. Thus the third, fourth, and fifth line correspond to the *object_files* field. The sixth line corresponds to the *absolute_file* field.

Only the /S option is specified; the absolute file specification is omitted. The seventh line corresponds to the *map_file* field. The ninth line corresponds to the libraries field.

Assuming this response file is called TMX1.LNK, invoke RL66K as follows.

```
RL66K @TMX1.LNK
```

The above command specifies the following to RL66K using the response file.

- RL66K links the six object files TMX1.OBJ, GETDATA.OBJ, CALC.OBJ, DISPLAY.OBJ, and TABLE.OBJ, creating an absolute object file called TMX1.ABS.
- RL66K extracts and links modules needed from library file MATH.LIB.
- RL66K generates a map file called TMX1LIST.MAP.
- The /S option is specified, so RL66K outputs a table of public symbols and communal symbols to the map file.

5.3.3 Termination Code

When RL66K terminates operation, it will return one of the following termination codes. The termination code can be used in a MAKE file or batch file.

Table 5-1. Termination Code

Termination Code	Description
0	No errors occurred.
1	Warning errors occurred.
2	Errors occurred.
3	Fatal errors occurred.
4	Command line errors occurred.
5	User input Ctrl+C.

Note that RL66K will not generate an absolute object file if the termination code is 2, 3, or 4.

5.4 RL66K Options

This section explains how to use options for controlling RL66K operation and modifying its output. It also introduces each option's specification and interpretation.

5.4.1 Option Specifications

First, this section describes the rules for using options

5.4.1.1 Syntax

Option syntax is as follows.

```
    /option_name [( argument_list )]
```

All options begin with a slash (/). The option name (*option_name*) follows the slash. Some options also need arguments (*argument_list*). Arguments follow the option name and are enclosed in parentheses. The specifications should be made without delimiting between the slash, option name, and left parenthesis with spaces.

RL66K does not distinguish between upper-case and lower-case letters in option names. For example, the /CODE option can also be specified as /Code or /code.

5.4.1.2 Usage

Options can be specified anywhere in a command line, in a response to a prompt, or in a response file. Options can be specified in multiple locations or can be gathered in one location.

When multiple options are specified contiguously, spaces may or may not be used to delimit between them.

5.4.1.3 Name Arguments

Some options have names given as arguments. RL66K does distinguish between upper-case and lower-case letters in names given as arguments. For example, the arguments "MOUSE," "mouse," and "Mouse" are handled as different names for the /CODE option below.

```
    /CODE(MOUSE mouse Mouse)
```

■ Attention ■

RAS66K provides the /CD option for case sensitivity of symbols. If the /CD option is not specified, RAS66K will convert all symbols defined by the programmer to upper-case letters before outputting them to the object file. Thus, if you will use symbols defined in modules assembled without the /CD option as arguments of RL66K operands, then be sure to specify upper-case letters.

5.4.1.4 Address Arguments

Some options have addresses in memory space given as arguments. Address specifications are coded as a physical segment address and offset address separated by a semicolon (:).

```
physical_seg : offset
```

Specify a physical segment address 0 to 0FFH for *physical_seg*, and an offset address 0 to 0FFFFH for *offset*. These may be coded as either decimal or hexadecimal numbers. If the physical segment address and colon are omitted, then RL66K will assume that the physical segment address is 0.

Decimal numbers use the digits 0 to 9. Hexadecimal numbers use digits 0 to 9 and letters A to F (or a to f). Append an “H” or “h” to hexadecimal addresses. For example, the hexadecimal number 1234 is expressed as 1234H or 1234h. If the first character of the number is a letter A to F (or a to f), then specify the digit 0 immediately before that letter. For example, the hexadecimal number C800H starts with the letter C, so prefix it with 0 and specify it as 0C800H.

5.4.2 List Of Options

Table 5-2 lists the options provided by RL66K. The asterisks (*) indicate that an option's functions are specified by default.

Table 5-2. List Of Options

Option		Function
/D		Output debugging information.
/ND	*	Do not output debugging information.
/S		Output a public symbol list.
/NS	*	Do not output a public symbol list.
/CODE		Control allocation of CODE segments.
/DATA		Control allocation of DATA segments.
/BIT		Control allocation of BIT segments.
/EDATA		Control allocation of EDATA segments.
/EBIT		Control allocation of EBIT segments.
/ORDER		Control allocation order of segments with same precedence.
/CM		Set maximum address in program memory space.
/DM		Set maximum address in data memory space.
/CC		Automatically search emulation library.
/SD		Output C source level debugging information.
/NSD	*	Do not output C source level debugging information.
/STACK		Change the stack segment size.
/A		Generate an ABL file.
/NA	*	Do not generate an ABL file.

5.4.3 Option Use

5.4.3.1 Assembly Level Debugging Information Output Control (/D, /ND)

■ Syntax ■

```
/D
/ND
```

■ Description ■

The /D option tells RL66K to output assembly level debugging information for symbolic debugging to the absolute object file. This debugging information includes local symbols, public symbols, communal symbols, and segment names. The /D option will not be effective unless the input object files include assembly level debugging information.

The /ND option suppresses the effect of the /D option. If both /D and /ND are specified in the command line, then the one specified last will be valid.

The default is /ND.

5.4.3.2 Map File Data Output Control (/S, /NS)

■ Syntax ■

```
/S
/NS
```

■ Description ■

The /S option tells RL66K to add a list of all public symbols defined in the object files to the map file. Symbols will be output in alphabetic order.

If NUL is specified in the map_file field, then the map file will not be generated, so specifying this option will have no effect.

The /NS option suppresses the effect of the /S option. If both /S and /NS are specified in the command line, then the one specified last will be valid.

The default is /NS.

5.4.3.3 CODE Segment Allocation Control (/CODE)

■ **Syntax** ■

```
/CODE( segment_name [ - ] [ address ] ... )
```

■ **Description** ■

The /CODE option is used to control allocation of relocatable CODE segments. CODE segments are normally allocated to program memory space in accordance with the precedence shown in Section 5.5.4.3, "Allocation Precedence." Relocatable segments specified with the /CODE option will be given a higher precedence than other relocatable segments and processed first.

The *segment_name* specifies a segment name. Specify the *address* using the syntax explained in Section 5.4.1.4, "Address Arguments." For example, offset address 1234H in physical segment 0 is specified as 1234H, while offset address 1234H in physical segment 8 is specified as 8:1234H. This address must be less than or equal to the maximum address of program memory usable by RL66K. If it is not, then the specification will be invalid.

The following specifications for segments can be made using the /CODE option.

- Allocate a segment in memory higher than the specified offset address in the specified physical segment.
- Allocate a segment to the specified offset address in the specified physical segment.
- Allocate multiple physical segments within the same physical segment.

These are explained in order below. This is followed by an introduction to specifying combinations.

(1) Allocate a segment in memory higher than the specified offset address in the specified physical segment.

Specify the segment name and the reference address. The level 1 precedence of the specified segment will become 3.

The initial value of the reference address is physical segment 0, offset address 0. When an address is encountered in parentheses, the reference address will be changed to it. Also, the reference address will return to the initial value each time a /CODE option is encountered. Here is an example.

■ **Example** ■

```
/CODE(SEG1 SEG2 3:100H SEG3 SEG4) /CODE(SEG5)
```

Because the initial value of the reference address is physical segment 0, offset address 0, the segments SEG1 and SEG2 will be allocated above offset address 0 in physical segment 0. At the point where 3:100H is encountered the reference address will be changed to that value, so the following segments SEG3 and SEG4 will be allocated above offset address 100H in physical segment 3. The segment SEG5 is specified in a new /CODE option. This sets the reference address back to the initial value, so SEG5 will be allocated above offset address 0 in physical segment 0. The table below summarizes this information with the ranges to which each segment can be allocated.

Segment	Allocatable Range
SEG1	0:0000H to 0:FFFFH
SEG2	0:0000H to 0:FFFFH
SEG3	3:0100H to 3:FFFFH
SEG4	3:0100H to 3:FFFFH
SEG5	0:0000H to 0:FFFFH

(2) Allocate a segment to the specified offset address in the specified physical segment.

This method allocates segments to specified addresses. Place a hyphen (-) after the segment name and then specify the address. RL66K will place the start of that segment at the specified address. The level 1 precedence of the specified segment will become 2. Here is an example.

■ Example ■

```
/CODE(SEG1-2:3800H SEG2-1:8000H SEG3-4:2000H)
```

In this example, first SEG1 is allocated to offset 3800H in physical segment 2. Next SEG2 is allocated to offset 8000H in physical segment 1. Finally SEG3 is allocated to offset 2000H in physical segment 4. This example could also be specified by splitting it up as follows.

```
/CODE(SEG1-2:3800H) /CODE(SEG2-1:8000H) /CODE(SEG3-4:2000H)
```

(3) Allocate multiple physical segments within the same physical segment.

This method allocates multiple segments within the same physical segment. The reference segment is specified as explained in (2) above, followed by a list of the other segment names. The level 1 precedence of the reference segment will become 2, and the level 1 precedence of the following segments will be 3.

The specified segments will be allocated in order of appearance after the reference segment toward higher memory. For example, the following can be specified.

■ Example ■

```
/CODE(SEG1-2:3800H SEG2 SEG3)
```

In this example, first SEG1 is allocated to offset 3800H in physical segment 2. Then SEG2 and SEG3 are allocated in the same physical segment as SEG1 above offset address 3800H.

The address specified for the reference segment is valid until another address is specified in the same /CODE option or until a right parenthesis is encountered. Its effect does not carry over to other appearances of the /CODE option. Thus, this specification method differs from the others in that it cannot be divided. If the previous example is divided as follows, then SEG3 will be allocated above offset address 0 in physical segment 0, not in the same physical segment as SEG1.

```
    /CODE(SEG1-2:3800H SEG2) /CODE(SEG3)
```

(4) Specify a combination of all methods.

The methods explained above can be used in combination.

```
    /CODE(1:0F0H SEG1 SEG2-100H SEG3 5:200H SEG4 SEG5-2:300H SEG6 SEG7)
```

In this example, RL66K assigns a level 1 precedence of 2 to SEG2 and SEG5, and a level 1 precedence of 3 to all other segments. The segments will be allocated in the ranges shown below.

Segment	**Allocation Range**
SEG1	1:00F0H to 1:FFFFH
SEG2	0:0100H
SEG3	0:0100H to 0:FFFFH
SEG4	5:0200H to 5:FFFFH
SEG5	2:0300H
SEG6	2:0300H to 2:FFFFH
SEG7	2:0300H to 2:FFFFH

The physical segment is determined when the /CODE option is specified. However, you cannot make a specification that would change a segment that was assigned a physical segment address in the source program. RL66K will display an error message if an option attempts to make such a change. For example, SEG3 in the above example is specified as physical segment 0, but if it had been defined as follows, then an error would occur.

```
    SEG3    SEGMENT CODE #1
```

Addresses specified with the /CODE option may contradict special area attribute and boundary value attributes specified for segments in the source program. In this case, RL66K will ignore the attributes and allocate the segments as specified. Then PL66K will output a warning message. Refer to Section 4.12.8, “Using Relocatable Segments,” regarding special area attributes and boundary value attributes.

RL66K will output a warning message if a specified segment has a group attribute. It will then remove that segment from the segment group. Refer to Section 4.12.9, “Segment Group Definition,” regarding group attributes.

5.4.3.4 DATA Segment Allocation Control (/DATA)

■ Syntax ■

```
   /DATA( segment_name [ - ] [ address ] ... )
```

■ Description ■

The /DATA option is used to control allocation of relocatable DATA segments. DATA segments are normally allocated to DATA memory space in accordance with the precedence shown in Section 5.5.4.3, “Allocation Precedence.” Relocatable segments specified with the /DATA option will be given a higher precedence than other relocatable segments and processed first.

The segment_name specifies a segment name. Specify the address using the syntax explained in Section 5.4.1.4, “Address Arguments.” This address must be less than or equal to the maximum address of DATA memory usable by RL66K. If it is not, then the specification will be invalid.

Use of the /DATA option is the same as for the /CODE option. For details, refer to Section 5.4.3.3, “CODE Segment Allocation Control.”

5.4.3.5 BIT Segment Allocation Control (/BIT)

■ Syntax ■

```
/BIT( segment_name [ - ] [ address ] ... )
```

■ Description ■

The /BIT option is used to control allocation of relocatable BIT segments. BIT segments are normally allocated to data memory space in accordance with the precedence shown in Section 5.5.4.3, “Allocation Precedence.” Relocatable segments have a level 1 precedence of 4 or more by default, but those specified with the /BIT option will be given a level 1 precedence of 2 or 3 and processed before other segments.

The *segment_name* specifies a segment name. The *address* specifies a bit address.

Bit addresses can be specified using one of two methods.

- Directly specify a bit address as offset address and physical segment address, using the syntax explained in Section 5.4.1.4, “Address Arguments.”
- Specify the offset address as a byte address and bit position separated by a period (.), as shown below.

```
data_address.bit_position
```

The *data_address* specifies a data address. The *bit_position* specifies a number 0 to 7 indicating the bit position. These are coded using the syntax explained in Section 5.4.1.4, “Address Arguments.”

For example, bit 5 of offset address 1234H in physical segment 0 is specified as 8:91A5H or 8:1234H.5. This address must be less than or equal to the maximum address of data memory usable by RL66K. If it is not, then the specification will be invalid.

Use of the /BIT option is the same as for the /CODE option. For details, refer to Section 5.4.3.3, “CODE Segment Allocation Control.”

■ Attention ■

The maximum value that can be specified as an address argument is 0FFFFFH. Accordingly, when directly specifying a bit address as the argument of the /BIT option, you can specify only bit addresses up to 0FFFFFH. To specify a bit address beyond this, use a combination of a byte address and bit offset. For example, specify 0FFFE.3 for the bit address 7FFF3H.

5.4.3.6 EDATA Segment Allocation Control (/EDATA)

■ **Syntax** ■

```
   /EDATA( segment_name [ - ] [ address ] ... )
```

■ **Description** ■

The /EDATA option is used to control allocation of relocatable EDATA segments. EDATA segments are normally allocated to EEPROM space in accordance with the precedence shown in Section 5.5.4.3, “Allocation Precedence.” Relocatable segments have a level 1 precedence of 4 or more by default, but those specified with the /EDATA option will be given a level 1 precedence of 2 or 3 and processed before other segments.

The *segment_name* specifies a segment name. Specify the *address* using the syntax explained in Section 5.4.1.4, “Address Arguments.” However, there is no concept of physical segments in EEPROM space, so a physical segment address cannot be specified. This address must be less than or equal to the maximum address of the EEPROM memory usable by RL66K. If it is not, then the specification will be invalid.

Use of the /EDATA option is the same as for the /CODE option. For details, refer to Section 5.4.3.3, “CODE Segment Allocation Control.”

5.4.3.7 EBIT Segment Allocation Control (/EBIT)

■ Syntax ■

```
/EBIT( segment_name [ - ] [ address ] ... )
```

■ Description ■

The /EBIT option is used to control allocation of relocatable EBIT segments. EBIT segments are normally allocated to EEPROM space in accordance with the precedence shown in Section 5.5.4.3, “Allocation Precedence.” Relocatable segments have a level 1 precedence of 4 or more by default, but those specified with the /EBIT option will be given a level 1 precedence of 2 or 3 and processed before other segments.

The *segment_name* specifies a segment name. The *address* specifies a bit address.

Bit addresses can be specified using one of two methods.

- Directly specify a bit address as offset address and physical segment address, using the syntax explained in Section 5.4.1.4, “Address Arguments.”

- Specify the offset address as a byte address and bit position separated by a period (.), as shown below.

  data_address.*bit_position*

The *data_address* specifies a data address. The *bit_position* specifies a number 0 to 7 indicating the bit position. These are coded using the syntax explained in Section 5.4.1.4, “Address Arguments.”

For example, bit 5 of EEPROM address 1234H is specified as 91A5H or 1234H.5. This address must be less than or equal to the maximum address of EEPROM space usable by RL66K. If it is not, then the specification will be invalid.

Use of the /EBIT option is the same as for the /CODE option. For details, refer to Section 5.4.3.3, “CODE Segment Allocation Control.”

■ Attention ■

The maximum value that can be specified as an address argument is 0FFFFH. Accordingly, when directly specifying a bit address as the argument of the /EBIT option, you can specify only bit addresses up to 0FFFFH. To specify a bit address beyond this, use a combination of a byte address and bit offset. For example, specify 0FFFE.3 for the bit address 7FFF3H.

5.4.3.8 Segment Allocation Order Control (/ORDER)

■ Syntax ■

```
   /ORDER( segment_name ... )
```

■ Description ■

The /ORDER option is used to control the order that segments with the same precedence are allocated. Segments are always allocated in an order that accords with their own precedence, but the order for processing of segments with the same priority is optional.

The *segment_name* specifies segment names. The specified segment names will be allocated in their order of appearance. For example, assume the following program.

```
    INP_DAT1  SEGMENT  DATA INPAGE(1)
    INP_DAT2  SEGMENT  DATA WORD INPAGE(1)

              RSEG     INP_DAT1
    TOP1:     DS       3

              RSEG     INP_DAT2
    TOP2:     DS       253
              END
```

In this program, segments INP_DAT1 and INP_DAT2 have the same precedence. If RL66K processes INP_DAT1 first and allocates it on the boundary of page 1, then RL66K will no longer be able to allocate INP_DAT2 in page 1 since it has the word boundary value attribute. In this case, make sure that INP_DAT2 is allocated first by using the /ORDER option.

```
    /ORDER(INP_DAT2 INP_DAT1)
```

If multiple /ORDER options are specified, then each /ORDER option will have no affect on other /ORDER options. For example, assume that SEG1, SEG2, SEG3, and SEG4 have the same precedence, and that the following is specified.

```
    /ORDER(SEG1 SEG2) /ORDER(SEG3 SEG4)
```

The first /ORDER options specifies that RL66K is to process SEG1 and SEG2 in that order, while the second ORDER option specifies that RL66K is to process SEG3 and SEG4 in that order. However, the order in which the two pairs are processed is not specified.

The segments specified with a single /ORDER option must have the same precedence. For example, assume that SEG1 and SEG2 have the same precedence, while SEG3 and SEG4 have the same but higher precedence, and that the following is specified.

```
    /ORDER(SEG1 SEG2 SEG3 SEG4)
```

Based on the actual precedence of the segments and the /ORDER option specification, the segments will be processed in the order SEG3, SEG4, SEG1, SEG2.

5.4.3.9 Program Memory Space Maximum Address Setting (/CM)

■ **Syntax** ■

```
   /CM( address )
```

■ **Description** ■

RL66K allocates communal symbols and segments of usage type CODE to the usable range of program memory space as defined by the DCL file. To make the usable range smaller than the defined range, specify the new range's maximum address using the /CM option. This address must be smaller than the maximum address of the range defined in the DCL file. If it is not, then the /CM option specification will be invalid.

The *address* specifies a physical segment address and offset address using the syntax explained in Section 5.4.1.4, "Address Arguments." For example, if the range of program memory space defined by the DCL file is 0 to 0FH:0FFFFH, then specify the following to change the range to 0 to 3:7FFFH.

```
    /CM(3:7FFFH)
```

5.4.3.10 Data Memory Space Maximum Address Setting (/DM)

■ **Syntax** ■

```
   /DM( address )
```

■ **Description** ■

RL66K allocates communal symbols and segments of usage type DATA or BIT to the usable range of data memory space as defined by the DCL file. To make the usable range smaller than the defined range, specify the new range's maximum address using the /DM option. This address must be smaller than the maximum address of the range defined in the DCL file. If it is not, then the /DM option specification will be invalid.

The *address* specifies a physical segment address and offset address using the syntax explained in Section 5.4.1.4, "Address Arguments." For example, if the range of data memory space defined by the DCL file is 0 to0 FH:0FFFFH, then specify the following to change the range to 0 to 3:7FFFH.

```
    /DM(3:7FFFH)
```

Data memory space exists separately from EEPROM space and dual port RAM space. Even if the /DM option restricts the EEPROM area and dual port areas mapped to data memory space, the EEPROM space and dual port RAM space will not restricted. Therefore the /DM option has no relation on the allocation of EDATA and EBIT segments, as well as DATA and BIT segments with the DUAL special area attribute.

Refer to Section 5.2, "RL66K Memory Space Management," regarding EEPROM space and dual port RAM space.

5.4.3.11 Emulation Library Automatic Search (/CC)

■ Syntax ■

```
/CC
```

■ Description ■

The /CC option tells RL66K to automatically search the emulation library provided for C language programs and to extract and link necessary modules. For details refer to Section 5.3.1.4, “libraries Field.”

5.4.3.12 C Source Level Debugging Information Output Control (/SD, /NSD)

■ Syntax ■

```
/SD
/NSD
```

■ Description ■

The /SD option tells RL66K to output C source level debugging information to the absolute object file. The debugging information includes information about line numbers and variables. Unless the input object files include C source level debugging information, the /SD option will be ineffective.

The /NSD option suppresses the effect of the /SD option. If both /SD and /NSD are specified in the command line, then the one specified last will be valid.

The default is /NSD.

5.4.3.13 Stack Segment Size Change (/STACK)

■ Syntax ■

```
/STACK( size )
```

■ Description ■

The /STACK option is used to change the size of the stack segment. The stack segment is defined by the assembler directive STACKSEG.

The stack is always reserved as an even number of bytes, so specify an even number for size. If an odd number is specified, then RL66K will make it even by adding one to that value.

If no stack segment is defined in the input modules when the /STACK option is specified, then an error will occur.

5.4.3.14 ABL File Generation Control (/A, /NA)

■ Syntax ■

```
/A [(abl_file)]
/NA
```

■ Description ■

The /A option tells RL66K to generate an ABL file. An ABL file is necessary to have RAS66K generate an absolute print file. Refer to Chapter 8, “Absolute Print File Generation,” for details on absolute print files.

Specify the name of the ABL file to be generated in *abl_file*. If omitted, then the ABL file name will be the absolute object file name with the extension “.ABL.”

The /NA option suppresses the effect of the /A option. If both /A and /NA are specified in the command line, then the one specified last will be valid.

The default is /NA.

5.5 Link Processing

To read the object modules from the files specified in the *object_files* field and generate an absolute object file, RL66K goes through the following process.

- RL66K matches corresponding global symbols. If necessary, RL66K searches specified libraries to resolve external symbols.
- RL66K links segments with the same name.
- RL66K links communal symbols with the same name.
- RL66K allocates segments, communal symbols, and quasi-segments in memory.
- RL66K fixes up unresolved operands and outputs the absolute object file.

The items you need to understand link processing are explained below.

5.5.1 Global Symbol Matching

RL66K reads the object modules specified in the *object_files* field in the order they were specified, and resolves external symbols with public symbols or communal symbols that have the same name. If there are still unresolved external symbols after all object modules have been read, then RL66K will search for the library files specified in the libraries field. If it finds the library files, then RL66K will check whether or not there are modules in the libraries that define public symbols with the same name as the unresolved external symbols. If such a module exists, then RL66K extracts it from the library and adds it to the modules to be linked. This process is repeated until all external symbols have been resolved.

Whether an external symbol can be resolved with another module's public symbol or communal symbol depends on their usage types. If their usage types are the same, then the external symbol will be resolved.

When RL66K encounters communal symbols and public symbols with the same name, the name will be handled as a public symbol, not a communal symbol. This is conditional on the symbols having the same segment type and a valid matching of physical segment attributes. Valid physical segment attribute combinations are both symbols having the same physical segment address, both having the COMMON attribute, either one having the ANY attribute. When a public symbol and communal symbol match, the information about the communal symbol will be discarded, so it will not be allocated to memory space.

Every symbol has flag attributes that indicate the state of DD and SF. When RL66K matches symbols it compares those states. If different, RL66K outputs a warning but the symbols will still be correctly resolved.

5.5.2 Segment Linking

When multiple modules a linked, multiple segments with the same name (called partial segments) may appear. RL66K will try to link the partial segments as a single segment. Segment linking is performed by iterations of linking two segments. When RL66K links partial segments, it compares their various attributes to determine if they actually can be linked.

Segments must satisfy the following conditions to be linked.

- Both segments must have the same segment type.
- When one segment has a group attribute, the other must not have a group attribute.
- When one segment has the group attribute ANY, the other must not have a physical segment address.
- The combination of physical segment attributes of the symbol must be valid. Valid physical segment attribute combinations are both symbols having the same physical segment address, both having the COMMON attribute, either one having the ANY attribute.
- The combination of special area attributes of the symbol must be valid.
- The total size of the two segments must not exceed the memory size of the area to which they will be allocated.

The attributes of the segment linked and created by RL66K are as follows.

- The segment type is inherited from the segments before linking.
- The larger boundary value attribute of the segments before linking will be inherited.
- The size will be the total size of the two segments.
- When segments with the same physical segment attribute are linked, the created segment will be given the same attribute and same physical segment address. When one segment is ANY and the other is not, then the created segment will be given the attribute and physical segment address of the other segment.
- The special area attribute will be a combination of the special area attributes of the segments before linking.

As shown in Figure 5-3, partial segments are placed contiguously at the end in order of appearance within the modules. In other words, all partial segments linked into one segment will be located contiguously in memory. Because of this, some partial segments may not be placed on the boundary value specified in the source program. For example, consider a case where two partial segments are both specified to have boundary value 2. The linked segment will also have boundary value 2, so the start of the segment will be placed at an even address. If the size of the first partial segment happens to be an even number of bytes, then the second partial segment will be placed at an even address. However, if the size of the first partial segment happens to be an odd number of bytes, then second partial segment will start at an odd address. Be careful of word based addressing in these segments.

RL66K checks flag attributes at the same time it performs global symbol matching. The link will be completed normally even when warnings are output.

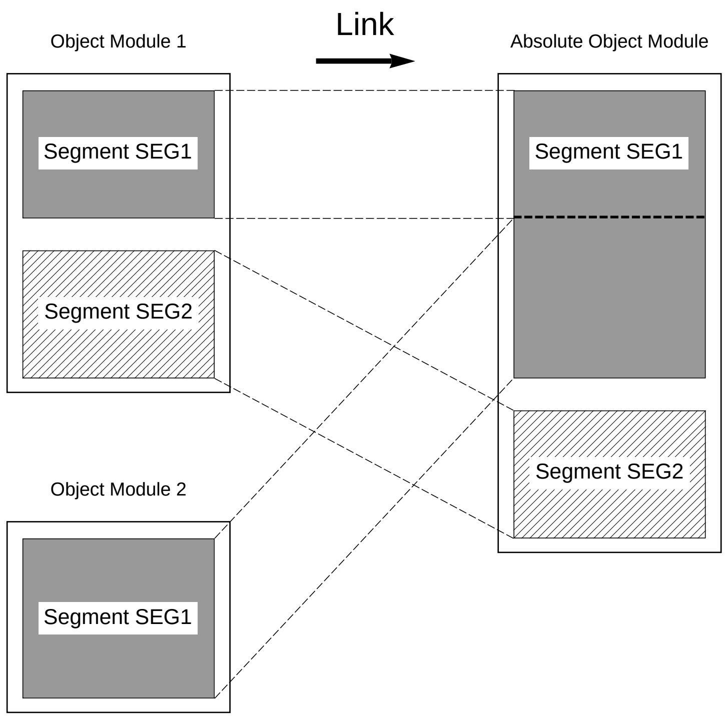


Figure 5-3. Segment Linking

5.5.3 Communal Symbol Linking

Similarly as for segments, RL66K links communal symbols with the same name into a single communal symbol. The linking procedure is the same as for segments. However, partial segments are placed contiguously at the end in order of appearance within the modules, but communal symbols are linked to overlay at their start address, as shown in Figure 5-4. The communal symbol generated will have the same size is the largest of the linked symbols.

Conditions for communal symbol linking are the same as for segment linking.

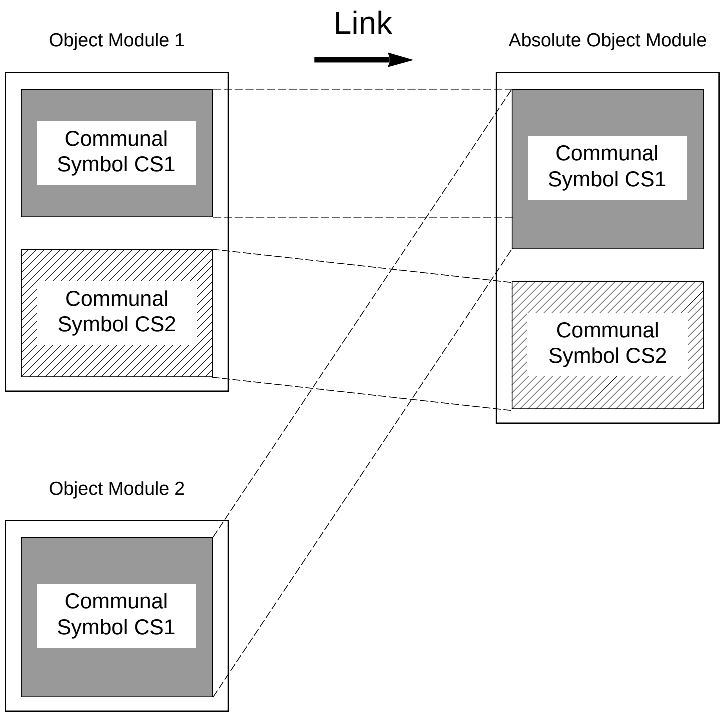


Figure 5-4. Communal Symbol Linking

5.5.4 Segment Allocation

When segment linking and communal symbol linking are complete, RL66K allocates segments, communal symbols, and quasi-segments in memory space. The areas to which RL66K allocates segments and communal symbols are described next in Section 5.5.4.1, “Allocation Spaces And Areas.” Quasi-segments are explained in Section 5.5.4.2, “Quasi-Segments.”

The range of allocatable memory is specified in the DCL file or with /CM and /DM options. For details, refer to Section 5.4.3.9, “Program Memory Maximum Address Setting,” and Section 5.4.3.10, “Data Memory Maximum Address Setting.”

RL66K allocates to memory in order from the segment with the highest precedence. Segments with the same precedence can be processed in any order. When segments and communal symbols have the same precedence, the segments will be allocated first. Precedence is explained later in Section 5.5.4.3, “Allocation Precedence.”

RL66K searches free areas for allocating segments and communal symbols in physical segment 0 from offset 0 up toward high memory. It then searches within physical segment 1 in the same way. It searches this way until the last physical segment. RL66K will perform allocations to the first free area that it finds during its search process. If no allocatable area is found, then RL66K will display an error message and stop link processing.

Segments are allocated on the boundary values specified in the program. The boundary value for communal symbols depends on usage type and size. Communal symbols of type CODE, DATA, and EDATA will be allocated on byte boundaries if their size is 1 byte, or word boundaries if their size is 2 bytes or greater. Communal symbols of type BIT or EBIT will be allocated on bit boundaries without regard to size. Except for boundary value attributes, RL66K handles segments and communal symbols in the same way.

5.5.4.1 Allocation Spaces And Areas

The space and area to which RL66K allocates a segment or communal symbol is determined by the segment type, physical segment attribute, and relocation attribute. Each of these is explained below.

● **Allocation space due to segment type**

The allocation space is determined by the segment type. Segment types and their corresponding allocation spaces are as follows.

Segment Type	**Allocation Space**
CODE	Program memory space
DATA	Data memory space
BIT	Data memory space
EDATA	EEPROM space
EBIT	EEPROM space

● **Allocation area due to physical segment attribute**

CODE, DATA, and BIT segments have physical segment attributes. The correspondence between physical segment attributes and allocation area is shown below.

Physical Segment Attribute	**Allocation Area**
ANY	Any physical segment
COMMON	COMMON area in data memory space
Fixed*	Specified physical segment

* “Fixed” means the physical segment address is specified.

● Allocation area due to special area attribute

When a segment has a special area attribute, its allocation area will be determined by that attribute. The correspondence between special area attributes and allocation areas is as follows.

Special Area Attribute	**Allocation Area**
INACAL	Within the ACAL area of program memory space.
ACAL	Segment start address within the ACAL area of program memory space.
WINDOWALL	Within ROM window area of program memory space. (Refer to Figure 5-5)
WINDOW	Within ROM window area of program memory space. However, overlapping with the EEPROM area, dual port RAM area, and internal RAM area in data memory space will be excluded. (Refer to Figure 5-6)
ZERO	Within zero page area of data memory space.
FIX	Within fixed page area of data memory space.
LREG	Within LREG area of data memory space.
DUAL	Dual port RAM space.
DYNAMIC	Data memory space. Refer to “Dynamic Segment Allocation “ below.
INPAGE	If page value is specified, then within that page. Otherwise, within any page.
SBA	If page value is specified, then within the SBA area of that page. Otherwise, within the SBA area of any page.
SBAFIX	Within the SBA area of the fixed page area.

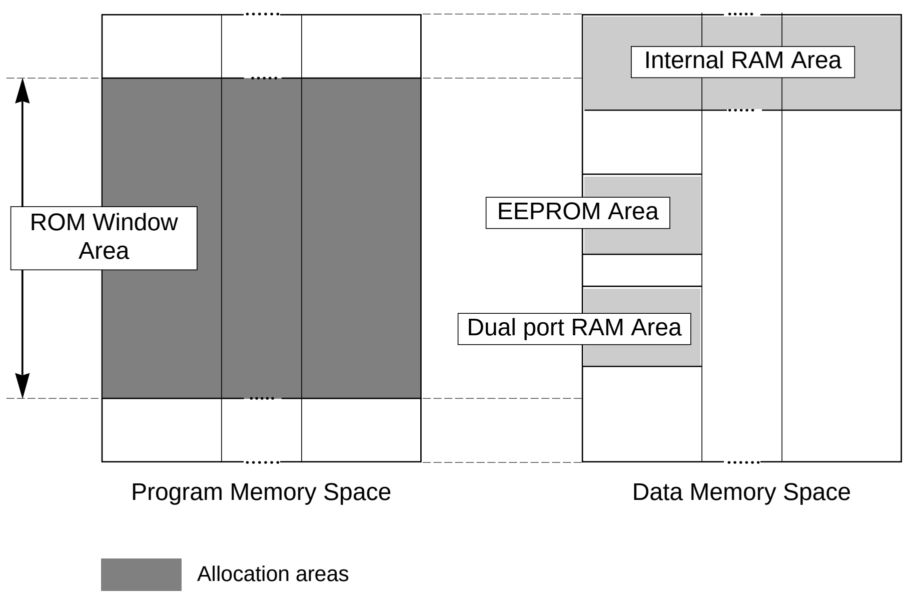


Figure 5-5. Allocation Area For Segments With Special Area Attribute WINDOWALL

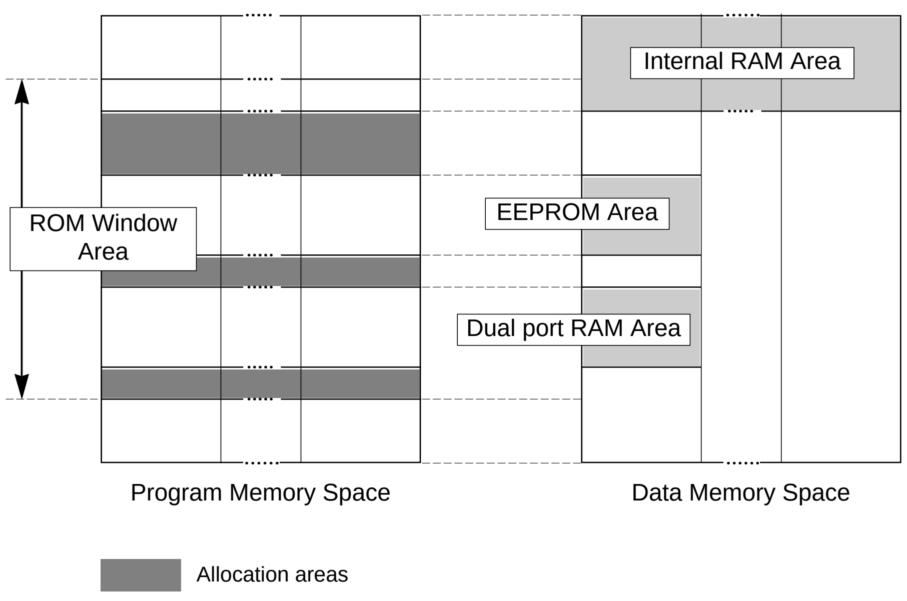


Figure 5-6. Allocation Area For Segments With Special Area Attribute WINDOW

Dynamic Segment Allocation

Segments with the special area attribute DYNAMIC are called dynamic segments. Dynamic segments differ from other segments in that their size is not determined during assembly. After RL66K has allocated all segments, communal symbols, and quasi-segments in memory space, RL66K allocates dynamic segments in the largest remaining free areas in data memory space. The segment names will then be assigned the starting addresses of those areas.

A dynamic segment with the physical segment attribute COMMON will be allocated in the largest remaining free area in the COMMON area.

A dynamic segment with the physical segment attribute ANY will be allocated in the largest remaining free area in any physical segment.

A dynamic segment specified with a physical segment address will be allocated in the largest remaining free area in the specified physical segment.

5.5.4.2 Quasi-Segments

There are several special areas in memory space that must not be allocated to segments defined in a program. RL66K makes its own segments and assigns them to those areas in advance, so it will handle segments and communal symbols defined in the program such that they are not allocated to those areas. The segments created by RL66K itself are called quasi-segments.

Please keep in mind that quasi-segments are not defined by the programmer.

Here are the areas to which quasi-segments are allocated.

- SFR area in data memory space
- XSFR area in data memory space
- Pointing register areas in data memory space used by the programmer
- Local register areas in data memory space used by the programmer
- EEPROM area in data memory space
- Dual port RAM area in data memory space
- ROM window area in data memory space
- Gaps in data memory space
- Gaps in program memory space

Memory space gaps arise only from memory range definitions in the DCL file and from the /CM and /DM options. They are unusable areas.

For example, assume that the range of data memory space is defined by the DCL file as follows.

```
#RAM   0H, 0FFFFH, 1
       DATA  0:0H,     0:7FFFH
       DATA  0:0A000H, 0:FFFFH
```

In this definition, the area from offset address 8000H to 9FFFH in physical segment address 0 of data memory space will be unusable. A quasi-segment will be allocated in that area.

Now assume that the following /DM option is specified with the above definition.

```
    /DM(3FFFH)
```

The area from offset address 4000H to 0FFFFH in physical segment address 0 of data memory space will become unusable. A quasi-segment will be allocated in that area.

5.5.4.3 Allocation Precedence

RL66K assigns the highest precedence to segments with absolute addresses. It then considers option specifications, physical segment attributes, and group attributes to assign precedence to segments in order of how strict their conditions are. This is called level 1 precedence. All segments with the same level 1 precedence are then assigned a further precedence based on their relocation attributes. This is called level 2 precedence. The final precedence is obtained from the combination of level 1 and level 2 precedence, as shown in the table below.

	Level 1 Precedence	**Level 2 Precedence**
High	1	1
↑	1	2
	1	3
	⋮	⋮
	2	1
	2	2
	2	3
↓	⋮	⋮
Low		

The following tables show the level 1 and level 2 precedence for each space.

● **Program memory space**

Table 5-3. Level 1 Precedence Of Segments Allocated In Program Memory Space

Precedence	Segment Conditions
1	Absolute segment defined by CSEG directive or quasi-segment .
2	Segment in /CODE option specified with address.
3	Segment in /CODE option specified without address.
4	Segment has group attribute of physical segment address specification.
5	Segment has physical segment address.
6	Segment has group attribute ANY.
7	Other segments.

Table 5-4. Level 2 Precedence Of Segments Allocated In Program Memory Space

Precedence	Segment Conditions
1	Segment has INACAL attribute.
2	Segment has ACAL attribute*.
3	Segment has WINDOW attribute and SBA attribute with page value.
4	Segment has WINDOW attribute and INPAGE attribute with page value.
5	Segment has WINDOWALL attribute and SBA attribute with page value.
6	Segment has WINDOWALL attribute and INPAGE attribute with page value.
7	Segment has WINDOW and SBA attribute.
8	Segment has WINDOW and INPAGE attribute.
9	Segment has WINDOWALL and SBA attribute.
10	Segment has WINDOWALL and INPAGE attribute.
11	Segment has WINDOW attribute, but not SBA or INPAGE attribute.
12	Segment has WINDOWALL attribute, but not SBA or INPAGE attribute.
13	Other segments.

* Of segments with the ACAL attribute, those with smaller size have higher precedence.

● **Data memory space**

Table 5-5. Level 1 Precedence Of Segments Allocated In Data Memory Space

Precedence	Segment Conditions
1	Absolute segment defined by DSEG or BSEG directive or quasi-segment.
2	Segment in /DATA or /BIT option specified with address.
3	Segment in /DATA or /BIT option specified without address.
4	Segment has COMMON attribute.
5	Segment has group attribute of physical segment address specification.
6	Segment has physical segment address.
7	Segment has group attribute ANY.
8	Segment does not have physical segment address.
9	Stack segment.
10	Dynamic segment that has COMMON attribute.
11	Dynamic segment that has physical segment address.
12	Dynamic segment that does not have physical segment address.

Table 5-6. Level 2 Precedence Of Segments Allocated In Data Memory Space

Precedence	Segment Conditions
1	Segment has LREG attribute and SBA attribute with page value.
2	Segment has LREG attribute and INPAGE attribute with page value.
3	Segment has FIX and SBA attribute.
4	Segment has FIX attribute.
5	Segment has SBA attribute with page value.
6	Segment has INPAGE attribute with page value.
7	Segment has LREG attribute and SBA attribute.
8	Segment has LREG attribute and INPAGE attribute.
9	Segment has LREG attribute.
10	Segment has SBA attribute.
11	Segment has INPAGE attribute.
12	Other segments.

● **EEPROM space**

Table 5-7. Level 1 Precedence Of Segments Allocated In EEPROM Space

Precedence	Segment Conditions
1	Absolute segment defined by ESEG or EBIT directive.
2	Segment in /EDATA or /EBIT option specified with address.
3	Segment in /EDATA or /EBIT option specified without address.
4	Other segments.

Table 5-8. Level 2 Precedence Of Segments Allocated In EEPROM Space

Precedence	Segment Conditions
1	Segment has SBA attribute with page value.
2	Segment has INPAGE attribute with page value.
3	Segment has SBA attribute.
4	Segment has INPAGE attribute.
5	Other segments.

● **Dual Port RAM space**

Table 5-9. Level 1 Precedence Of Segments Allocated In Dual Port RAM Space

Precedence	Segment Conditions
1	Absolute segment defined by DSEG or BSEG directive.
2	Segment in /DATA or /BIT option specified with address.
3	Segment in /DATA or /BIT option specified without address.
4	Other segments.

Table 5-10. Level 2 Precedence Of Segments Allocated In Dual Port RAM Space

Precedence	Segment Conditions
1	Segment has SBA attribute with page value.
2	Segment has INPAGE attribute with page value.
3	Segment has SBA attribute.
4	Segment has INPAGE attribute.
5	Other segments.

5.5.5 Segment Groups

Multiple segments can be specified as a single group using the GROUP directive in a source program. This type of group is called a segment group, or simply group. RL66K will try to allocate all segments residing in the same group to be within the same physical segment. Groups are handled differently depending on whether or not they had physical segment addresses specified in the source program.

When a group had a physical segment address specified in the source program, RL66K will allocate the group's segments within that physical segment.

When a group did not have a physical segment address specified in the source program, RL66K will search for a physical segment in which it can allocate all segments of the group. If it finds such a physical segment, it will allocate the group's segments to it.

5.5.6 Reserving The Stack Area

5.5.6.1 Stack Segment ($STACK)

When the assembler directive STACKSEG is coded in the source program, it defines the stack segment with the segment name $STACK. RL66K allocates the stack segment in physical segment 0 of data memory space. If the free area of physical segment 0 is not enough to hold a stack segment of the specified size, then RL66K will reduce the stack segment's size until it can be allocated. This will also cause a warning.

To assign an absolute address to the stack segment, specify it with the /DATA option as for other segments. For example, to assign absolute address 8000H, specify the following.

```
    /DATA($STACK-8000H)
```

Because the stack segment has physical segment address 0, a physical segment address other than 0 cannot be specified. For example, the following will cause an error.

```
    /DATA($STACK-1:8000H)
```

5.5.6.2 Stack Symbol (_$$SSP)

The stack symbol _$$SSP is a special symbol that provides the initial setting of the system stack pointer SSP. When _$$SSP remains as an unresolved external symbol, RL66K refers the stack segment and sets the symbol to the stack's starting address. If the stack segment is not defined in this circumstance, then an error will occur.

The value of the starting address of the stack set by RL66K is one less than the ending address of the stack segment.

In order to use the stack segment as a stack, initialize the system stack pointer using the stack symbol.

```
    EXTRN    DATA: _$$SSP
    MOV      SSP, #_$$SSP
```

The programmer may also defined _$$SSP, as show below. This means that _$$SSP will not remain as an unresolved symbol, so RL66K will not handle it as a special symbol.

```
_$$SSP   DATA   8000H
MOV      SSP, #_$$SSP
```

5.5.7 Fix-Up Processing

When absolute values are assigned to all symbols, RL66K resolves (fixes up) operands that were left unresolved during assembly.

Normally not all parts of a program are given as absolute values when the program is assembled. Operands that have relocatable symbols cannot be resolved during assembly and are left unresolved. The assembler temporarily assigns 0 to these parts. Then it outputs information for fixing up these operands to the object file. RL66K fixes up the operands based on this information.

The fix-up process for an operand is performed as follows.

(1) RL66K calculates the absolute value from the fix-up information.

(2) RL66K performs checks on that value as necessary. (For details on checks, refer to Section 4.10, “Check Functions.”)

(3) When all checks have been complete, RL66K replaces the portion temporarily assigned 0 by RAS66K with the new value.

The above process corresponds to a single fix-up. RL66K repeats this process until all fix-ups are complete.

5.6 Map File

This section explains how to read the map file generated by RL66K.

The first part of the map file is shown below. It shows details of options that were given when RL66K was invoked.

```
    RL66K Object Linker,Ver.4.23 Linkage Information
(1) [Wed Jun 2 18:34:04 1993]

                          -----------------
                           Control Synopsis
                          -----------------

(2) I/O controls:    D NSD  S  A

(3) Locating controls:

         Type     Address             Name
         ----     -------             ----
         DATA     After 00:0000       DSEG02
         DATA     At    00:8000       DSEG01

(4) Other controls: CC STACK( 0002H(2) ) CM(0:7FFFH)
```

(1) A header is added to the top of the map file. It shows the RL66K version and the date that the map file was created.

(2) This line shows which of the following options were given when RL66K was invoked: /D, /ND, /SD, /NSD, /S, /NS, /A, /NA.

(3) This section shows details of /CODE, /DATA, /BIT, /EDATA, and /EBIT options given when RL66K was invoked. In the “Address” field, the *address* of “At *address*” shows the given absolute address. The *address* of “After *address*” shows the address above which the segment was specified to be allocated.

(4) This line shows details of options other that those of (2) and (3) that were given when RL66K was invoked.

The next part of the map file shows information about processed modules.

```
                          -----------------------
                           Object Module Synopsis
                          -----------------------

(5) Module Name           File Name                 Creator
    -----------           ---------                 -------
    MAIN                  MAIN.OBJ                  RAS66K Ver.4.21
    GETDATA               GETDATA.OBJ               RAS66K Ver.4.22
    CALC                  CALC.OBJ                  RAS66K Ver.4.22

(6) Number of Modules: 3

(7) Number of Symbols:
    +--------------------------------------------------------------+
    |        |CODE |DATA |BIT  |EDATA |EBIT |NUMBER|CBIT || total |
    |--------+-----+-----+-----+------+-----+------+-----++-------|
    |SEGMENT |    4|    3|    1|     2|    0|      |     ||     10|
    |--------+-----+-----+-----+------+-----+------+-----++-------|
    |COMMUNAL|    0|    1|    0|     0|    0|      |     ||      1|
    |--------+-----+-----+-----+------+-----+------+-----++-------|
    |PUBLIC  |    4|    1|    0|     0|    0|     0|    0||      5|
    +--------------------------------------------------------------+

(8) Target:  MSM66507 (nX-8/500)
(9) Model:   LARGE
```

(5) For each module processed by RL66K, this section shows its name, the name of the file that stores it, and the name and version number of the assembler that generated it.

(6) This is the number of modules processed.

(7) For each segment type, this section shows the number of segments, communal symbols, and public symbols in all modules processed. Linked segments or linked communal symbols are counted as one.

(8) This line shows the name of the target microcontroller and, in parentheses, the name of the CPU core.

(9) This line shows the memory model selected by the program when the CPU core is nX-8/500. One of SMALL, MEDIUM, COMPACT, or LARGE will be shown.

The next part of the map file shows the allocation states of segments and communal symbols for each address space.

```
                                  ------------------
                                   Segment Synopsis
                                  ------------------

(10) Link Map - ROM Area ( ROMWINDOW: 1000 - 1FFF )

           Type     Start      Stop       Size          Name
          ----------------------------------------------------
           S CODE   00:0000    00:0015    0016(22)      (absolute)
           S CODE   00:0018    00:0023    000C(12)      CSEG01
     >GAP<                                0FDC(4060)

           --- ROM WINDOW Start: 1000 ---
           S CODE   00:1000    00:100B    000C(12)      CSEG02
           S CODE   00:100C    00:1017    000C(12)      CSEG03
           --- ROM WINDOW Stop: 1FFF ---

(11) Total size = 0003C(60)


(12) Link Map - RAM Area ( COMMON MAX: 03FF )

           Type     Start      Stop       Size          Name
          ----------------------------------------------------
           Q SFR       0000       00FF    0100(256)     (SFR)
           S BIT       0100.0     0107.7  0008.0(8.0)   (absolute)
     *OVL* S DATA      0100    00:04FF    0400(1024)    (absolute)

           --- The above are in COMMON area ---

           S DATA   00:0500    00:05FF    0100(256)     DSEG00
           S DATA   00:0600    00:09FF    0400(1024)    $STACK
     >GAP<                                3600(13824)
           Q EEPROM 00:4000    00:5FFF    2000(8192)    (EEPROM)
           Q DUAL   00:6000    00:7FFF    2000(8192)    (DUAL)
      ---
           C DATA   01:0400    01:0401    0002(2)       DCOM01

     Total size = 090A(2314)


(13) Link Map - EEPROM Area: 4000 - 5FFF

           Type     Start      Stop       Size          Name
          ----------------------------------------------------
           S EDATA  4000       403F       0040(64)      (absolute)
           S EDATA  4040       407F       0040(64)      ESEG01

     Total size = 00080.0(128.0)


(14) Link Map - DUAL_PORT_RAM Area: 6000 - 7FFF

           Type     Start      Stop       Size          Name
          ----------------------------------------------------
           S DATA   6000       603F       0040(64)      DSEG02
           S DATA   6040       607F       0040(64)      DSEG03

     Total size = 00080.0(128.0)
```

(10) This section shows the allocations to program memory space. When a ROM window has been defined, the range of the ROM window area will be shown first.

```
    ( ROMWINDOW: 1000 - 1FFF )
```

When a ROM window has not been defined, the following message will be shown.

```
    ( ROMWINDOW: Not exist )
```

If a ROM window has been defined, then the following messages, the start and end addresses of the ROM window areas, and the segments that contain the start addresses will be shown as below.

```
    --- ROM WINDOW Start: 1000 ---
    S CODE   00:1000    00:100B    000C(12)       CSEG02
    S CODE   00:100C    00:1017    000C(12)       CSEG03
    --- ROM WINDOW Stop: 1FFF ---
```

The “Type” field first shows one of the single characters S, C, or Q. These indicate a segment, communal symbol, or quasi-segment, respectively. Next the “Type” field shows a segment type or a quasi-segment type listed in Table 5-11.

The “Start” and “Stop” fields show the start and stop addresses of the area occupied by the segment.

The “Size” field shows the size of the segment in hexadecimal and decimal.

The “Name” field shows the name of the segment. It shows “(absolute)” for absolute segments. It shows one of the names given in Table 5-11 for quasi-segments.

When segments or communal symbols were not allocated anywhere due to some error, the following will be shown.

```
Ignored 3 segments or communal symbols:

      Type     Start      Stop       Size          Name
     ----------------------------------------------------
      S CODE                         0400(1024)    PROC1
      S CODE                         0800(2048)    LOAD_PROC
      C DATA                         0002(2)       TMP_VAL
```

Also, one of the following messages might be displayed to the left of the “Type” field.

Message	Description
>GAP<	Indicates there is a free area in memory space.
OVL	Indicates segments overlap in memory.
—-	Indicates the physical segment address changed there.

(11) This line shows the total size of segments and communal symbols in hexadecimal and decimal. The sizes of quasi-segments are not included.

(12) This section shows the allocations to data memory space. First, the maximum address of the COMMON area is shown.

```
( COMMON MAX: 03FF )
```

Then the following message is shown to point out segments allocated in the COMMON area.

```
--- The above are in COMMON area ---
```

The start addresses of segments shown above this message exist within the COMMON area.

The format of other information is the same as for program memory space.

(13) This section shows the allocations to EEPROM space, when that space has been defined. First, the address range of EEPROM space is shown.

```
EEPROM Area: 4000 - 5FFF
```

The format of other information is the same as for program memory space.

(14) This section shows the allocations to dual port RAM space, when that space has been defined. First, the address range of dual port RAM space is shown.

```
DUAL_PORT_RAM Area: 6000 - 7FFF
```

The format of other information is the same as for program memory space.

Table 5-11. "Type" and "Name" Field Shown For Quasi-Segments

Quasi-Segment	"Type" Field Shown	"Name" Field Shown
SFR	SFR	(SFR)
XSFR	XSFR	(XSFR)
Pointing registers	PREG	(PREG*xx*)[*1]
Local register set	LREG	(LREG*yy*)[*2]
ROM gap	ROMGAP	(ROMGAP)
RAM gap	RAMGAP	(RAMGAP)
Dual port RAM	DUAL	(DUAL)
EEPROM	EEPROM	(EEPROM)
ROM window	ROMW	(ROMW)

*1 The *xx* is a number representing the bank number of the pointing register set.
*2 The *yy* is a number representing the bank number of the local register set.

The above are standard output items. RL66K will output several other types of information to the map file depending on the options. This additional information is explained below.

When the /D option is specified, RL66K will extract symbol information from debugging information that was output to the absolute object file, and output it to the map file. Symbols are gathered by module.

```
          ----------------------
           Symbol Table Synopsis
          ----------------------

Module        Value     Type          Symbol
------        -----     ----------    ------
MAIN
              00:0100   Loc CODE      START
              00:0240.0 Loc BIT       BSYM00
              00:6000   Loc DATA      DSYM20
              00:6010   Pub DATA      INDEX
GETDATA
              00:0509   Pub CODE      PUTDATA
              00:1000   Loc DATA      CBUF
              00:020C   Pub CODE      GETDATA
CALC
              00:0660   Pub CODE      CALC
              00:090C   Pub CODE      CHKVAL
```

The "Pub" and "Loc" of the "Type" field indicate public symbols and local symbols respectively.

When the /S option is specified, RL66K outputs in alphabetical order information about public symbols and communal symbols used in the program.

```
Public Symbols Regerence

Symbol    Value     Type    Module
------    -----     ----    ------
CALC      00:0660   CODE    CALC
CHKVAL    00:090C   CODE    CALC
GETDATA   00:020C   CODE    GETDATA
INDEX     00:6010   DATA    MAIN
PUTDATA   00:0509   CODE    GETDATA
```

If there were no errors other than warnings, then the following line will be output at the end of the map file.

```
End of mapfile.
```

5.7 RL66K Messages

This section describes all the messages output by RL66K. There are messages that indicate processing status and error messages that indicate problems.

5.7.1 Messages Indicating Processing Status

RL66K displays the following message to the screen immediately after it is invoked.

```
RL66K Object Linker, Ver.4.23 Jun 1993
Copyright (C) 1988-1993. Oki Electric Ind. Co.,Ltd.
```

It then proceeds with link processing and displays the following messages.

```
Loading segments and symbols...
Allocating segments...
Writing fixed data...
```

These messages have the following meanings.

- RL66K is loading segments and symbols.
- RL66K is allocating segments in memory space.
- RL66K is fixing up unresolved operands.

When all processing completes normally, RL66K displays the following message.

```
Linkage completed.

Absfile: absolute_file
Mapfile: map_file
```

The *absolute_file* and *map_file* are the names of the absolute file and map file created as a result of linking. If the /A option was specified to generate an ABL file, then the following message will also be displayed. The *abl_file* is the ABL file name.

```
ABLfile: abl_file
```

5.7.2 Error Message Format

When RL66K detects an error during processing, it displays a message to the screen. Error messages output by RL66K are command line error messages, fatal error messages, error messages, and warning messages.

Command line error messages are displayed when the RL66K command line input is invalid and processing cannot continue. When one of these errors occurs, RL66K discontinues processing immediately. Command line error messages are displayed with the following format.

```
Command line error: message
```

Fatal error messages are displayed when a clear error is found, and RL66K processing cannot continue. When one of these errors occurs, RL66K discontinues processing immediately. Fatal error messages are displayed with the following format.

```
Fatal error: message
```

Error messages are displayed when a clear error is found, and absolute object file is impossible. Error messages are displayed with the following format.

```
Error: message
```

When RL66K outputs an error message, it will also output one of the following messages corresponding to how far it has processed. It will then discontinue processing.

```
Discontinue! Loading error detedted.
Discontinue! Allocation error detedted.
Discontinue! Fix up error detedted.
```

These messages have the following meanings.

- An error occurred while loading segments and symbols. Processing will discontinue.
- An error occurred while allocating segments in memory space. Processing will discontinue.
- An error occurred while fixing up unresolved operands. Processing will discontinue.

Warning messages are displayed when something unusual but not fatal is found in the program. In these cases, RL66K processing will continue, and the absolute object file will be generated. Warning messages are displayed with the following format.

```
Warning: message
```

In addition to messages, RL66K outputs as much information as needed to understand the causes of errors, such as file names or symbol names. In particular, when an error is detected while RL66K is fixing up unresolved data, the location where the error occurred will be displayed after the message as follows.

```
offset / segment_name / module_name
```

This means that the error occurred at offset address offset in segment *segment_name*, which is defined in object module *module_name*.

The offset addresses correspond to values in the location column of the assembly list in the print file generated by RL66K.

■ **Example** ■

The assembly list of print file TEST.PRN generated by RL66K is shown below.

```
## Loc. Object        Line  Source Statements

                         1  TYPE    (m66507)
                         2  EXTRN   DATA: FIX_DAT DAT_BUF
                         3  PROC    SEGMENT  CODE
                         4          RSEG     PROC
??:0000 A4 00'97         5          MOV      A, fix FIX_DAT
??:0003 38               6          ST       A, ER0
??:0004 AC 00'           7          ADD      A, fix DAT_BUF
??:0006 DF               8          SWAP
```

Assume that when the object modules of this program were linked, the following message was displayed.

```
Error: Out of FIXED PAGE area, 0005/PROC/TEST
```

This means that an error occurred when RL66K tried to fix up the unresolved data at offset address 0005 in segment PROC, which was defined in the module TEST. The programmer can see from this message that the problem is the symbol DAT_BUF coded as the instruction operand on the seventh line.

5.7.3 Error Message Redirection

All error messages are sent to the standard output device. This device is the screen by default, but error messages can also be redirected to devices such as files and printers.

■ **Example** ■

```
RL66K MAIN WORD CALC; > ERRORS.MSG
```

In this example, all error messages generated during link processing will be output to the file ERRORS.MSG.

5.7.4 List Of Error Messages

The individual error messages displayed by RL66K are described in alphabetical order.

5.7.4.1 Command Line Error Messages

Bad constant

An address argument of an option is not correct. For example, this message might be displayed if an “H” is not appended to a hexadecimal constant, such as in /DM(4C0).

Command line syntax error

A syntax error was found in the command line. Input the command line again with correct syntax.

Missing object file name

No input file was specified. Specify at least one input file to call RL66K. For example, this message would be output if the following is invoked.

```
RL66K ;
```

Unrecognized option name

An incorrect option was found in the command line input.

5.7.4.2 Fatal Error Messages

All Machine names are STANDARD

All input modules are standard modules. At least one dedicated module must be given. A dedicated module is an object module that was created using a DCL file that specified a particular microcontroller name.

Bad module

Contents of an input module are corrupted. Reassemble the module and link again.

Cannot close file

The file cannot be closed.

Cannot open file

The file cannot be opened.

Cannot write, disk full!?

The file cannot be written. Disk capacity is probably insufficient. Free up enough space, and link again.

Check sum error

A check sum error was found in an input module. Reassemble the module and link again.

File seek error

A file seek error occurred.

File used in conflicting contexts

An input file and output file with the same name are specified. Specify a different name for the output file.

Illegal translation ID

A given module was not generated by RAS66K. It might have been generated by other software, such as RL66K or a debugger. RL66K handles only modules generated by RAS66K as input.

Inconsistent Machine name

The microcontroller name differs between input modules.

Invalid Core ID

An input module is not an object module for a CPU core that RL66K can handle.
The following are CPU cores that RL66K can handle.

nX-8/100, nX-8/200, nX-8/300, nX-8/400, and nX-8/500

Inconsistent Memory model

Memory models differ between input modules. Specify the same memory model in the program, reassemble, and link the object modules again.

Inconsistent Memory values

There is a problem with the combination of memory information between input modules. Define valid combinations of memory information in COMMON directives, WINDOW directives, and DCL instructions.

Insufficient memory

Memory is insufficient for RL66K to execute. The input modules may include too many symbols.

Invalid Family ID

An input module is not an object module for the OLMS-66K series.

Not a library file

A file specified as a library file is not a library file.

Record length too long

A given module includes a record that is too long. The module might have been corrupted. Reassemble the module and link again.

Specified module not found

A module specified as "library(modname...)" in the object_files field was not found in the library file. Verify the correct module name and link again.

Unexpected end of file

A given module does not end correctly. The module might have been corrupted. Reassemble the module and link again.

Version not compatible

A specified object module cannot be linked with the current version of RL66K. Reassemble with a compatible version of RAS66K and link again.

Each object module has information that indicates the software that created it and that software's version number. RL66K looks at the version number to determine if it can link a module.

5.7.4.3 Error Messages

Cannot find segment

A segment specified in a /CODE, /DATA, /BIT, /EDATA or /EBIT segment was not found. Verify the correct segment name and link again.

Cannot change physical segment

A /CODE, /DATA, /BIT, /EDATA or /EBIT directive tried to change a physical segment address specified during assembly. Physical segment addresses specified during assembly cannot be changed. For example, consider the following program.

```
SEG0    SEGMENT DATA
        RSEG    SEG0
TOP0:   DS      10H

SEG1    SEGMENT DATA #1
        RSEG    SEG1
TOP1:   DS      10H
        END
```

Segment SEG1 in this program is assigned physical segment address #1. If the following specifications are made to change the physical segment address of SEG1, this error will occur.

```
/DATA(SEG1-3:800H)
/DATA(SEG0-3:800H SEG1)
```

Control type mismatch

A /CODE, /DATA, /BIT, /EDATA or /EBIT directive tried to allocate a segment to the wrong memory space. For example, this might occur if a segment of usage type CODE is specified with a /DATA option.

Duplicate public symbol

The same public symbol is defined more than once in more than one module.

Group incomplete

RL66K tried to allocate all segments of a group in the same physical segment, but there was not enough free area. These segments will not be allocated anywhere.

Not allocated segment

A segment was not allocated to memory space.

Out of *area_name* area

A calculated value exceeds the range of an area. The *area_name* is the name of the area.

Out of range: *min* to *max*

A calculated value is not in the permitted range. The *min* and *max* are the minimum and maximum permitted values.

Physical segment address mismatch

The physical segment addresses of two partial segments linked as one segment do not match. The link processing will not be performed.

Physical segment attribute mismatch

The physical segment attributes of two partial segments linked as one segment do not match. The link processing will not be performed.

Segment size out of range

When two partial segments are linked, the permitted size is exceeded. The link processing will not be performed.

Segment type mismatch

The segment types of two partial segments linked as one segment do not match. The link processing will not be performed.

Special area specification mismatch

There is a problem with the combination of special area attributes of two partial segments linked as one segment. The link processing will not be performed.

Unresolved external symbol

An external symbol does not match any other public symbol or external symbol. The symbol declared as an external symbol must be declared as a public symbol or communal symbol in one other module.

Usage type mismatch

Usage types did not match when RL66K was matching external symbols, communal symbols, or public symbols. Both symbols must have the same usage type.

VCAL address must be even number

A VCAL address must be an even number.

5.7.4.4 Warning Messages

Branch to different segment area

The branch source and branch destination of a near branch instruction have different physical segment addresses.

Cannot access to high byte

An instruction is performing a word access on an odd address in RAM.

Cannot write to ROM WINDOW area

An instruction is trying to write to the ROM window area.

CODE segments overlap

A new segment is being allocated to an area in code memory space where another segment was previously allocated. This message is displayed when the following segments overlap.

- Absolute CODE segment
- Quasi-segment
- Relocatable segment specified with /CODE option

DATA/BIT segments overlap

A new segment is being allocated to an area in data memory space where another segment was previously allocated. This message is displayed when the following segments overlap.

- Absolute DATA segment
- Absolute BIT segment
- Quasi-segment
- Relocatable segment specified with /DATA option
- Relocatable segment specified with /BIT option

DUAL type DATA/BIT segments overlap

A new segment is being allocated to an area in dual port RAM space where another segment was previously allocated. This message is displayed when the following segments overlap.

- Absolute DATA segment
- Absolute BIT segment
- Relocatable segment specified with /DATA option
- Relocatable segment specified with /BIT option

EDATA/EBIT segments overlap

A new segment is being allocated to an area in EEPROM space where another segment was previously allocated. This message is displayed when the following segments overlap.

- Absolute EDATA segment
- Absolute EBIT segment

- Relocatable segment specified with /EDATA option
- Relocatable segment specified with /EBIT option

Ignore Boundary specification

An absolute address specified with a /CODE, /DATA, /BIT, /EDATA, or /EBIT option does not match the boundary value of the segment. RL66K will allocate the specified absolute address in the segment, ignoring the segment's boundary value attribute.

Ignore Group specification

The group attribute of a relocatable segment specified with a /CODE, /DATA, /BIT, /EDATA, or /EBIT option was ignored. RL66K removes that segment from the group before processing it.

No stack segment, ignore /STACK option

No stack segment exists, so the specified /STACK option will be ignored.

No stack segment, set 0 to _$$SSP

No stack segment exists, so the stack start address in stack symbol _$$SSP cannot be set. RL66K will set _$$SSP to 0.

Out of *area_name* area, due to allocation control

An absolute address specified with a /CODE, /DATA, /BIT, /EDATA, or /EBIT option is not in the relocation area of the segment. RL66K will allocate the specified absolute address in the segment, ignoring the segment's special area attribute. The *area_name* is the name of the special area attribute specified by the program for that segment.

For example, assume the fixed page area is 200H to 2FFH, and consider the following program.

```
SEG1    SEGMENT  DATA FIX
        RSEG     SEG1
TOP:    DS       10H
        END
```

This warning message will be displayed when an address outside the above range is specified for segment SEG1 of this program, as in the option below.

```
/DATA(SEG1-400H)
```

Out of ROM WINDOW area

The calculated code address is outside the ROM window area.

Over a page boundary, due to allocation control

An absolute address that was specified with a /CODE, /DATA, /BIT, /EDATA, or /EBIT option for a segment with the INPAGE attribute was allocated across a page boundary.

Specified stack size is too big, adjusted to *size16*(*size10*) bytes

There is not enough space to allocate a stack segment of the size specified by the program or /STACK option. RL66K will reduce the stack segment size and allocate it. The *size16*(*size10*) represent the reduced size in hexadecimal and decimal.

Stack size must be even number
An odd number was specified with the /STACK option for the stack size. The stack size must be an even number. RL66K will adjust the specified value up by one to make it even.

Using data type mismatch
DD states did not match when RL66K was linking segments, linking communal symbols, or matching external symbols, communal symbols, and public symbols.

Using operation type mismatch
SF states did not match when RL66K was linking segments, linking communal symbols, or matching external symbols, communal symbols, and public symbols.

Within ROM WINDOW area
The calculated data address is outside the ROM window area.

0 size segment detected
The size of a segment is 0. RL66K will allocate this segment to the highest address of memory space.

(USING DSREG check) out of RAM physical segment
The physical segment address of an operand value differs from the DSR value specified by the USING directive.

(USING PAGE check) out of current page
The current page value differs from the page value specified by the USING directive.

(USING TSREG check) out of ROM physical segment
The physical segment address of an operand value differs from the TSR value specified by the USING directive.

5.7.5 Internal Processing Error Messages

Internal processing error messages are displayed when there is a problem with RL66K's internal processing. The format of these messages is as follows.

RL66K internal error(position)

The position is a string that shows the location that generated the internal processing error. These errors should never occur, but if one does, then please inform Oki Electric of your RL66K version, the state when the error occurred, and the contents of position.

Chapter 6

LIB66K

The librarian LIB66K is the librarian for the OLMS-66K Series. It gathers multiple object files created by RAS66K into a single library file. Library files are used by the linker RL66K.

This chapter explains how to use LIB66K.

6.1 Introduction

LIB66K is software for managing library files.

A library file is a single file that gathers multiple object files that were created by RAS66K. It is created and modified using LIB66K. An object file that is added to a library file is called an object module. This chapter sometimes calls library files and object modules as simply libraries and modules.

Generated library files are used by RL66K.

6.1.1 LIB66K Functions

LIB66K functions are as follows. Section 6.3, “ LIB66K Operations” explains these functions in detail.

- Creates new library files.
- Adds object modules to library files.
- Adds library files to other library files.
- Deletes modules from library files.
- Replaces modules in library files with new modules.
- Copies modules in library files to object files.
- Extracts modules in library files to object files.
- Creates list files.

6.1.2 Advantages Of Using LIB66K

When you have created a program split into many modules, some modules will probably have generic use for other programs. A few of these generic modules will not be a problem, but as their number increases, it becomes difficult for the user to manage them all.

By registering these modules in a library, you can solve the problem described above. If you specify the library file when linking with RL66K, RL66K will search the library file for the necessary object modules.

6.1.3 Differences Between File Names And Module Names

LIB66K defines file names and module names as follows.

A file name indicates a DOS file name. A file name can specify a drive name, directory name, and extension.

A module name is the name that will indicate the module in the library file. This name is determined by RAS66K. RAS66K removes the drive name, directory name, and extension from the source file name specified in the command line, and uses the remaining base name as the module name. This information is then output in the object file.

Module names are case sensitive. This means that RAS66K may generate object files from the same source file but with different module names. Take the following for example.

```
RAS66K MODULE
```

This is coded with upper-case letters, so the module name will be “MODULE.”

```
ras66k module
```

This is coded with lower-case letters, so the module name will be “module.” The object file will be MODULE.OBJ in both cases.

Generate a list file to see the module names of the modules in the library file.

■ Example ■

```
LIB66K MYLIB;
```

This will generate the list file MYLIB.L66 of MYLIB.LIB.

To add a module using LIB66K, specify the object file name. To delete or copy from a library file, specify the module name.

6.2 Executing LIB66K

There are four ways to execute LIB66K.

- Execute from a command line
- Execute using prompts
- Use a combination of command line and prompts
- Use redirection

This section explains each of these in order.

6.2.1 Command Line Execution

In this execution method, all input is specified to LIB66K at the DOS prompt. Command line format is as follows.

```
LIB66K library_file [operations] [, [list_file]
[, [output_library_file]]] [;]
```

The number of characters in the command line can be up to the DOS limit of 127. To specify more than this, refer to Section 6.2.4, "Redirection." Note that wild card should not be contained in the file name.

Spaces delimit between *library_file* and *operations*. Use commas (,) to delimit all fields after *operations*. Options can be specified at any position on the command line. The contents specified in each field are shown in Table 6-1.

Table 6-1. Contents Specified In Each Field

Field	Contents specified
library_file	Input library file name (name of library file to be generated or modified).
operations	Operation on the library file specified by *library_file*.
list_file	List file name.
output_library_file	Output library file name (name of library file to generate by the operation).

The *library_file* cannot be omitted. The following fields can be omitted by specifying commas. If a semicolon (;) is specified, then all following fields will be omitted. In other words, a semicolon indicates the end of input. All characters after the semicolon until the carriage return are ignored. Default values will be used in fields omitted by comma or semicolon.

Table 6-2. Default Value Of Each Field

Field	Default value
operations	No operation performed.
list_file	List file generated. Its name will be that of the output library file with extension changed to ".L66."
output_library_file	The input file name specified with *library_file*. Invalid if library contents are not changed.

■ Example 1 ■

```
LIB66K MYLIB;
```

The *operations* are omitted, so no operation will be performed. Only a list file will be generated. The *list_file* specification is also omitted, so the list file name will be MYLIB.L66.

■ Example 2 ■

```
LIB66K MYLIB +CALC;
```

This command adds CALC.OBJ to MYLIB.LIB. The *list_file* and *output_library_file* are omitted, so the list file name will be MYLIB.L66 and the output library file name will be MYLIB.LIB.

(1) *library_file* Field

The *library_file* specifies the file name of the library file to be generated or operated upon. This field cannot be omitted. If this field is omitted from the command line, then LIB66K will display a prompt for it. If no file name is specified at this point, then LIB66K will display a brief explanation of how to use it and then terminate.

The default extension of the library file is ".LIB." For example, if MYLIB is specified, the LIB66K will interpret that as MYLIB.LIB. Extensions other than the default can be appended. To do that, specify up to and including the extension. To append no extension, append a period (.) to the end of the file name, as in "MYLIB.". If the drive name and directory name are not specified, then LIB66K will proceed as if the current drive and directory were specified.

If the specified library file does not exist, then a new library file will be created. For details, refer to Section 6.3.1, "Creating New Libraries."

(2) *operations* Field

The *operations* field determines the operation on the library specified with *library_file*. If this field is omitted, then no operation will be performed on the library file. However, LIB66K will check the library file and output a list file.

The operation is coded as an operation symbol (+, -, %, *, &) that expresses the operation and a file name or index name that will be operated on. When specifying multiple options, always separate them with spaces, as in this example.

```
    LIB66K MYLIB +ADCON +CALC + DISPLAY;
```

It does not matter if there are spaces after the operation symbols. The meanings of the operation symbols are as shown below.

Table 6-3. Operation Symbols

Function	Operation Symbol	Description
Add	+	Adds an object file or module in a library file to the library.
Delete	-	Deletes a module from the library.
Replace	%	Replaces a module in the library with a new module.
Copy	*	Copies a module in the library to an object file.
		The module will remain in the library.
Extract	&	Extracts a module in the library to an object file.
		The module will not remain in the library.

If the file name extension is omitted, then the default extension will be ".OBJ." If a drive name, directory name, or extension are specified for an module name, then those specifications will be ignored.

The operation symbols (-, %, &) can be used with file names or index names. The other operation symbols (+, *) can be used with file names, subject to the limitations of DOS.

■ Example ■

```
    LIB66K MYLIB +GET-KEY;
    LIB66K MYLIB +GET -KEY;
```

In the top example, GET-KEY.OBJ is added to MYLIB.LIB. In the bottom example, module KEY will be deleted from MYLIB.LIB, and then GET.OBJ will be added.

When multiple operations are coded, their order of execution will be determined by the precedence of the operations. For details, refer to Section 6.3.8, "Operation Precedence."

(3) *list_file* Field

The *list_file* field specifies the file name of the list file. The list file is a text file that shows in alphabetical order information about the modules in the library along with the public symbols included in those modules. For details on list files, refer to Section 6.4, "List Files."

The list file will be generated by default even if not explicitly specified. The file name in this case will be the output library file name with the extension changed to ".L66." When you do not want to use the default file name, specify a file name in this field.

To not generate the list file, specify NUL in this field. If CON is specified instead of NUL, then the list file will be output to the screen. If PRN is specified instead of NUL, then the list file will be output to the printer.

■ Example 1 ■

```
LIB66K MYLIB +CALC, C:\WORK\LIBLIST. ;
```

In this example, the list file name will be C:\WORK\LIBLIST.

■ Example 2 ■

```
LIB66K MYLIB %CALC, NUL;
```

NUL is specified in this example, so a list file will not be generated.

(4) *output_library_file* Field

The *output_library_file* field specifies the output library file name. If this specification is omitted, then the output library file name will be the same name as the input library file. When you want to change the default file name, you must specify this field.

If a file with the same name as the output library file already exists, then LIB66K will change the extension of that file to ".LBK" and make a backup.

■ Example ■

```
LIB66K MYLIB -DISPLAY,,NEWLIB
```

In this example, module DISPLAY is deleted from MYLIB.LIB, and the result is output to NEWLIB.LIB. If NEWLIB.LIB already exists, then LIB66K will change its name to NEWLIB.LBK and make a backup. MYLIB.LIB will not be rewritten. A list file NEWLIB.L66 will be generated.

This field is valid only when the contents of the library file are changed. When no operation is specified, or when the operations are only copies, or when a new library will be created, this field will have no meaning even if specified, so LIB66K will output a warning.

(5) Options

Options can be specified at any position. Options are used to restrict the modules that can be registered in the library file. They can be used only when creating a new library file. If they are specified in any other case, then they will be ignored. Options can be specified any number of times, but only the last one specified will be valid.

Option types are listed below. Spaces must not be inserted after the slash (/).

Option	Function
/100	Generate library for nX-8/100 CPU core.
/200	Generate library for nX-8/200 CPU core.
/300	Generate library for nX-8/300 CPU core.
/400	Generate library for nX-8/400 CPU core.
/500	Generate library for nX-8/500 CPU core.
/[500]S	Generate library for SMALL memory model of nX-8/500 CPU core.
/[500]C	Generate library for COMPACT memory model of nX-8/500 CPU core.
/[500]M	Generate library for MEDIUM memory model of nX-8/500 CPU core.
/[500]L	Generate library for LARGE memory model of nX-8/500 CPU core.

When the /S, /C, /M, and /L options for restricting memory model are specified, the CPU core will automatically be restricted to nX-8/500. Therefore an error will occur for specifications such as /200S and /L300.

If no options are specified when creating a new library file, then any file can be registered in it, regardless of CPU core or memory model.

Generate a list file to see the settings of the library file.

■ Example 1 ■

```
LIB66K NEW-FILE /300 +FOR300 +FOR500;
```

This creates the new library file NEW-FILE.LIB. Because the /300 option is specified, object files that can be registered are restricted to only those generated for the nX-8/300 CPU core.

The above example attempts to register two object files in NEW-FILE.LIB. FOR300.OBJ is an object file generated for nX-8/300, and FOR500.OBJ is an object file generated for nX-8/500. In this example, FOR300.OBJ will be registered correctly in NEW-FILE.LIB, but FOR500.OBJ will not be registered because it has a different CPU core.

■ Example 2 ■

```
LIB66K NEW-FILE /L +SMALL +LARGE;
```

This creates the new library file NEW-FILE.LIB. Because the /L option is specified, object files that can be registered are restricted to only those generated for the LARGE memory model of the nX-8/500 CPU core.

The above example attempts to register two object files in NEW-FILE.LIB. SMALL.OBJ is an object file generated for the SMALL memory model, and LARGE.OBJ is an object file generated for the LARGE memory model. In this example, LARGE.OBJ will be registered correctly in NEW-FILE.LIB, but SMALL.OBJ will not be registered because it is a different memory model.

6.2.2 Prompt-Based Execution

In this execution method, all input is specified in response to prompts output by LIB66K. When LIB66K is typed at the DOS prompt, LIB66K will display the following prompts one line at a time and wait for user response. LIB66K will not display the next prompt until the user responds to the current prompt. If there is any input, then LIB66K will display the next prompt.

```
Library file     :
Operations       :
List file        :
Output library   :
```

These prompts correspond to the fields of the command line.

Table 6-4. Prompts And Corresponding Command Line Fields

Prompt		Command Line Field
Library file	:	library_file field
Operations	:	operations field
List file	:	list_file field
Output library	:	output_library_file field

Options can be input at any prompt.

■ Example 1 ■

```
Library file  :  ABC
Operations    :  +A +B +C;
```

This example adds object files A.OBJ, B. OBJ, and C.OBJ to the library file ABC. It will also generate the list file ABC.L66.

■ Example 2 ■

```
Library file   :  ABC
Operations     :  %B:XYZ
List file      :  NUL
OUTPUT LIBRARY :  B:ABC
```

In this example, module XYZ of library file ABC.LIB will be replaced by object file B:XYZ.OBJ. The result will be output to B:ABC.LIB. The original ABC.LIB will not be rewritten. NUL is specified, so a list file will not be generated.

■ Example 3 ■

```
Library file      :    MYLIB;
```

or

```
Library file      :    MYLIB
Operations        :    ,
```

Both of these examples operate the same. Both generate list file MYLIB.L66 of library file MYLIB.LIB. The library file will not change.

■ Example 4 ■

```
Library file      :    MYLIB
Operations        :    -DISPLAY
List file         :    ,
Output library    :    NEWLIB
```

In this example, module DISPLAY will be deleted from library file MYLIB.LIB, and the result will be output to NEWLIB.LIB. If NEWLIB.LIB already exists, then LIB66K will change its name to NEWLIB.LBK and make a backup. MYLIB.LIB will not be rewritten. A list file NEWLIB.L66 will be generated.

6.2.3 Using Command Line And Prompts Together

When the command line is insufficient input to LIB66K, then LIB66K will prompt for the missing input. For example, when the following line is entered, LIB66K will display prompts after the *operations*.

```
LIB66K MYLIB +CALC
```

When the following prompts are answered, LIB66K will execute.

```
List file         :
Output library    :
```

To prevent these prompts from being displayed, add a semicolon at the end of the command line to tell LIB66K that input has ended.

```
LIB66K MYLIB +CALC;
```

In this case, prompts will not be displayed.

6.2.4 Redirection

The DOS redirection function can be used when LIB66K is invoked. This is convenient for commands that are too long to fit on one command line or for repeating the same operation many times.

First, create a file for redirection using an editor. Write the needed operations in either command line or prompt input format. Then redirect LIB66K to this file.

■ Example ■

The file COMFILE is coded in command line input format. All fields are specified.

```
MYLIB +A +B +C +D +E +F +G +H +I +J +K +L, PRN, OUTLIB;
```

Type the following DOS command line to execute.

```
LIB66K < COMFILE
```

6.2.5 Redirecting Output Messages

Messages that LIB66K displays on the screen are all output to the standard output device. Thus, messages can be output to a file using the DOS redirection function. If the redirection destination is PRN, then messages can be output to a printer. To prevent messages from being output to the screen, specify NUL.

■ Example 1 ■

```
LIB66K MYLIB + ERR; > ERRMES
```

Messages displayed to the screen will be redirected to file ERRMES.

■ Example 2 ■

```
LIB66K MYLIB %A %B %C; > PRN
```

Messages displayed to the screen will be output to the printer.

■ Example 3 ■

```
LIB66K < COMFILE > NUL
```

LIB66K will execute the operations coded in the file COMFILE. It will not display messages to the screen.

6.2.6 Termination Code

LIB66K returns the following termination codes to DOS when it terminates.

Table 6-5. Termination Codes

Termination Code	Termination Status	Description
0	Normal termination	No errors occurred.
1	Warning	Operations with problems were executed.
2	Error	Operations with problems were ignored (other operations were executed).
3	Fatal error	Execution could not be continued due to problems in operations.

Refer to Section 6.5, “Error Messages,” regarding fatal errors, errors, and warnings.

6.3 LIB66K Operations

LIB66K provides the following operations.

(1) Creating new libraries.
(2) Adding modules.
(3) Adding library files.
(4) Deleting modules.
(5) Replacing modules.
(6) Copying modules.
(7) Extracting modules.
(8) Generating list files.

This section describes operations (1) to (7) in detail.

The list files of (8) are generated by default. Section 6.2.1, "*list_file* Field," explains how to specify list file names. Section 6.4, "List File Format," describes list file format.

6.3.1 Creating New Libraries

To create a new library, specify its file name in the *library_file* field of the command line or at the "Library_file" prompt.

If the specified file name has no extension, then LIB66K will automatically append ".LIB." Other extensions can be specified, but both LIB66K and RL66K use ".LIB" as the default extension for library file names, so that extension is recommended.

A drive name and directory name can be specified in the library file name. If they are not specified, then the library file will be created in the current drive and current directory.

Unless a name other than of a library file is specified when LIB66K is invoked, the following prompt will be displayed.

```
file_name.LIB - File does not exist, Create ? [Y/N]
```

If either 'n' or 'N' is entered here, then the file will not be created and LIB66K will terminate. Enter a 'y' or 'Y' to create the file. A carriage return has the same meaning as 'Y.'

If other fields are specified, then LIB66K will assume that the library file is to be created. In such cases, the prompt will not be displayed. This is identical to when there is a semicolon or other fields in the command are omitted.

■ Example 1 ■

```
   LIB66K NEWLIB
```

LIB66K will display a prompt asking if the operation is valid or not.

■ Example 2 ■

```
   LIB66K NEWLIB;
```

LIB66K will create NEWLIB.LIB and NEWLIB.L66 without displaying a prompt. No modules will be entered in NEWLIB.LIB.

■ Example 3 ■

```
   LIB66K NEWLIB /500S;
```

LIB66K will create NEWLIB.LIB and NEWLIB.L66 without displaying a prompt. Object files that can be registered in NEWLIB.LIB are restricted to only those generated for a SMALL memory model of the nX-8/500 CPU core.

■ Example 4 ■

```
   LIB66K NEWLIB +A;
```

LIB66K will create NEWLIB.L66 and NEWLIB.LIB containing module A without displaying a prompt.

When a new library is created, operations other than addition will cause an error. There will be no modules in the library file, so deletion, copying, replacement, and extraction will be invalid.

When a new library is created, the output library file cannot be specified. If specified, then LIB66K will issue a warning and ignore the output library file specification.

6.3.2 Adding Modules

■ **Syntax** ■

```
+object_file
```

■ **Description** ■

The ‘+’ will add the specified object file to the library. The *object_file* specifies the name of the object file to be added to the library. DOS file names can be used in *object_file*. If the file extension is omitted, then the default extension “.OBJ” will be added.

LIB66K takes the base name of the specified file as the module name. The modules in the created library file will be in alphabetic order by module name.

When LIB66K adds a module to a library, it performs some checks on the modules that have already been added. These checks are as follows.

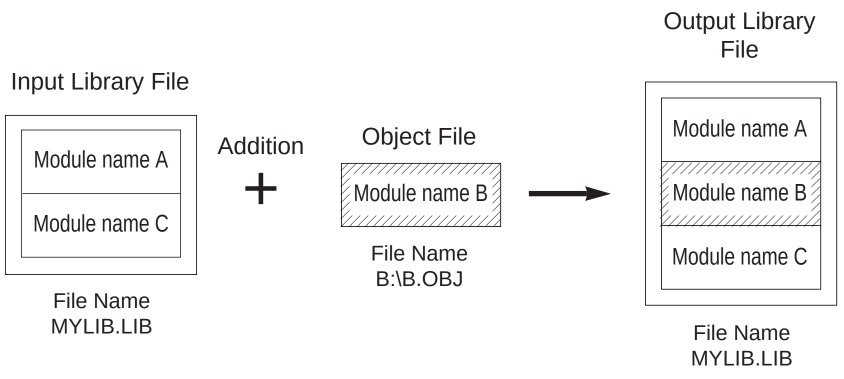
(1) Does a module with the same module name already exist in the library?

(2) Are the same public symbols declared in the module to be added already declared in other modules of the library? (Public symbols are symbols declared using the PUBLIC directive of RAS66K. They can be referred from other modules. For details, refer to Section 4.12.13.1, “Public Symbol Declarations.”)

If there is a single error in any of these checks, then LIB66K will not add the module.

■ **Example** ■

```
LIB66K MYLIB +B:\B. OBJ;
```



6.3.3 Adding Library Files

■ Syntax ■

```
+library_file
```

■ Description ■

When a library file is specified after the '+' all of the modules in it will be added to the input library file. DOS file names can be specified in the *library_file*. However, the extension cannot be omitted. This is the *operations* field, so if the file name extension is omitted, then LIB66K will assume an extension of ".OBJ." The library file extension can be different than ".LIB."

When LIB66K adds a module to a library, it checks whether that module can be added. These checks are as follows.

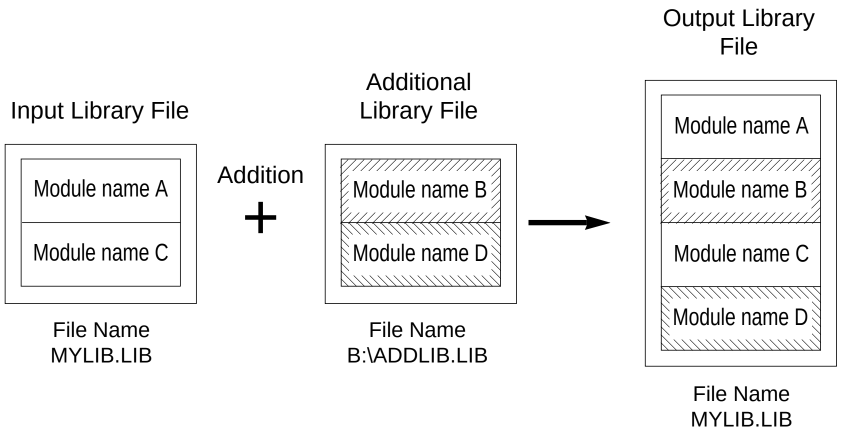
(1) Does a module with the same name as the module to be added exist in the library being modified?
(2) Are the same public symbols declared in the module to be added already declared in other modules of the library being modified?

If there is a single error in any of these checks, then LIB66K will not add the module. Other modules that had no errors will be added.

The modules in the created library file will be in alphabetic order by module name.

■ Example ■

```
LIB66K MYLIB +B:\ADDLIB.LIB;
```



6.3.4 Deleting Modules

■ Syntax ■

```
-module_name
```

■ Description

Use ‘-’ to delete a module from a library file. The *module_name* specifies the name of the module to be deleted. The following example deletes module B from library MYLIB.LIB.

```
LIB66K MYLIB -B;
```

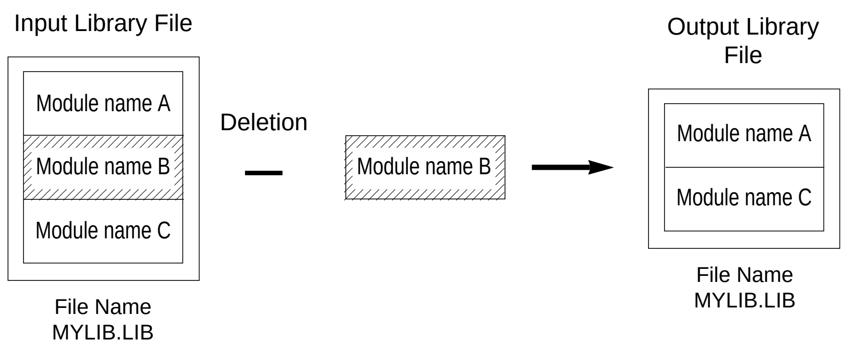
If *module_name* specifies a drive name, directory name, or extension, then they will be ignored. The following example deletes module B from library MYLIB.LIB.

```
LIB66K MYLIB -C:\WORK\B;
```

If the specified module name is not found in the library, then an error occurs.

■ Example ■

```
LIB66K MYLIB -B;
```



6.3.5 Replacing Modules

■ **Syntax** ■

```
%object_file
```

■ **Description** ■

The '%' replaces a module in the library will the specified object file. The *object_file* specifies the name of the object file to be replaced in the library. DOS file names can be used in *object_file*. If the file extension is omitted, then the default extension ".OBJ" will be added.

LIB66K takes the base name of the specified file as the module name. This module name indicates the module in the library.

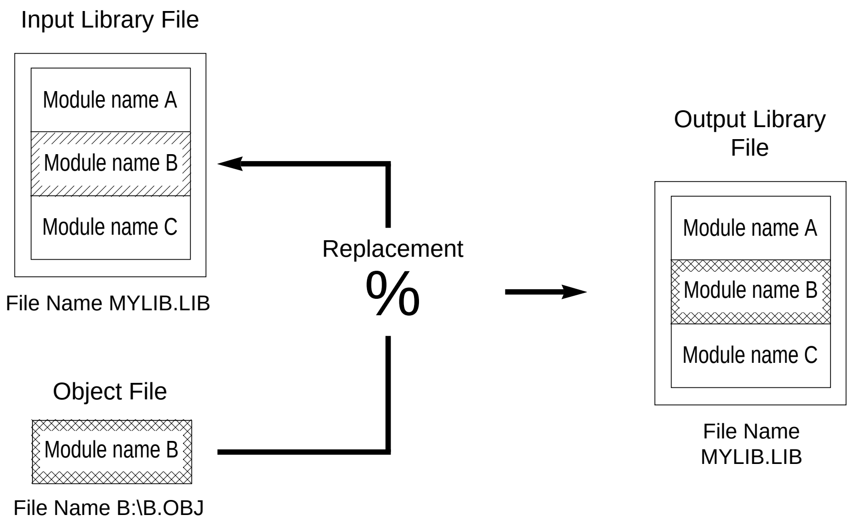
LIB66K first performs the following checks. If there is a single error in any of these checks, then LIB66K will not replace the module.

(1) Is the specified module name in the library?
(2) Do the names of the library module and replacement module match?
(3) Are the public symbols in the replacement module not defined in other library modules? (The public symbols of the replaced module are not considered.)

If no errors occur in the above checks, then LIB66K will delete the specified module from the library. If the deletion finishes normally, then LIB66K will add the object file. If the specified object file does not exist or an error occurs, then LIB66K will leave the library as is, without deleting the module for replacement.

■ **Example** ■

```
LIB66K MYLIB %B:\B.OBJ;
```



6.3.6 Copying Modules

■ **Syntax** ■

```
*object_file
```

■ **Description** ■

The ‘*’ extracts a module from the library and copies to the object file. The copied module will remain in the library.

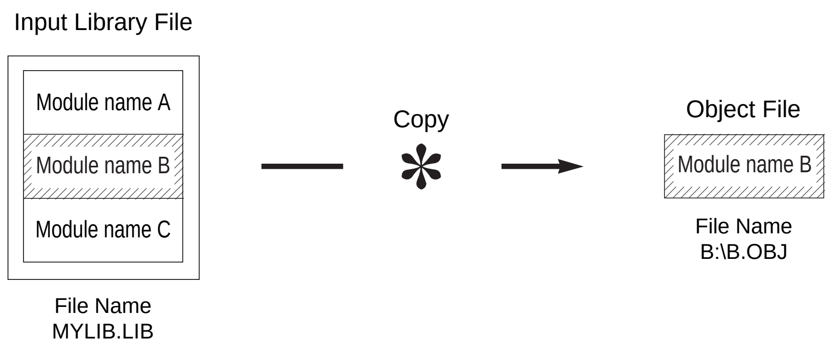
DOS file names can be used in *object_file*. If the file extension is omitted, then the default extension “.OBJ” will be added.

LIB66K takes the base name of the specified file as the module name. LIB66K will search for the module in the library using this module name. If it cannot find the module, then an error will occur and this operation will be ignored. If it can find the module, then it creates an object file with the name specified in *object_file* and copies the module to it. If a file with the same name already exists, then LIB66K will output a warning message and overwrite the module.

This operation does not modify the contents of the library. Therefore, when only copy operations are specified, an output library file will not be created. In this case, if an output library file was specified when LIB66K was invoked, then an error will occur. No backup file will be created.

■ **Example** ■

```
LIB66K MYLIB *B:\B.OBJ;
```



6.3.7 Extracting Modules

■ Syntax ■

```
&object_file
```

■ Description ■

The ‘&’ extracts a module from the library and copies to the object file. The copied module will be deleted from the library. This operation is the same as a module copy (*) followed by a module deletion (-).

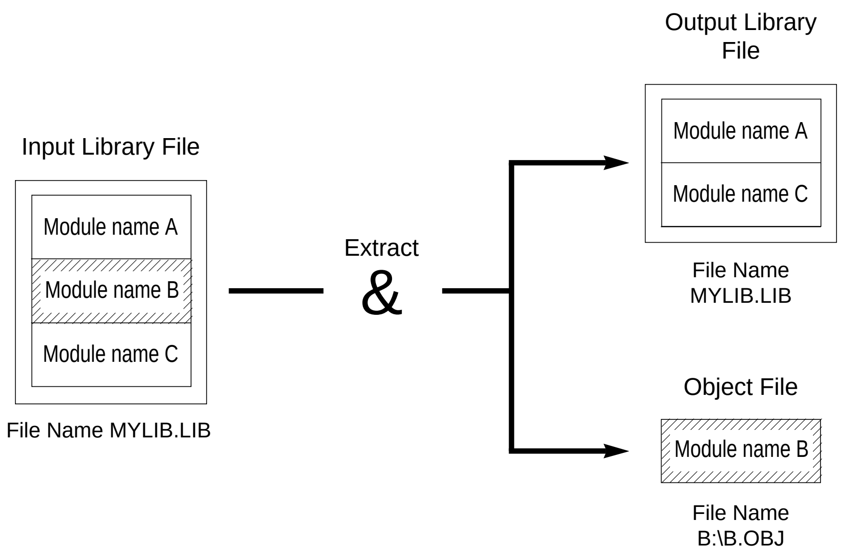
DOS file names can be used in *object_file*. If the file extension is omitted, then the default extension “.OBJ” will be added.

LIB66K takes the base name of the specified file as the module name. LIB66K will search for the module in the library using this module name. If it cannot find the module, then an error will occur and this operation will be ignored. If it can find the module, then it creates an object file with the name specified in *object_file* and copies the module to it. If a file with the same name already exists, then LIB66K will output a warning message and overwrite the module.

When the copy is complete, LIB66K will delete the module from the library.

■ Example ■

```
LIB66K MYLIB &B:\B.OBJ;
```



6.3.8 Operation Precedence

Operations have precedence. LIB66K executes operations from highest precedence to lowest, regardless of their order of specification. A table of precedence is shown below. A precedence of 1 is the highest. When two operations have the same precedence, they will be executed in their order of specification (from left to right).

Table 6-6. Operation Precedence

Precedence	Operations
1	Delete (-) Copy (*) Extract (&)
2	Replace (%)
3	Add (+)

■ Example ■

```
LIB66K MYLIB +ADCON %RAMCHK -DISPLAY  *CALC  &EXTMEM;
```

LIB66K first deletes module DISPLAY, copies module CALC, and extracts module EXTMEM from library MYLIB. It then replaces module RAMCHK, and finally adds ADCON.OBJ to the library.

6.3.9 Cautions During Execution

(1) Disk Capacity

LIB66K normally creates temporary files and backup files. Their directories and sizes are shown in the table below. When LIB66K is executed, there must be sufficient disk capacity to create these files.

Table 6-7. Directories And Sizes Of Temporary Files And Backup Files

File	Created In Directory	File Size
Temporary file	Same directory as output library file	Same size as output library file
Backup file	Same directory as input library file	Same size as input library file

(2) Temporary Files

LIB66K creates a temporary file in the same directory as the output library file. LIB66K performs its operations using this file. When the operations complete normally, the temporary file's name is changed to the output library file name. Accordingly, the temporary file will not remain in the directory.

However, if a fatal error occurs during execution or if the user interrupts LIB66K execution with a CTRL+C, then the temporary file may be left as is. Even though the temporary file remains, LIB66K operation will not be incorrect.

The temporary file name will be "$LIB66K$.$$$," so the user should avoid creating files with the same name. If temporary file is to be handled as a library file, then change the file name with using the DOS REN command.

When the library contents do not change, LIB66K will not create a temporary file. This will happen in the following cases.

(1) No options are specified.
(2) Only copy operations are specified.

6.4 List File Format

The list file is a text file that shows the contents of the library. It gives information about the library itself, all modules included in the library, and public symbols defined in those modules.

A list file is always generated by default. To not generate a list file, specify “NUL” in the *list_file* field.

An example list file is shown below. The numbers (1) to (14) are added for the explanation, but they are not shown in the actual list.

```
LIB66K Object Librarian, Ver.4.00 Library Information
[Mon Nov 09  19:49:38  1992].......... (1)

LIBRARY FILE : TEST.LIB.......... (2)
MODULE COUNT : 1.............................. (3)
CPU CORE     : nX-8/500.......... (4)
MEMORY MODEL : COMPACT............ (5)
                :............. (6)          :..........(7)      :......... (8)
MODULE NAME  : TEST               11-09-1992    19:49:38
BYTE SIZE    : 000001A8H(424) ..........(9)
CORE ID      : nX-8/500.........................(10)
MEMORY MODEL : COMPACT .......................... (11)
TRANSLATOR   : RAS66K(Ver.4.00)......... (12)
TARGET       : MSM66507.........................(13)

= PUBLIC SYMBOLS =           :......... (14)
BUF      FLAG     LOOP     NUM
```

Figure 6-1. List File Example

The numbers added to the above list file example are explained in order. (1) to (5) show information about the library itself. (6) to (14) show information about each module in the library file.

(1) This is the date when LIB66K was executed. It is displayed in the following format.
[day month date hour:minute:second year]

(2) This is the library file name.

(3) This is the number of modules included in the library.

(4) This shows the CPU core restriction. It will not be displayed when the library file is not restricted to a CPU core.

(5) This shows the memory model restriction. It will not be displayed when the library file is not restricted to a memory model.

(6) This is a module name. Module names can be up to 32 characters in length.

(7) This is the date that the module was registered in the library. It is shown in the format “month-date-year.”

(8) This is the time that the module was registered in the library. It is shown in the format “hour:minute:second.”

(9) This is the size of the module in bytes. It is shown as a hexadecimal number followed by the equivalent decimal number in parentheses.

(10) This is the target CPU core. It shows which CPU core the module was created for.

(11) This is the target memory model. It is shown only when the CPU core is nX-8/500. It shows which memory model the module was created for.

(12) This is the name of the assembler that created the module. The assembler version is shown in parentheses.

(13) This is the target microcontroller of the module. It shows the device name of the microcontroller type.

(14) These are public symbols declared in the modules. They are shown in alphabetic order. If there are no public symbols, then “-None-” will be shown.

6.5 Error Messages

LIB66K outputs three types of error messages: fatal errors, errors, and warnings.

● **Fatal errors**

When a fatal error occurs, LIB66K displays an error message and immediately stops operation. Existing files are not changed. The temporary file might remain on disk when a fatal error occurs. Refer to Section 6.3.9 (2), “Temporary Files,” regarding processing temporary files.

● **Errors**

When an error occurs, LIB66K will ignore the operation that caused the error, but will perform other operations. The file it creates can be used. LIB66K will display the number of errors when it terminates.

● **Warnings**

When a warning occurs, LIB66K will still execute the operation that caused the warning. The file it creates can be used. LIB66K will display the number of warnings when it terminates.

6.5.1 Error Message Format

Error messages are displayed on the screen in the following format.

```
error_level:error_message
```

Depending on the error, an additional line may be displayed in order to provide more information about the error. The additional line will be displayed in one of the formats below.

Library file : *library_file*   Module name : *module_name*
Library file : *library_file*   Module : *module_name*   Offset : *XXXX*H
Library file : *library_file*   Offset : *XXXX*H
Module name : *module_name*   Offset : *XXXX*H

The explanation below uses the above formats.

Table 6-8. Symbols Used To Explain Error Messages

Symbol Used In Manual	Contents Displayed On Screen
error_level	Either “Fatal error,” “Error,” or “Warning.”
error_message	Message indicating contents of the error.
library_file	The name of the library file that caused the error.
module_name	The name of the module that caused the error.
XXXX	The location of the error (an offset from the beginning of the file), displayed in hexadecimal.

The error messages are explained below divided by error type. Error messages are shown in alphabetical order.

6.5.2 Fatal Error Messages

“*file_name*” is write-protected

The output library file or backup file shown in *file_name* is a read-only file. Cancel the file’s read-only attribute by using the DOS ATTRIB command.

Bad input format

The command line format is improper.

Bad object filename specification

There was no object file name given after an operation symbol, or the object file name contains an invalid character.

Cannot open temporary file.

The temporary file “$LIB66K$.$$$” is a read-only file. Cancel the file’s read-only attribute by using the DOS ATTRIB command.

Cannot rename old library

The input library file cannot be changed to a backup file. This message will be displayed when the extension specified for the input library file is “.LBK.”

LIB66K does not check the extension of the input library file. When the library is to be rewritten, the library file name’s extension will be changed unconditionally to “.LBK.” However, if the input library file name’s extension is already “.LBK,” then it cannot be changed to the backup file name.

To handle the backup file as an input library file, change the file name using the DOS REN command.

Checksum error

The checksum of the record currently being processed is incorrect. The file may be corrupted. Execute LIB66K after rebuilding the file.

Disk full error

Disk capacity is insufficient. Move or delete unnecessary file to create a free area on the disk.

EOF expected after module index records

There is no EOF at the end of the library file. The library file is corrupted.

File name too long

The length of a file name or module name specified when LIB66K was invoked exceeds 255 characters.

File operation failure

A seek error occurred.

I/O read error

Data could not be read from the file.

I/O write error

Data could not be written to the file.

Illegal input

An EOF was input at an LIB66K prompt. An EOF was encountered before the end of LIB66K specification when redirection was used to invoke LIB66K. The line below is an example.

```
MYLIB %ADCON %CALC %DISPLAY , PRN ,<EOF>
```

The line needs to end with a carriage return code, as shown below.

```
MYLIB %ADCON %CALC %DISPLAY , PRN ,<carriage return code>
```

The line could also end with a semicolon.

```
MYLIB %ADCON %CALC %DISPLAY , PRN ;
```

Insufficient memory

Memory is insufficient for LIB66K execution. Increase usable memory by releasing resident programs and reducing device drivers.

Invalid library file

The specified library file includes an abnormal record or information.

Library file might be corrupted

The library file contains some improper information.

Record length too long for processing

An input file record was too long to be processed. This message will be displayed if the record length exceeds 0FFECH bytes.

Too many public symbols

There are too many public symbols to be registered in the library. Create a separate library or remove public declarations for symbols that do not need to be public symbols (that are not referred from other modules). The number of public symbols that can be registered in a single library file will be 1844 when all symbols are the maximum 32 characters long.

Unable to create new library file

The output library file could not be created.

Unable to open library file

The input library file could not be opened.

Unexpected end of file

LIB66K could not read data that it expected to exist. This error will be output when the file ends prematurely. The specified file may be corrupted.

6.5.3 Error Messages

Cannot exceed 65535 modules; ignored

Up to 65535 modules can be registered in a library file. Further modules cannot be registered. Create a separate library file.

CPU core is different; ignored

A module of one CPU core would be registered in a library restricted to a different CPU core. The module will not be registered.

Invalid object module; ignored

The specified object module is improper. Specify an object file generated by RAS66K when adding a module to the library file.

Memory model is different; ignored

A module of one memory model would be registered in a library restricted to a different memory model. The module will not be registered.

Module already included in the library; ignored

The specified module is already registered in the library. This module will not be registered.

Module name redefinition; ignored

The name of the specified module is already registered in the library. This module will not be registered.

Module names are different; ignored

This error occurs only during module replacement. The module in the object file is different from the module in the library file. This module will not be registered.

Module not found in the library; ignored

The module specified to be copied (*), extracted (&), or deleted (-) is not registered in the library. This operation will be ignored.

No space in the library; ignored

The library file is too big for the module to be registered. Delete unneeded modules, or create a new library file and register the module there.

Public symbol redefinition; ignored

A public symbol of a module to be registered is already defined an another module. This module will not be registered.

Unable to open file; ignored

A file specified for a copy (*) or extract (&) operation could not be opened. This operation will be ignored.

Unable to open list file

The list file could not be opened.

Unable to open object file; ignored

The specified object file could not be opened. This file will be ignored.

6.5.4 Warning Messages

File already exists in the directory; overwritten

This warning occurs only for copy (*) or extract (&) operations. The specified file already exists. LIB66K will overwrite the file, erasing the previous object file.

Module not found in the library; ignored

This warning occurs only for replacement (*) operations. The specified module does not exist in the library.

Output library specification; ignored

Even though the input library file will not be rewritten, an output library file was specified. The output library file specification will be ignored. This message is displayed in the following cases.

1) The input library file does not exist (when a new library is being created).
2) No operation was specified.
3) Only copy (*) operations are specified.

Chapter 7

OH66K

This chapter explains how to use OH66K and describes the files generated by OH66K.

OH66K is software that converts absolute object files generated by RAS66K or RL66K into HEX files.

7.1 Introduction

The object converter OH66K is software that converts absolute object files into HEX files.

Absolute object files are object files that do not contain relocatable information. Generally these are .ABS files generated by RL66K. Some .OBJ files generated by RAS66K can also be converted, but they must have been written without any relocatable code.

The HEX files generated by OH66K have two possible formats, either Intel HEX format or Motorola S2 format. This chapter calls files in Intel HEX format Intel HEX files, and files in Motorola S2 format S2 format files. Refer to Section 7.3.2, "Output Files," for a detailed explanation of HEX files.

The generated HEX file can be used with emulators or PROM programmers. To do symbolic debugging with an emulator, output debugging information by specifying the /D option. Refer to "(3) Options" of Section 7.2.1, "Command Line Conversion," regarding options. The format of debugging information is explained in Section 7.3.2 (3), "Debugging Information."

Figure 7-1 shows the basic concepts of OH66K input and output.

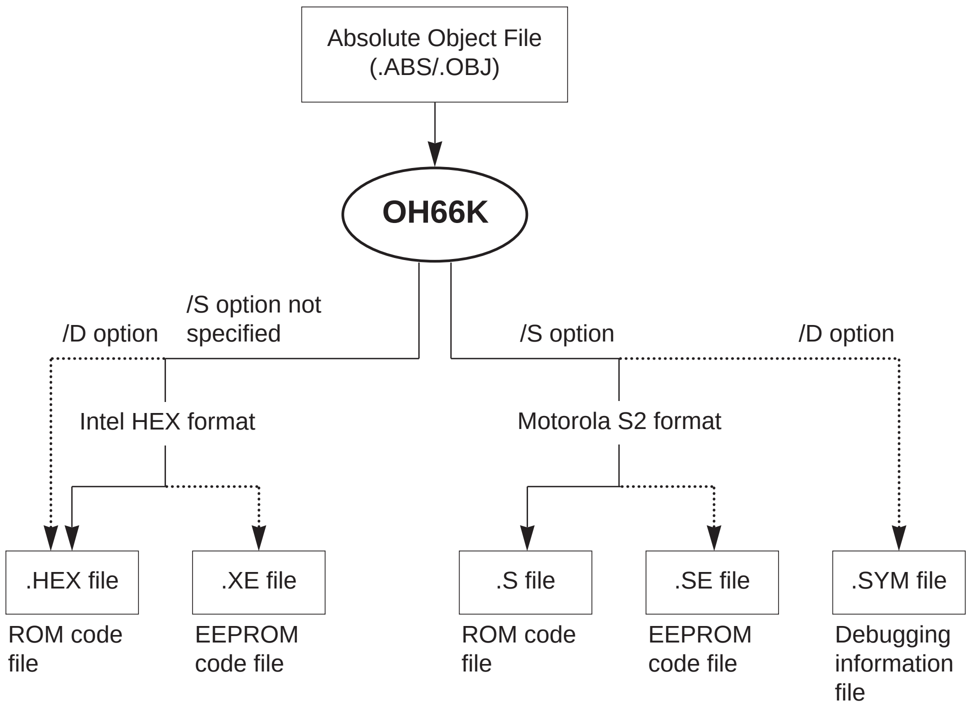


Figure 7-1. OH66K Input/Output

If the /S option is not specified, then the output file will be in Intel HEX format. If the /S option is specified, then the output file will be in Motorola S2 format.

A ROM code file is a HEX file converted from object code located in the program area. An EEPROM code file is a HEX file converted from object code located in the EEPROM area.

An EEPROM code file is generated only when the absolute object file includes object code located in the EEPROM area. Otherwise only a ROM code file will be generated.

When the /D option is specified with Intel HEX format, debugging information will be output at the start of the ROM code file with extension ".HEX." When the /D option is specified with Motorola S2 format, debugging information will be output to a debugging information file with extension ".SYM."

7.2 Using OH66K

OH66K's operation can be invoked in the following ways.

(1) Conversion from DOS command line
(2) Conversion using prompts output by OH66K

This sections describes these methods in order.

7.2.1 Command Line Conversion

This method specifies OH66K at the DOS prompt. Command line format is as follows.

```
OH66K object_file [hex_file] [;]
```

Separate the fields with spaces. The semicolon (;) indicates end of input, so all characters after it will be ignored. If a field is not specified fully, then OH66K will use default specifications. Options can be specified anywhere on the command line.

■ **Example** ■

```
OH66K TEST;
```

This example converts TEST.ABS to create TEST.HEX.

Each field is explained below.

(1) *object_file* Field

This field contains the name of the absolute object file to be converted by OH66K. This field specification cannot be omitted. If omitted, then OH66K will output a prompt. If *object_file* is not specified at the prompt either, then OH66K will display an explanation of its use on the screen and then terminate.

If the *object_file* has no extension, then the default extension ".ABS" will be used. A drive name and directory name can be specified in *object_file*. If not specified, then OH66K will assume the current drive and current directory.

(2) *hex_file* Field

This field specifies the name of the HEX file to be generated by OH66K. A drive name and directory name can be specified, but if an extension is specified, then an error will occur.

This field can be omitted. To omit it, specify a semicolon as follows.

```
OH66K TEST;
```

If omitted, then the input file name extension will be changed to match the format of the output file. The output file format can be specified by options, which will be explained later.

File Format And Extension

File Format	ROM Code File	EEPROM Code File
Intel HEX format	.HEX	.XE
Motorola S2 format	.S	.SE

■ Example 1 ■

```
OH66K TEST SAMPLE
```

In this example, the HEX file name is specified. OH66K will convert TEST.ABS and generate SAMPLE.HEX.

If TEST.ABS includes object code located in the EEPROM area, then SAMPLE.XE will also be created. The following examples are the same, so the description of the EEPROM code file will be omitted below.

■ Example 2 ■

```
OH66K TEST B:
```

In this example, only the drive name is specified. OH66K will convert TEST.ABS and generate B:TEST.HEX.

■ Example 3 ■

```
OH66K TEST \DATA\
```

In this example, only the directory name is specified. OH66K will convert TEST.ABS and generate \DATA\TEST.HEX. When specifying only a directory name, add a backslash (\) at the end.

■ Example 4 ■

```
OH66K TEST B:\DATA\;
```

In this example, both a drive name and directory name are specified. OH66K will convert TEST.ABS and generate B:\DATA\TEST.HEX.

■ Example 5 ■

```
OH66K TEST SAMPLE.HHH
```

When an extension is added like this, an error will occur.

(3) Options

Options can be specified in any position. When multiple options are specified, it does not matter if spaces are placed between them. The following options are provided.

/S Specifies that the output files are to be in Motorola S2 format. Extensions will be ".S" and ".SE."

If the /S option is not specified, then the output file will be in Intel HEX format.

/D Output debugging information.

For Intel HEX format, public symbols and local symbols will be output as debugging information at the start of a file with the extension ".HEX." For Motorola S2 format, public symbols only will be output as debugging information to a file with the extension ".SYM."

Refer to 7.3.2 "Output Files" regarding Intel HEX format, Motorola S2 format, and the format of debugging information.

■ Example 1 ■

```
OH66K TEST /D;
```

OH66K will generate Intel HEX file TEST.HEX, and output debugging information at the start of this file.

■ Example 2 ■

```
OH66K TEST /S /D;
```

OH66K will output debugging information for public symbols only to TEST.SYM. Object code will be output to TEST.S.

7.2.2 Prompt-Based Conversion

In this method, all input is specified in response to prompts output by OH66K. When OH66K is typed at the DOS prompt, OH66K will display the following prompts one line at a time and wait for user response. OH66K will not display the next prompt until the user responds to the current prompt.

```
INPUT FILE  [.ABS] :
OUTPUT FILE(S) [input.ext] :
```

These prompts correspond to the fields of the command line.

Table 7-1. Prompts And Corresponding Command Line Fields

Prompt	Corresponding Command Line Field
`INPUT FILE  [.ABS] :`	`object_file`
`OUTPUT FILE(S) [input.ext] :`	`hex_file`

If OH66K is not given enough information to execute, then it will prompt the user for input. OH66K will display prompts in the following cases.

If only OH66K is typed at the DOS prompt, then OH66K will not know the input file name. OH66K will display the following prompt.

```
INPUT FILE  [.ABS] :
```

The [.ABS] means that the default extension of the input file is “.ABS.” Specify the input file name at this prompt. If a return is entered without an input file name being specified, then OH66K will display an explanation of its use and then terminate. Options can be specified after the input file name. After the input file name is specified, OH66K will display the following prompt.

```
OUTPUT FILE(S)  [input.ext] :
```

This prompt displays the default output file name in brackets [ ]. The *input* is the base name of the input file, and the *ext* is the appropriate extension for the output file format.

Table 7-2. Prompts and Output File Format

Option	Output File Name Shown At Output File Prompt	Output File Format
None	[*input*.HEX]:	Intel HEX format
/S	[*input*.S]:	Motorola S2 format

input indicates the base name of the input file name.

If the file name displayed at the prompt is acceptable, then press return.

To change it, specify the new file name. When you want to change the output file format at an Intel HEX format prompt, specify the /S option. Refer to Table 7-2 for the relationship between the output file prompt and output file. When output file name specification is complete, OH66K will begin the conversion operation.

If only the input file name is specified on the command line, then OH66K will start by displaying the prompt for the output file name.

● **Using semicolons with prompts**

Even with prompts, specifications can be omitted using semicolons. The meaning is the same as that for command lines.

The next example uses a semicolon at the input file prompt in order to suppress the output file prompt.

```
   INPUT FILE [.ABS] :TEST;
```

The next example uses a semicolon to omit the output file specification.

```
   OUTPUT FILE(S) [TEST.HEX] : ;
```

7.2.3 Redirecting Output Messages

All messages displayed by OH66K to the screen can be redirected to a file using the DOS redirection function. In the following example, messages displayed to the screen will be redirected to the file ERRMES.

```
   OH66K BROKEN; > ERRMES
```

Output can be sent to a printer by using PRN instead of a file name. To prevent messages from being displayed on the screen, you may redirect to NUL.

7.2.4 Termination Code

When OH66K terminates, it returns a termination code to DOS. The termination code indicates the state of OH66K when it terminated. When OH66K terminates normally it returns 0. When OH66K terminates on a fatal error it returns 3. OH66K does not return other termination codes. Refer to Section 7.4, “Error Messages,” regarding fatal errors.

7.3 Files Used With OH66K

7.3.1 Input Files

Two types of files can be input to OH66K.

(1) Object files generated by RAS66K that contain no relocatable information.
(2) Absolute object files generated by RL66K.

The input file must not contain relocatable information, or an error will occur. OH66K will display which software generated the input file to the screen during its conversion operation.

7.3.2 Output Files

OH66K generates two types of HEX files.

Table 7-3. HEX File Types

Option	HEX File	Extension (ROM Code File/EEPROM Code File)
None	Intel HEX file	.HEX / .XE
/S option	S2 format file	.S / .SE

Intel HEX files are files in Intel HEX format. S2 format files are files in Motorola S2 format.

An EEPROM code file is generated only when the object file includes object code located in the EEPROM area. A ROM code file is always generated.

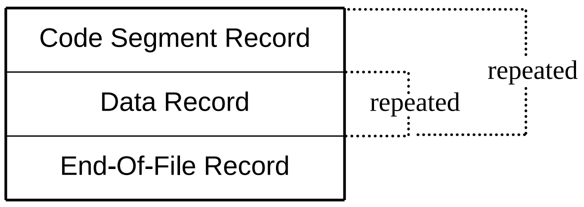
To perform symbolic debugging with an emulator, specify the /D option when invoking OH66K so it will output debugging information. Debugging information is in the same format for all HEX files, but where it is written will differ depending on the output file format. For Intel HEX format, it will be written at the start of the ROM code file with the extension ".HEX." For Motorola S2 format, it will be written in an independent file with the extension ".SYM."

The format each HEX file and debugging information is shown next. First, the file structure is shown, followed by an explanation of each record. The record descriptions start with an output example to show how each field is structured, and then explain the fields.

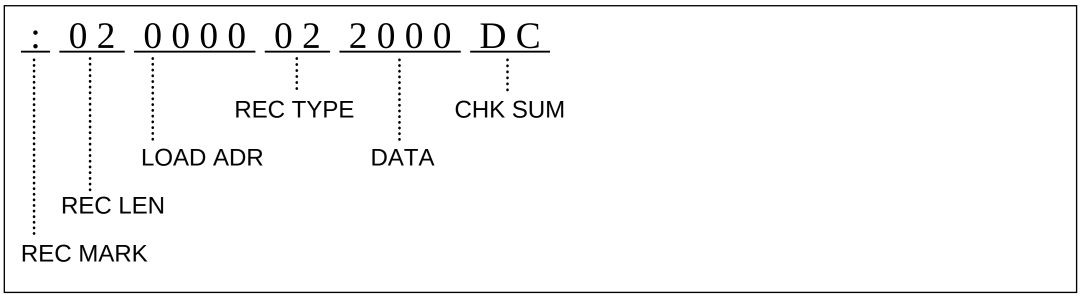
(1) Intel HEX Files

These files are in the Intel HEX format of Intel Corporation.

● **Intel HEX file structure**



● **Code Segment Record**



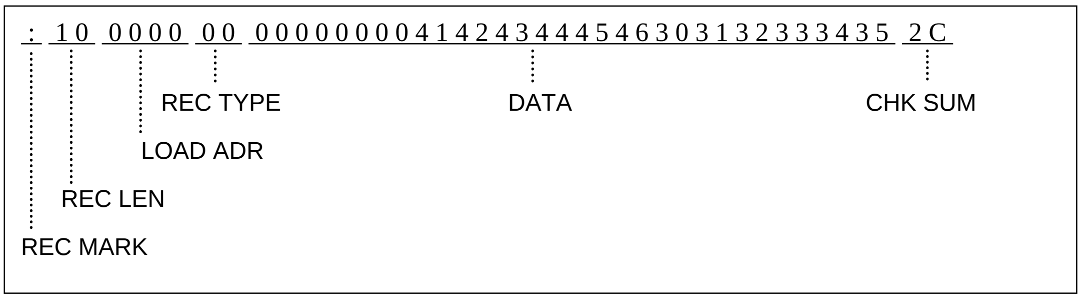
Field	Description
REC MARK	The character “:” (3AH).
REC LEN	Fixed as “02.”
LOAD ADR	Fixed as “0000.”
REC TYP	Fixed as “02.” This indicates a code segment record.
DATA	The physical segment address. This value is only the upper 4 bits, so it can be 0 to F. The rest of the field is fixed to “000.”
CHK SUM	A check sum. This is the two’s complement of the lower 8 bits of the sum of all individual bytes of the REC LEN, LOAD ADR, REC TYP, and DATA fields.

The OLMS-66K Series can have a maximum logical 16 Mbytes, which is a program memory space up to 0FFH:0FFFFH.

However, Intel HEX format data records can only express values up to 0FFFFH. Because of this, physical segment addresses of program memory space are represented with code segment records. The value represented by the DATA field in a code segment record is added as bits 19 to 4 of a load address of data records encountered after the code segment record. This means that only the upper 4 bits will be valid as an actual physical segment address. Therefore, only physical segment addresses 0 to 0FH can be represented in code segment records. If a physical segment exceeds this, or in other words, if a file includes object code located at an address above 10H:0000H, then the file cannot be converted to Intel HEX format. In such cases, convert the file to Motorola S2 format by specifying the /S option.

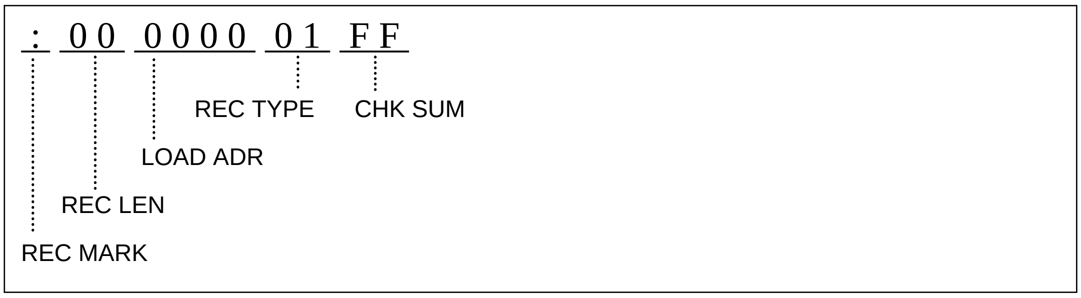
Code segment records are output when the physical segment address is to be changed. If only segment address 0 is used, then no code segment records will be output.

● Data Record



Field	Description
REC MARK	The character ":" (3AH).
REC LEN	The number of bytes of object code stored in the DATA field.
LOAD ADR	The load address of the first object code stored in the DATA field.
REC TYP	Fixed as "00." This indicates a data record.
DATA	Stores object code.
CHK SUM	A check sum. This is calculated the same as for code segment records.

● End-Of-File Records

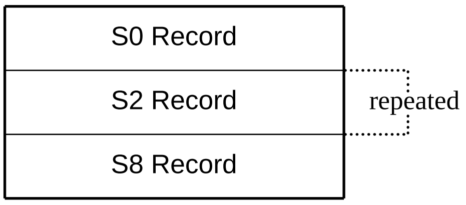


Field	Description
REC MARK	The character ":" (3AH).
REC LEN	Fixed as "00."
LOAD ADR	Fixed as "0000."
REC TYP	Fixed as "01." This indicates an end-of-file record.
CHK SUM	Fixed as "FF."

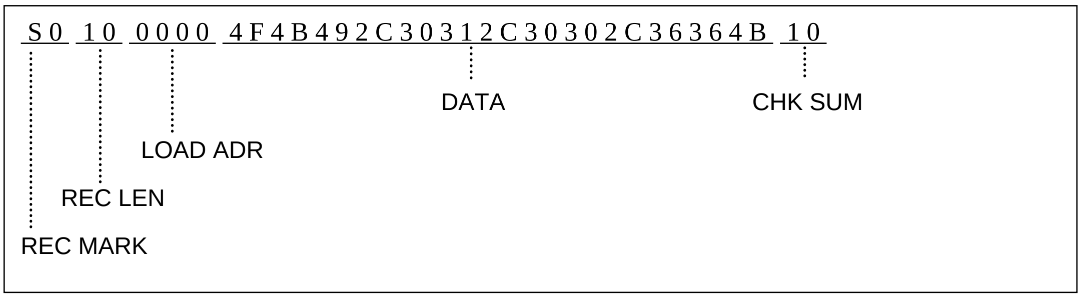
(2) S2 Format File

This file is of Motorola S2 format.

● **S2 format file structure**

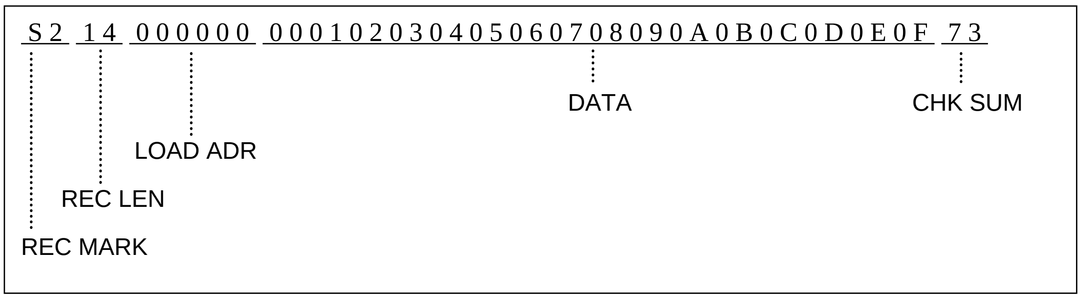


● **S0 Record**



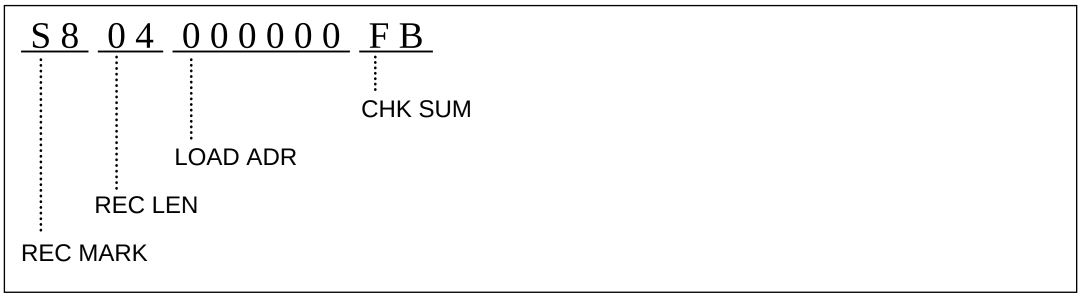
Field	Description
REC MARK	Fixed as “S0.”
REC LEN	The number of bytes (1 byte=2 characters) from the next field until the CHK SUM field. The S0 records generated by OH66K have the following LOAD ADR, DATA, and CHK SUM fields are fixed, so the REC LEN field is fixed as “10.”
LOAD ADR	Fixed as “0000.”
DATA	Fixed as the constant “4F4B492C30312C30302C36364B.” This DATA is used by Oki Electric; it has no relation with the program generated by the user.
CHK SUM	This is the one’s complement of the lower 8 bits of the sum of all individual bytes from the REC LEN field to the field immediately preceding the CHK SUM field.

● **S2 Record**



Field	Description
REC TYP	Fixed as “S2.”
REC LEN	This is calculated in the same way as for S0 records.
LOAD ADR	This is the address where the first byte of the DATA field will be loaded.
DATA	Field where object code is stored.
CHK SUM	This is calculated the same as for S0 records.

● **S8 Record**



Field	Description
REC MARK	Fixed as “S8.”
REC LEN	This is calculated the same as for S0 records. However, the LOAD ADR and CHK SUM fields are fixed in S8 records generated by OH66K, so this field is fixed as “04.”
LOAD ADR	Fixed as “000000.”
CHK SUM	This is calculated the same as for S0 records. However, the LOAD ADR and CHK SUM fields are fixed in S8 records generated by OH66K, so this field is fixed as “FB.”

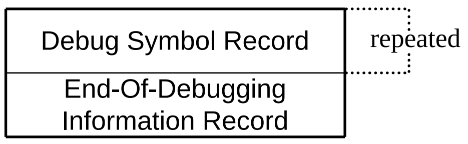
(3) Debugging Information

When the /D option is specified, OH66K will add debugging information to the output file. This information is used for symbolic debugging with an emulator. The debugging information has the same format for any type of HEX file. However, depending on the output file format, the debugging information will be written at different locations.

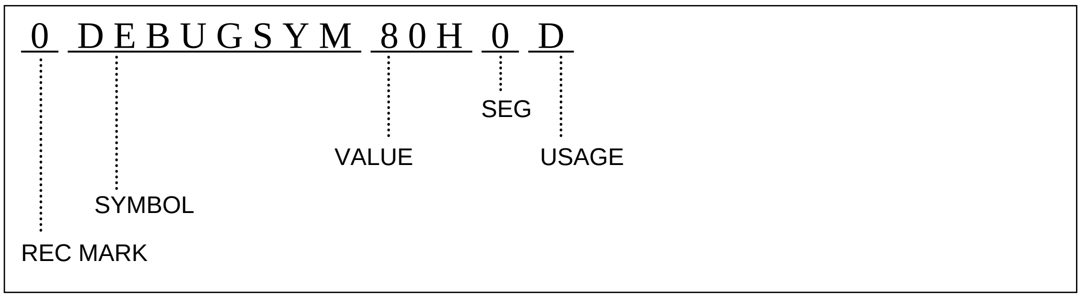
Table 7-4. Debugging Information and Output Location

Output File Format	Necessary Options	Output Location of Debugging Information
Intel HEX format	/D option	Start of .HEX file
Motorola S2 format	/S, /D options	In newly created .SYM file

● Debug information structure

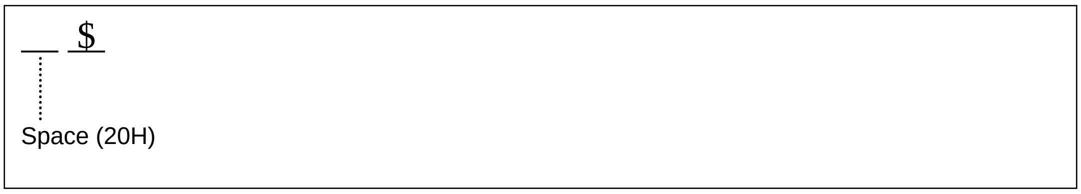


● **Debug Symbol Record**



Field	**Description**
REC MARK	The character “0.” Indicates this record.
SYMBOL	A local symbol or public symbol in the module (only public symbols for S2 format).
VALUE	The value of the symbol, expressed in hexadecimal.
SEG	The physical segment address. Expressed as a hexadecimal number 0 to 0FFH.
USAGE	The usage type of the symbol.
	C: Usage type CODE
	CB: Usage type CBIT
	D: Usage type DATA
	B: Usage type BIT
	ED: Usage type EDATA
	EB: Usage type EBIT
	N: Usage type NUMBER

● **End-Of-Debugging Information Record**



This record indicates the end of debugging information. It consists of one space and one dollar sign ($).

7.3.3 Input And Output File Examples

This section uses some examples to shown how source files are actually converted. First assume that source file TEST.ASM was assembled with the /D option, creating object file TEST.OBJ.

```
RAS66K TEST /D
```

● **Source file TEST.ASM**

```
TYPE (M665XX)

MODEL   LARGE

NUM     EQU     100H

        CSEG
CODE_SYM1:
        DB      0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

        CSEG    AT      1:0000H
CODE_SYM2:
        DW      0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

CBIT_SYM1:      CBIT    3000H:0
CBIT_SYM2:      CBIT    3000H:1

        DSEG
DATA_SYM1:
        DS      1

        DSEG    AT      1:1000H
DATA_SYM2:
        DS      1

        BSEG
BIT_SYM1:
        DBIT    1

        BSEG    AT      2:1000H.0
BIT_SYM2:
        DBIT    1

        ESEG
EDATA_SYM1:
        DB      0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
EDATA_SYM2:
        DW      0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

        EBSEG
EBIT_SYM1:
        DBIT    1
EBIT_SYM2:
        DBIT    1

        PUBLIC  NUM CODE_SYM1 CBIT_SYM1 DATA_SYM1 BIT_SYM1
        PUBLIC  EDATA_SYM1 EBIT_SYM1
        END
```

Next are shown examples of the two types of HEX file conversions of TEST.OBJ using OH66K.

■ Example 1 ■

```
OH66K TEST.OBJ /D;
```

This command will generate Intel HEX files, TEST.HEX and TEST.XE with debugging information.

● Intel HEX file with debugging information (ROM code file) : TEST.HEX

```
0 CBIT_SYM2 18001H 0 CB
0 BIT_SYM2 8000H 2 B
0 EBIT_SYM2 20001H 0 EB
0 CODE_SYM2 0H 1 C
0 DATA_SYM2 1000H 1 D
0 EDATA_SYM2 4010H 0 E
0 CBIT_SYM1 18000H 0 CB
0 BIT_SYM1 1000H 0 B
0 EBIT_SYM1 20000H 0 EB
0 CODE_SYM1 0H 0 C
0 DATA_SYM1 200H 0 D
0 EDATA_SYM1 4000H 0 E
0 NUM 100H 0 N
 $
:10000000000102030405060708090A0B0C0D0E0F78
:020000021000EC
:1000000000000100020003000400050006000700D4
:10001000080009000A000B000C000D000E000F0084
:00000001FF
```

● Intel HEX file (EEPROM code file) : TEST.XE

```
:10400000000102030405060708090A0B0C0D0E0F38
:104010000000010002000300040005000600070084
:10402000080009000A000B000C000D000E000F0034
:00000001FF
```

■ Example 2 ■

```
OH66K TEST.OBJ /S /D;
```

This command will generate S2 format files TEST.S and TEST.SE, as well as a debugging information file TEST.SYM for only public symbols.

● S2 format file (ROM code file) : TEST.S

```
S01000004F4B492C30312C30302C36364B10
S214000000000102030405060708090A0B0C0D0E0F73
S21401000000000100020003000400050006000700CE
S214010010080009000A000B000C000D000E000F007E
S804000000FB
```

● S2 format file (EEPROM code file) : TEST.SE

```
S01000004F4B492C30312C30302C36364B10
S214004000000102030405060708090A0B0C0D0E0F33
S2140040100000010002000300040005000600070007F
S214004020080009000A000B000C000D000E000F002F
S804000000FB
```

● Debugging information file (public symbols only) : TEST.SYM

```
0 CBIT_SYM1 18000H 0 CB
0 BIT_SYM1 1000H 0 B
0 EBIT_SYM1 20000H 0 EB
0 CODE_SYM1 0H 0 C
0 DATA_SYM1 200H 0 D
0 EDATA_SYM1 4000H 0 E
0 NUM 100H 0 N
 $
```

7.3.4 Temporary Files

During the conversion operation, OH66K uses up to three temporary files: “$_$,” “$__$,” and “$___$.” The number of temporary files used changes depending on the output file format.

OH66K reads data from the input file. It then outputs converted data to temporary files. After conversion completes normally, temporary files are written to output files. OH66K erases the temporary files when it terminates.

Temporary files are created in the current directory. Do not place files with the same names in the current directory.

7.4 Error Messages

All errors generated by OH66K are fatal errors. After an error, the conversion operation will halt, and no output file will be created.

7.4.1 Error Message Format

Error message formats can be classified in two types. The first is for errors during conversion, and the second is for other errors.

- Format for errors during conversion

```
Error : error_messsage
File Offset : hhhhhhhhH (dddddddd)
```

- Format for other errors

```
Error : error_message
```

The symbols used in the above format descriptions are explained below.

Symbol Used In Manual	Screen Display
`hhhhhhhhH`	File offset when error occurred (in hexadecimal).
`(dddddddd)`	The offset in decimal.
`error_message`	A message indicated error status.

Error messages and their explanations are given below. The error messages are listed in alphabetical order.

7.4.2 Fatal Error Messages

Bad syntax in command line

The command line specification is improper.

Checksum failure

A record read from the input file has an incorrect checksum. The input file format is improper. Remake the input file.

Command option duplicated

The same option is specified twice on the command line

Extension not allowed in output file name

An extension cannot be specified in the output file name. Drive name, directory name, and base name can be specified. To change the extension, use the DOS REN command after the HEX file has been created.

File not absolute

There is relocatable information in the input file. If you have specified an .OBJ file generated by RAS66K, then create an .ABS file using RL66K and specify that file.

File offset error

The file offset of a record read from the input file could not be obtained.

File read error

An error occurred when the input file was read.

File remove error

An error occurred when a temporary file was erased.

I/O error

An error occurred when the file was closed.

Illegal command option

An improper option was specified on the command line.

Input file not specified

The input file specification is improper. Something other than a file name was specified in the field for the input file name.

Insufficient disk space

An error occurred when OH66K tried to close files upon termination.

Insufficient memory

There is not enough memory for the conversion operation.

Invalid family ID

The input file was not created for OLMS-66K. OH66K can convert only absolute object files generated by RAS66K or RL66K.

Invalid object module

The input file is improper.

Invalid record type

There is a record type not recognized by OH66K in the input file. Check if the RAS66K or RL66K version that created the input file was provided together with OH66K. Use software provided in the same package.

Invalid segment type

A debugging information symbol in the input file has a segment type that is not supported.

Invalid target

The input file was created as a general-purpose module.

Invalid version number

The version number of the software that generated the object file is abnormal.

Unable to convert Intel HEX format

The input file includes object code that exceeds the range 0H:0H to 0FH:0FFFFH. This file cannot be converted to Intel HEX format. Convert it to Motorola S2 format by specifying the /S option.

Unable to open *file_name*

The file *file_name* could not be opened.

Unable to open OH66K temporary file

One of the temporary files “$_$”, “$__$” or “$___$” is read-only. Remove the read-only attribute of the file using the DOS ATTRIB command.

Chapter 8

Absolute Print File Generation

This chapter explains how to generate absolute print files.

An absolute print file is a print file that does not include unresolved machine code information or address information. All of its information is resolved.

Make use of the absolute print file when splitting a program into multiple modules or when debugging a program that uses relocatable segments.

8.1 Introduction

An absolute print file is a print file that does not include unresolved machine code information or address information. All of its information is resolved.

When splitting a program into multiple modules or debugging a program that uses relocatable segments, a normal print file will include unresolved machine code or address information. To debug using this print file requires that you simultaneously look at the symbol information included in the map file output by RL66K, which is extremely tedious.

The MAC66K Assembler Package supports the generation of absolute print files that do not include unresolved information as a solution to this problem. It provides functions for re-assembling programs that have been assembled once based on their link information.

8.2 Absolute Print File Generation Procedure

As an example, this section will explain the procedure for making absolute print files for a program configured as three files: FOO1.ASM, FOO2.ASM, and FOO3.ASM.

First, assemble the files as usual.

```
RAS66K FOO1
RAS66K FOO2
RAS66K FOO3
```

Then link the three object files. This time specify the /A option.

```
RL66K FOO1 FOO2 FOO3 /A;
```

RL66K will generate a file called FOO1.ABL. This file includes resolved machine code and absolute addresses of symbols. It is called an ABL file.

Next, assemble the source files again. This time specify /AFOO1 as an assembler option.

```
RAS66K FOO1 /AFOO1
RAS66K FOO2 /AFOO2
RAS66K FOO3 /AFOO3
```

This re-assembly will generate three files with the extension ".APR."

```
FOO1.APR
FOO2.APR
FOO3.APR
```

These files are absolute print files.

This process is shown in the figure below.

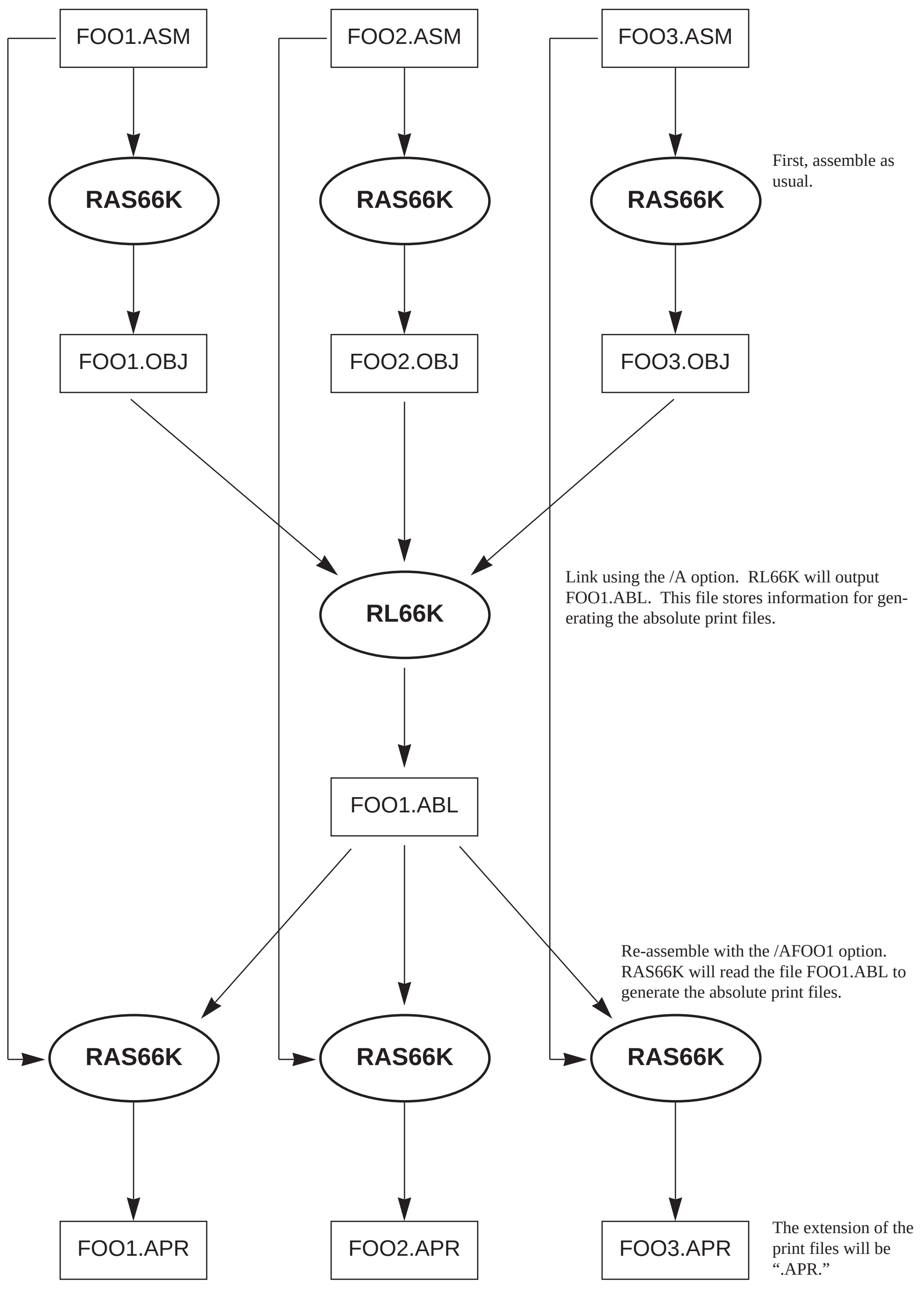

FOO1.ASM
FOO2.ASM
FOO3.ASM
RAS66K
RAS66K
RAS66K
First, assemble as usual.
FOO1.OBJ
FOO2.OBJ
FOO3.OBJ
RL66K
Link using the /A option. RL66K will output FOO1.ABL. This file stores information for generating the absolute print files.
FOO1.ABL
Re-assemble with the /AFOO1 option. RAS66K will read the file FOO1.ABL to generate the absolute print files.
RAS66K
RAS66K
RAS66K
FOO1.APR
FOO2.APR
FOO3.APR
The extension of the print files will be ".APR."

Figure 8-1. Flow Of Absolute Print File Generation Process

8.3 Link Processing For Absolute Print File Generation

To generate absolute print files, specify the /A option when linking. When linking is performed with the /A option. RL66K will generated a file called the ABL file, which contains resolved machine code and absolute addresses of symbols.

The proper format of the RL66K /A option is as below.

```
/A [(abl_file)]
```

The *abl_file* specifies the name of the ABL file to be generated. If *abl_file* is omitted, then the ABL file name will be the name of the absolute object file (ABS) with an extension ".ABL." If just the extension of *abl_file* is omitted, then the extension will become ".ABL."

■ **Example 1** ■

```
RL66K FILE1 FILE2 FILE3 /A;
```

The *abl_file* is omitted, so the ABL file name will be FILE1.ABL.

■ **Example 2** ■

```
RL66K FILE1 FILE2 FILE3 /A(PRNDATA);
```

The extension of *abl_file* is omitted, so the ABL file name will be PRNDATA.ABL.

■ **Example 3** ■

```
RL66K FILE1 FILE2 FILE3 /A(PRNDATA.DAT);
```

A file name is specified for *abl_file*, so the ABL file name will be PRNDATA.DAT.

8.4 Re-Assembly For Absolute Print File Generation

To generate absolute print files, add the RAS66K /A option when re-assembling

The proper format of the RAS66K /A option is as below.

```
/A [abl_file]
```

The *abl_file* specifies the name of the ABL file generated by RL66K. If *abl_file* is omitted, then the ABL file name will be the name of the source file with an extension ".ABL." If just the extension of *abl_file* is omitted, then the extension will become ".ABL."

When RAS66K is invoked with the /A option, it reads the ABL file to generate an absolute print file.

The absolute print file name will be the name of the source file with the extension ".APR." The absolute print file name can be changed by using the /PR option.

A comparison of re-assembly processing against normal assembly processing gives the following differences in operation.

(1) No object file is generated. The /O option and OBJ directive will be invalid.

(2) No processing is performed for C source level debugging information. The /CC option is invalid.

(3) No EXTRN declaration file is generated. The /X option is invalid.

(4) The absolute print file will be generated even if the /NPR option or NOPRN directive is specified.

The invalid options and directives are simply ignored. It does not matter if they are still specified.

Certain option must be specified identically when normal assembly and re-assembly are invoked. These are listed below.

(1) /M*x*	Memory model (nX-8/500 only). It is recommended that the memory model be specified using the MODEL directive.
(2) /CD, /NCD	Case sensitivity of symbols.
(3) /I*include_path*	Include file path specification.

When generating an absolute print file, it is recommended that you re-assemble using the exact options from invoking assembly the first time, with the /A option added.

For example, assume the options specified during normal assembly are these.

```
RAS66K FOO1 /D /S /R /PW120 /PL60 /T4 /V /IHEADER
```

During re-assembly, add the /A option.

```
RAS66K FOO1 /D /S /R /PW120 /PL60 /T4 /V /IHEADER /AFOO1
```

8.5 Re-Assembly Errors

A program that did not generate errors during normal assembly may still generate errors or warnings during re-assembly with the /A option. This is because the normal assembly process does not perform error checking on operands with unresolved addresses, while the re-assembly process performs error checking on all operands.

Consider the following example.

```
EXTRN    DATA:DATA_TBL
         MOV     X1,DATA_TBL ···················· (1)
```

The instruction statement (1) will not cause an error in the normal assembly process. However, this does not mean that there is no error. It is that error checking cannot be performed during normal assembly because the address of DATA_TBL is not clear.

Assume for the moment that the address of DATA_TBL is odd, such as 1001H. This access is a word boundary error. RL66K will display the following message when linking. (The source file name is assumed to be "foo1.asm" and the address of the instruction causing the warning to be 200H.)

```
Warning : cannot access to high byte ,  0200/(absolute)/foo1
```

The error's existence is confirmed. However, to find the actual position of errors in the source program, the programmer must search for all addresses included in RL66K error messages. For a large program, this can be very tedious work.

The generation of an absolute print file can be extremely convenient in these cases. Try to re-assemble the above source program with the /A option. This time, RAS66K will display an error message that has the same contents of that displayed by RL66K.

```
foo1.asm (215):215:Warning 28: cannot access to high byte
```

This informs the program of the precise position (line number) of the error.

Thus, the absolute print file generation function can also be used for purposes of learning the precise position in the source program of addressing-related errors and warnings generated during linking.

■ Attention ■

All link errors will not necessarily make re-assembly invalid. The link errors that will not affect re-assembly are limited to the following.

(1) All warnings.
(2) Other errors:
    VCAL address must be even number
    Out of range: *min* to *max*
    Out of *area_name* area

A correct absolute print file cannot be obtained if re-assembly is performed after any other errors are generated. Instead a fatal error will occur during re-assembly.

8.6 If Fatal Error 11 Occurs

The following fatal error message may be displayed when performing re-assembly.

```
Fatal Error 11 : illegal reading binary file
ABL file : error_message
```

The cause of this error is nearly always a problem in the contents of the ABL file. If this error occurs, then first verify the following items.

- Did the first assembly have no errors (excluding warnings)?
- Does the re-assembly match the first assembly in:

  memory model specification (/M*x* option or MODEL directive)?
  case sensitivity of symbols (/CD, /NCD option)?
  include path specification (/I*include_path* option)?

- Did linking have no fatal errors (excluding addressing errors)?
- Was the /A option specified when linking?

If you have verified these items and the error still occurs, then please contact Oki Electric.

For details on ABL file error messages, refer to Section 4.15.2.1, “Fatal Error Messages.”

Appendices

Appendix A. List Of Directives

RAS66K directives are listed below.

Directive	Syntax Function
TYPE	`TYPE (dcl_name)` Specify a DCL file.
MODEL	`MODEL memory_model` Specify a type of memory model.
COMMON	`COMMON bcb_value` Specify the COMMON area.
WINDOW	`WINDOW start_address , end_address` Specify the ROM window area.
EQU	`symbol EQU simple_expression` Define a local symbol.
SET	`symbol SET simple_expression` Define a local symbol (redefinition possible).
CODE	`symbol CODE simple_expression` Define a local symbol representing a byte address in CODE address space.
CBIT	`symbol CBTI simple_expression` Define a local symbol representing a bit address in CODE address space.
DATA	`symbol DATA simple_expression` Define a local symbol representing an address in DATA address space.
BIT	`symbol BIT simple_expression` Define a local symbol representing an address in BIT address space.
EDATA	`symbol EDATA simple_expression` Define a local symbol representing an address in EDATA address space.
EBIT	`symbol EBIT simple_expression` Define a local symbol representing an address in EBIT address space.

CSEG	`CSEG [# physical_segment_address] [AT start_address]`
	`CSEG [AT start_address] [# physical_segment_address]`
	Define an absolute CODE segment.

DSEG	`DSEG [{#physical_segment_address | COMMON}] [AT start_address]`
	`DSEG [AT start_address] [{#physical_segment_address | COMMON}]`
	Define an absolute DATA segment.

BSEG	`BSEG [{#physical_segment_address | COMMON}] [AT start_address]`
	`BSEG [AT start_address] [{#physical_segment_address | COMMON}]`
	Define an absolute BIT segment.

Directive	Syntax Function
ESEG	`ESEG [AT start_address]` Define an absolute EDATA segment.
EBSEG	`EBSEG [AT start_address]` Define an absolute EBIT segment.
SEGMENT	`segment_symbol SEGMENT segment_type [boundary_attr] [relocation_attr]` Define a segment symbol.
STACKSEG	`STACKSEG stack_size` Define a stack segment.
RSEG	`RSEG segment_symbol` Define a relocatable segment.
GROUP	`GROUP segment_symbol [segment_symbol...] [#physical_segment_address]` Define a segment group.
ORG	`ORG address` Set the location counter.
DS	`[label:] DS size` Allocate memory in a CODE segment, DATA segment, or EDATA segment.
DBIT	`[label:] DBIT size` Allocate memory in a BIT segment or EBIT segment.
DB	`[label:] DB {expression |string_constant} [,{expression |string_constant}]…` Initialize program memory in bytes.
DW	`[label:] DW expression [,expression]…` Initialize program memory in words.
PUBLIC	`PUBLIC symbol [symbol]…` Declare a public symbol.
EXTRN	`EXTRN usage_type [attribute] : symbol [symbol]… [usage_type [attribute] : symbol [symbol]… ]…` Declare an external symbol.
COMM	`communal_symbol COMM segment_type size [relocation_attr]` Declare a communal symbol.
USING	`USING register_name status` Check program states.
CHK	`CHK` Check flag attributes of branch instructions.

Directive	Syntax Function
INCLUDE	`INCLUDE (include_file)` Use an include file.
END	`END` End the program.
NAME	`NAME (module_name)` Set a module's name.

PRBANK
NOPRBANK
	`PRBANK bank_no [,bank_no…]`
	`NOPRBANK`
	Declare pointing register bank to be used.

LRBANK
NOLRBANK
	`LRBANK bank_no [,bank_no…]`
	`NOLRBANK`
	Declare local register bank to be used.

IF
IFDEF
IFNDEF
ELSE
ENDIF
	`IFxxx conditional operand` (IFxxx is one of IF, IFDEF, IFNDEF)
	`      true_conditional_body`
	`[ELSE`
	`      false_conditional_body]`
	`ENDIF`
	Conditional assembly.

DEFINE	`DEFINE symbol "macro_body"` Define a macro.

CFILE
CFUNCTION
CLINE
	`CFILE expression`
	`CFUNCTION expression`
	`CLINE expression`
	C source level debugging information.

GJMP	`GJMP symbol` Optimize a jump instruction.
GCAL	`GCAL symbol` Optimize a call instruction.

PRN
NOPRN
	`PRN [(print_file)]`
	`NOPRN`
	Control print file output.

Directive	Syntax Function
PAGE	`PAGE` Force a page break.
PAGE	`PAGE [page_length] [,page_width]` Set lines per page and characters per line.
TITLE	`TITLE "character_string"` Set the print file title.
DATE	`DATE "character_string"` Set the print file data.

LIST
NOLIST
	`LIST`
	`NOLIST`
	Control assembly list output.

SYM
NOSYM
	`SYM`
	`NOSYM`
	Control symbol list output.

REF
NOREF
	`REF`
	`NOREF`
	Control cross-reference list output.

TAB	`TAB [tab_width]` Replace tab codes.

OBJ
NOOBJ
	`OBJ [(object_file)]`
	`NOOBJ`
	Control object output.

DEBUG
NODEBUG
	`DEBUG`
	`NODEBUG`
	Control assembly level debugging information output.

ERR
NOERR
	`ERR [(error_file)]`
	`NOERR`
	Control error message output.

Appendix B. List Of Reserved Words

RAS66K reserved words are listed below in alphabetic order. This list also shows the use of each reserved word. For reserved words restricted to a CPU core, the name of the CPU core is shown in the CPU type column. If there is no restriction, then the column will be blank.

	Reserved Word	Use	CPU Type
A	A	Register name	
	ACAL	Instruction	nX-8/500
	ADC	Instruction	
	ADCB	Instruction	
	ADD	Instruction	
	ADDB	Instruction	
	ADP	Pointing register address	
	AER0	Local register address	nX-8/500
	AER1	Local register address	nX-8/500
	AER2	Local register address	nX-8/500
	AER3	Local register address	nX-8/500
	AND	Instruction	
	ANDB	Instruction	
	ANY	Directive operand	
	AR0	Local register address	nX-8/500
	AR1	Local register address	nX-8/500
	AR2	Local register address	nX-8/500
	AR3	Local register address	nX-8/500
	AR4	Local register address	nX-8/500
	AR5	Local register address	nX-8/500
	AR6	Local register address	nX-8/500
	AR7	Local register address	nX-8/500
	AT	Directive operand	
	AUSP	Pointing register address	
	AX1	Pointing register address	
	AX2	Pointing register address	
B	BAND	Instruction	nX-8/500
	BANDN	Instruction	nX-8/500
	BIT	Directive	
	BOR	Instruction	nX-8/500

	Reserved Word	Use	CPU Type
B	BORN	Instruction	nX-8/500
	BPOS	Operator	
	BRK	Instruction	
	BSEG	Directive	
	BXCHG	Instruction	nX-8/500
	BXOR	Instruction	nX-8/500
	BXORN	Instruction	nX-8/500
	BYTE	Directive operand	
C	C	Register name	
	CAL	Instruction	
	CBIT	Directive	
	CFILE	Directive	
	CFUNCTION	Directive	
	CHK	Directive	
	CLINE	Directive	
	CLR	Instruction	
	CLRB	Instruction	
	CMP	Instruction	
	CMPB	Instruction	
	CMPC	Instruction	
	CMPCB	Instruction	
	CODE	Directive	
	COMM	Directive	
	COMMON	Directive	
	COMPACT	Directive operand	nX-8/500
	CPL	Instruction	nX-8/500
	CR	Register set	nX-8/500
	CSEG	Directive	
	CY	JC instruction branch condition	
D	DAA	Instruction	
	DAS	Instruction	
	DATA	Directive	
	DATE	Directive	
	DB	Directive	
	DBIT	Directive	
	DEBUG	Directive	

	Reserved Word	Use	CPU Type
D	DEC	Instruction	
	DECB	Instruction	
	DEFINE	Directive	
	DI	Instruction	nX-8/500
	DIR	Addressing specifier	nX-8/500
	DIV	Instruction	
	DIVB	Instruction	
	DIVQ	Instruction	nX-8/500
	DJNZ	Instruction	nX-8/500
	DP	Register name	
	DPL	Register name	nX-8/500
	DS	Directive	
	DSEG	Directive	
	DSREG	Directive operand	
	DUAL	Directive operand	nX-8/500
	DW	Directive	
	DYNAMIC	Directive operand	
E	EBIT	Directive	
	EBSEG	Directive	
	EDATA	Directive	
	EI	Instruction	nX-8/500
	ELSE	Directive	
	END	Directive	
	ENDIF	Directive	
	EQ	JC instruction branch condition	
	EQU	Directive	
	ER	Register set	nX-8/500
	ER0	Register name	
	ER1	Register name	
	ER2	Register name	
	ER3	Register name	
	ERR	Directive	
	ESEG	Directive	
	EX	Instruction	nX-8/300
	EXTND	Instruction	
	EXTRN	Directive	

	Reserved Word	Use	CPU Type
F	FCAL	Instruction	nX-8/500
	FILL	Instruction	nX-8/500
	FILLB	Instruction	nX-8/500
	FIX	Addressing specifier	nX-8/500
	FJ	Instruction	nX-8/500
	FRT	Instruction	nX-8/500
G	GCAL	Directive	
	GE	JC instruction branch condition	
	GES	JC instruction branch condition	nX-8/500
	GJMP	Directive	
	GROUP	Directive	
	GT	JC instruction branch condition	
	GTS	JC instruction branch condition	nX-8/500
H	HIGH	Operator	
I	IF	Directive	
	IFDEF	Directive	
	IFNDEF	Directive	
	INACAL	Directive operand	nX-8/500
	INC	Instruction	
	INCB	Instruction	
	INCLUDE	Directive	
	INPAGE	Directive operand	
J	J	Instruction	
	JBR	Instruction	
	JBRS	Instruction	nX-8/500
	JBS	Instruction	
	JBSR	Instruction	nX-8/500
	JC	Instruction	
	JCY	Instruction	
	JEQ	Instruction	
	JGE	Instruction	
	JGES	Instruction	nX-8/500
	JGT	Instruction	
	JGTS	Instruction	nX-8/500
	JLE	Instruction	
	JLES	Instruction	nX-8/500

	Reserved Word	Use	CPU Type
J	JLT	Instruction	
	JLTS	Instruction	nX-8/500
	JNC	Instruction	
	JNE	Instruction	
	JNS	Instruction	nX-8/500
	JNV	Instruction	nX-8/500
	JNZ	Instruction	
	JOV	Instruction	nX-8/500
	JPS	Instruction	nX-8/500
	JRNZ	Instruction	
	JZ	Instruction	
L	L	Instruction	
	LARGE	Directive operand	nX-8/500
	LB	Instruction	
	LC	Instruction	
	LCB	Instruction	
	LE	JC instruction branch condition	
	LES	JC instruction branch condition	nX-8/500
	LIST	Directive	
	LOW	Operator	
	LRB	Register name	
	LRBANK	Directive	nX-8/500
	LREG	Operator	nX-8/500
	LT	JC instruction branch condition	
	LTS	JC instruction branch condition	nX-8/500
M	MAC	Instruction	nX-8/500
	MB	Instruction	
	MBR	Instruction	
	MEDIUM	Directive operand	nX-8/500
	MID	Operator	
	MODEL	Directive	nX-8/500
	MOV	Instruction	
	MOVB	Instruction	
	MUL	Instruction	
	MULB	Instruction	

	Reserved Word	Use	CPU Type
N	NAME	Directive	
	NC	JC instruction branch condition	
	NE	JC instruction branch condition	
	NEG	Instruction	nX-8/500
	NEGB	Instruction	nX-8/500
	NODEBUG	Directive	
	NOERR	Directive	
	NOLIST	Directive	
	NOLRBANK	Directive	nX-8/500
	NOOBJ	Directive	
	NOP	Instruction	
	NOPRBANK	Directive	
	NOPRN	Directive	
	NOREF	Directive	
	NOSYM	Directive	
	NS	JC instruction branch condition	nX-8/500
	NULL	Addressing	nX-8/300
	NUMBER	Directive	
	NV	JC instruction branch condition	nX-8/500
	NZ	JC instruction branch condition	
O	OBJ	Directive	
	OCT	Directive operand	
	OFF	Addressing specifier	
	OFFSET	Operator	
	OPRT	Directive operand	nX-8/300
	OR	Instruction	
	ORB	Instruction	
	ORG	Directive	
	OV	JC instruction branch condition	nX-8/500
P	PAGE	Directive	
	POPS	Instruction	
	POPU	Instruction	nX-8/300, 500
	POPUB	Instruction	nX-8/300
	PR	Register set	
	PRBANK	Directive	
	PREG	Directive operand	

	Reserved Word	Use	CPU Type
P	PRN	Directive	
	PS	JC instruction branch condition	nX-8/500
	PSW	Register name	
	PSWH	Register name	
	PSWL	Register name	
	PUBLIC	Directive	
	PUSHS	Instruction	
	PUSHU	Instruction	nX-8/300, 500
	PUSHUB	Instruction	nX-8/300
R	R0	Register name	
	R1	Register name	
	R2	Register name	
	R3	Register name	
	R4	Register name	
	R5	Register name	
	R6	Register name	
	R7	Register name	
	RB	Instruction	
	RBR	Instruction	
	RC	Instruction	
	RDD	Instruction	nX-8/500
	REF	Directive	
	RLNC	Instruction	nX-8/500
	RLNCB	Instruction	nX-8/500
	ROL	Instruction	
	ROLB	Instruction	
	ROR	Instruction	
	RORB	Instruction	
	RRNC	Instruction	nX-8/500
	RRNCB	Instruction	nX-8/500
	RSEG	Directive	
	RT	Instruction	
	RTI	Instruction	

	Reserved Word	Use	CPU Type
S	SB	Instruction	
	SBA	Addressing specifier	nX-8/500
	SBAFIX	Addressing specifier	nX-8/500
	SBAOFF	Addressing specifier	nX-8/500
	SBC	Instruction	
	SBCB	Instruction	
	SBR	Instruction	
	SC	Instruction	
	SCAL	Instruction	
	SCMP	Instruction	nX-8/300
	SCMPEQ	Instruction	nX-8/500
	SCMPEQB	Instruction	nX-8/500
	SCMPNE	Instruction	nX-8/500
	SCMPNEB	Instruction	nX-8/500
	SDD	Instruction	nX-8/500
	SEG	Operator	
	SEGMENT	Directive	
	SET	Directive	
	SFR	Addressing specifier	nX-8/500
	SIZE	Operator	
	SJ	Instruction	
	SLL	Instruction	
	SLLB	Instruction	
	SMALL	Directive operand	nX-8/500
	SMOV	Instruction	nX-8/500
	SMOVB	Instruction	nX-8/500
	SMOVD	Instruction	nX-8/300
	SMOVI	Instruction	nX-8/300
	SQR	Instruction	nX-8/500
	SQRB	Instruction	nX-8/500
	SRA	Instruction	
	SRAB	Instruction	
	SRL	Instruction	
	SRLB	Instruction	
	SS	Instruction	nX-8/300
	SSP	Register name	

	Reserved Word	Use	CPU Type
S	ST	Instruction	
	STACK	Directive operand	nX-8/300
	STACKSEG	Directive	
	STB	Instruction	
	SUB	Instruction	
	SUBB	Instruction	
	SWAP	Instruction	
	SWAPB	Instruction	
	SWI	Instruction	nX-8/500
	SYM	Directive	
T	TAB	Directive	
	TBR	Instruction	
	TITLE	Directive	
	TJNZ	Instruction	nX-8/500
	TJNZB	Instruction	nX-8/500
	TJZ	Instruction	nX-8/500
	TJZB	Instruction	nX-8/500
	TRNS	Instruction	nX-8/300
	TSREG	Directive operand	nX-8/500
	TYPE	Directive	
U	UNIT	Directive operand	
	USING	Directive	
	USP	Register name	
	USPL	Register name	nX-8/500
V	VCAL	Instruction	
W	WINDOW	Directive	nX-8/500
	WINDOWALL	Directive operand	nX-8/500
	WORD	Directive operand	

	Reserved Word	Use	CPU Type
X	X1	Register name	
	X1L	Register name	
	X2	Register name	
	X2L	Register name	
	XCHG	Instruction	
	XCHGB	Instruction	
	XNBL	Instruction	
	XOR	Instruction	
	XORB	Instruction	
Z	ZERO	Directive operand	nX-8/100 to 400
	ZF	JC instruction branch condition	