LAPIS SEMICONDUCTOR

MACU8 Assembler Package User's Manual

Program Development Support Software

Relocatable Assembler	RASU8
Linker	RLU8
Librarian	LIBU8
Object Converter	OHU8

7th Edition
ISSUE DATE: Aug. 2012

NOTICE

No copying or reproduction of this document, in part or in whole, is permitted without the consent of LAPIS Semiconductor Co., Ltd.

The content specified herein is subject to change for improvement without notice.

The content specified herein is for the purpose of introducing LAPIS Semiconductor's products (hereinafter "Products"). If you wish to use any such Product, please be sure to refer to the specifications, which can be obtained from LAPIS Semiconductor upon request.

Examples of application circuits, circuit constants and any other information contained herein illustrate the standard usage and operations of the Products. The peripheral conditions must be taken into account when designing circuits for mass production.

Great care was taken in ensuring the accuracy of the information specified in this document. However, should you incur any damage arising from any inaccuracy or misprint of such information, LAPIS Semiconductor shall bear no responsibility for such damage.

The technical information specified herein is intended only to show the typical functions of and examples of application circuits for the Products. LAPIS Semiconductor does not grant you, explicitly or implicitly, any license to use or exercise intellectual property or other rights held by LAPIS Semiconductor and other parties. LAPIS Semiconductor shall bear no responsibility whatsoever for any dispute arising from the use of such technical information.

The Products specified in this document are intended to be used with general-use electronic equipment or devices (such as audio visual equipment, office-automation equipment, communication devices, electronic appliances and amusement devices).

The Products specified in this document are not designed to be radiation tolerant.

While LAPIS Semiconductor always makes efforts to enhance the quality and reliability of its Products, a Product may fail or malfunction for a variety of reasons.

Please be sure to implement in your equipment using the Products safety measures to guard against the possibility of physical injury, fire or any other damage caused in the event of the failure of any Product, such as derating, redundancy, fire control and fail-safe designs. LAPIS Semiconductor shall bear no responsibility whatsoever for your use of any Product outside of the prescribed scope or not in accordance with the instruction manual.

The Products are not designed or manufactured to be used with any equipment, device or system which requires an extremely high level of reliability the failure or malfunction of which may result in a direct threat to human life or create a risk of human injury (such as a medical instrument, transportation equipment, aerospace machinery, nuclear-reactor controller, fuel-controller or other safety device). LAPIS Semiconductor shall bear no responsibility in any way for use of any of the Products for the above special purposes. If a Product is intended to be used for any such special purpose, please contact a ROHM sales representative before purchasing.

If you intend to export or ship overseas any Product or technology specified herein that may be controlled under the Foreign Exchange and the Foreign Trade Law, you will be required to obtain a license or permit under the Law.

Windows is a registered trademark of Microsoft Corporation (USA) in USA and other countries, and other product names and company names are trademarks or registered trademarks.

Contents

Before You Begin

MACU8 Assembler Package

The MACU8 assembler package is software for developing assembly language programs for microcontrollers based on the nX-U8 8-bit RISC processor core.

The package contains the following software.

RASU8 relocatable assembler

RASU8 generates object files from assembly language source code files. These object files contain object code corresponding to the source code file contents as well as information necessary for linking and debugging. The assembler also generates print files and error files.

RLU8 linker

RLU8 merges one or more object modules into a single absolute object file. It also generates a map file listing address assignments for segments and public symbols.

LIBU8 librarian

LIBU8 creates and manages library files, which merge multiple object files for use by RLU8.

OHU8 object converter

OHU8 converts absolute object files created by RLU8 or RASU8 into Intel HEX format or Motorola S2 format.

System Requirements

The software in the MACU8 assembler package requires the following environment to operate.

Hardware: Pentium-based IBM-PC/AT or equivalent clone

Operating system: Windows2000 and WindowsXP or later

The software in the MACU8 assembler package consists of command line tools that all run from the Windows command prompt.

About This Manual

This manual describes the software in the MACU8 assembler package. It assumes that the user is thoroughly grounded in assembly language and thus capable of creating and editing assembly language source code files.

The following is an overview each Chapter.

Before You Begin

This Chapter.

1. Introduction

This Section gives a functional overview of the tools in the MACU8 assembler package and their roles in overall program development.

2. Programming Basics

This Section summarizes the basic knowledge required to develop programs with the MACU8 assembler package.

3. Program Components

This Section gives the elements making up a program.

4. Addressing and Instruction Types

This Section describes the addressing types available to the assembly language programmer and the limits on their combination with instructions.

5. Detailed Directive Descriptions

This Section describes the directives available with the MACU8 assembler package.

6. RASU8 Relocatable Assembler

This Section describes the operating procedures for the RASU8 relocatable assembler.

7. RLU8 Linker

This Section describes the operating procedures for the RLU8 linker.

8. LIBU8 Librarian

This Section describes the operating procedures for the LIBU8 librarian.

9. OHU8 Object Converter

This Section describes the operating procedures for the OHU8 object converter.

10. Using Overlays

This Section describes the procedures for using overlays.

11. Absolute Listing Files

This Section describes absolute listings and the procedures for generating such files.

Appendices

These list directives and reserved words.

Notation

The following Table lists the symbols and other notational conventions that this manual uses to simplify descriptions.

Notation	Description
`SAMPLE`	This monospaced typeface indicates such things as screen messages, sample command line input, and the contents of sample list files.
CAPITALS	Upper case indicates items that must be typed as is—although not necessarily in upper case.
itarics	This character style indicates dummy names and variables. Do not type them as is. Substitute information of the appropriate type instead.
[ ]	Brackets indicate optional items that can be omitted or supplied as desired.
...	An ellipsis indicates repetition of the immediately preceding item as many times as necessary.
{*choice1*|*choice2*}	Braces indicate a list of choices separated with vertical bars (|). Unless the braces are themselves enclosed in brackets, one choice must always appear.
“title”	A title enclosed in double quotes indicates the name of either a manual or a Section.
Ctrl+C	This plus sign means to hold down the modifier key (Alt, Ctrl, or Shift) and press the following key (here C).

`PROGRAM`
`:`
`:`
`:`
`PROGRAM`
	A vertical line of dots in sample code indicates that a portion has been omitted.

In this manual, the letter “H” at the end of a numerical value indicates that the numerical value is given in hexadecimal. The notation 1000H, for example, means hexadecimal 1000 or decimal 4096.

1 Introduction

1.1 Program Development Flow

This section describes the operation flow for developing assembly language user application programs with the MACU8 assembler package. This manual does not cover program debugging. Refer to the manual for the debugger that you are using.

Figure 1.1 gives a flowchart for this program development.

The squares in the Figure are files. If there is only one default file extension for the file, the square includes it.

An oval indicates software; a diamond, a decision branch in the operation.

The following description uses the numbers from the flowchart.

(1) Write the source (.ASM) file for the program with your choice of text editor.

(2) Use the assembler (RASU8) to convert the source code file to an object (.OBJ) file. The assembler also generates a print (.PRN) file and, if specified, an error message (.ERR) file.

(3) If desired, use the librarian (LIBU8) to store the object file in a library (.LIB) file. Saving general-purpose program modules in library files for reuse in other projects can greatly boost development efficiency. These library files can serve as linker (RLU8) input. If so specified, LIBU8 also generates a list (.LST) file listing the object modules and public symbols in the library.

(4) Use the linker (RLU8) to merge all object files making up the program into a single absolute object (.ABS) file. RLU8 resolves all external references between the object files and assigns logical segments to memory. It also generates a map (.MAP) file.

(5) Use the object converter (OHU8) to convert the absolute object file to a .HEX file. For further details on .HEX file types and formats, see Section 9 “OHU8 Object Converter.”

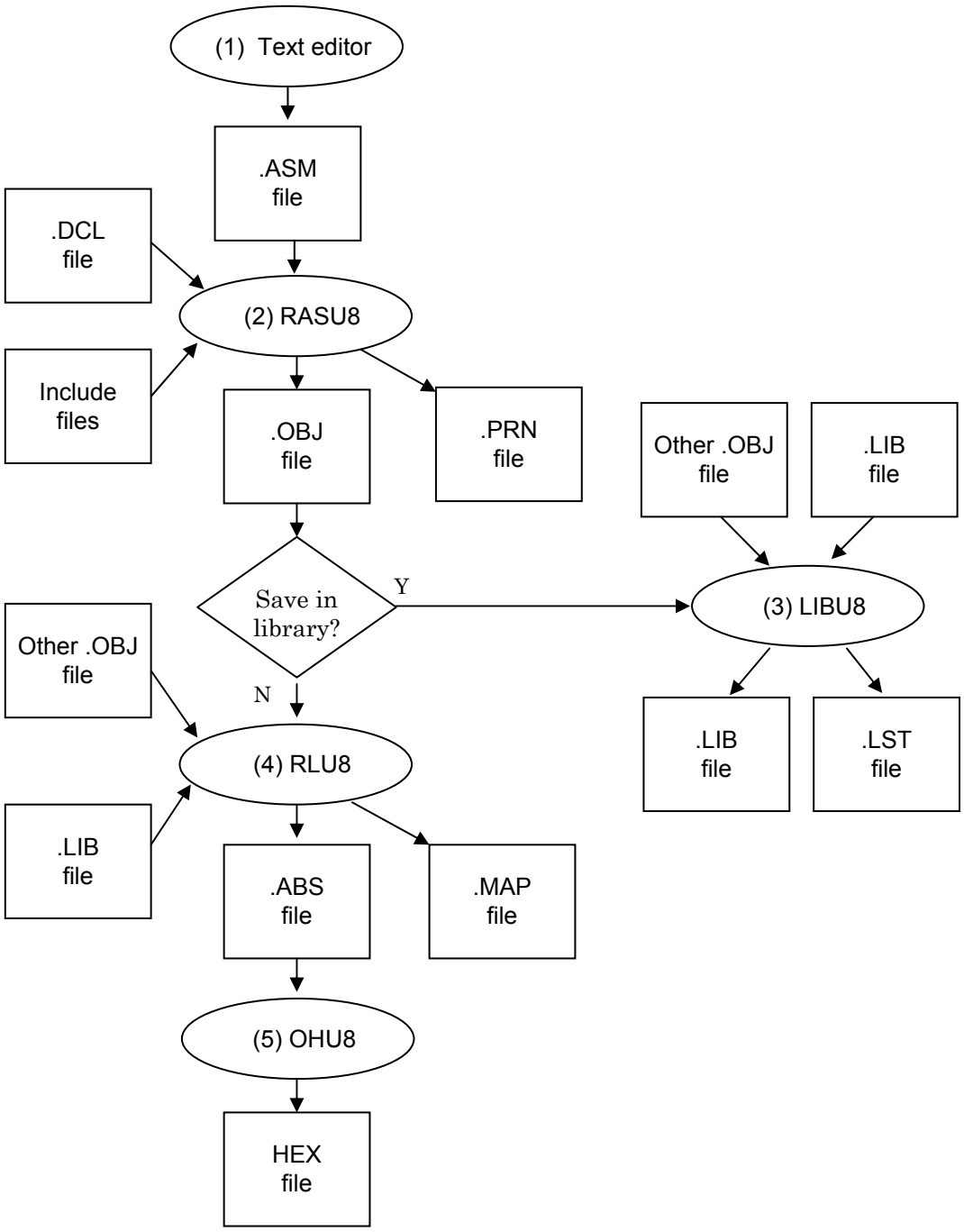


Figure 1.1 Program Development Flow

1.2 DCL Files

The tools in the MACU8 assembler package rely on DCL files to provide specific details on the target microcontroller. Changing DCL files thus allows them to support a variety of microcontrollers.

These text files always have the file extension .DCL. Each source code file must specify the target microcontroller by specifying the base name of the corresponding DCL file.

The only tool reading the DCL file is the assembler, which then stores the information from the DCL file in the object file for use by the linker.

Under no circumstances should you edit these files. The contents are very important to initializing the assembler, so any mistakes introduced can interfere with proper assembly.

The rest of this Section describes the contents of a DCL file.

1.2.1 Microcontroller Identifier

This portion names the target microcontroller. The linker checks these identifiers to determine whether it is possible to link modules.

1.2.2 Available ROM Window Range

This portion specifies the address range available for the ROM window as well as the values necessary for setting that range in the hardware. This information allows the assembler and linker to check whether the address range specified for the ROM window is valid and whether the program accesses memory properly.

1.2.3 Available Memory Space Ranges

This information allows the assembler to determine valid ranges for the code and data memory spaces and also to check operand values for access to the corresponding memory space.

This portion provides the following types of information about memory space availability.

(1) Number of physical segments in the code and data memory spaces

(2) Address ranges for each type of memory installed (ROM, RAM, and nonvolatile memory)

(3) Address ranges for the SFR region and other special regions

Note, however, that the DCL file lists only the memory ranges for memory built into the target microcontroller.

The assembler and the linker limit assignment to only those ranges specified as having memory installed.

A user application system using external memory must specify the external memory ranges not in the DCL file, but with the appropriate assembler and linker command line options.

The following is a list of those options.

RASU8 Option	RLU8 Option
/BROM(*start_address*,*end_address*)	/ROM(*start_address* , *end_address*)
/BRAM(*start_address*,*end_address*)	/RAM(*start_address* , *end_address*)
/BNVRAM(*start_address*,*end_address*)	/NVRAM(*start_address* , *end_address*)
/BNVRAMP(*start_address*,*end_address*)	/NVRAMP(*start_address* , *end_address*)

For further details on these options, see the individual option descriptions in Chapters 6 “RASU8 Relocatable Assembler” and 7 “RLU8 Linker,” respectively.

1.2.4 SFR Region Access

This information allows the assembler to check each access to the SFR region.

1.2.5 Reserved Words Representing Addresses

This portion gives the values for the reserved words representing address symbols as well as their usage types. Specific examples here include the names of registers assigned to the SFR region. Program operands can thus substitute symbolic names for actual addresses.

1.2.6 Available Instructions

This portion lists the instructions supported by the target microcontroller. Using an instruction not defined in the DCL file then produces an error message.

1.3 File Specifications

The tools in the MACU8 assembler package accept file specifications representing input sources and output destinations.

This manual divides these file specifications into the following four fields.

<drive:><directory><base_name><.extension>

A fifth term, path, represents a combination of the first two, drive and directory.

■ **Example** ■

```
C:\U8\MACU8\SRC\TEST.ASM
```

The following applies these five terms to the above example.

Name	Portion from above File Specification
Path	C:\U8\MACU8\SRC\
Drive	C:
Directory	\U8\MACU8\SRC\
Base name	TEST
Extension	.ASM

The tools support long file name specifications up to 255 bytes long, including path and file extension. Note, however, that file specifications must not include double-byte or whitespace characters.

1.4 Environment Variables

Some tools in the MACU8 assembler package provide environment variables. These are, however, optional, so set as needed or desired.

The following lists these environment variables.

Environment Variable	Description
DCL	The assembler searches the directory specified by this environment variable when it cannot find the specified DCL file in the current directory or the directory containing its executable (RASU8.EXE). For further details, see Section 5.1.1 “TYPE Directive.”
LIBU8	The library manager searches the directories specified by this environment variable when it cannot find the specified library file in the current directory. For further details, see Section 7.2.1.4 “libraries field.”

The following are examples of settings for these environment variables.

```
SET DCL=C:\U8\DCL
SET LIBU8=C:\U8\LIB
```

1.5 Using the Tools

This Section describes, using specific examples, applying the tools to achieve the desired objectives.

1.5.1 Assembling a Program

The tool for assembling source code files written in assembly language is RASU8, which has the following command line syntax.

RASU8 *source_file*

The *source_file* field specifies the source code file to assemble.

Options can appear anywhere on the command line.

For further details on the procedures for using the assembler and the options available, see Chapter 6 "RASU8 Relocatable Assembler."

■ Example ■

```
RASU8 MAIN
RASU8 SUB
RASU8 PROC1
RASU8 PROC2
```

This example assembles the source code files MAIN.ASM, SUB.ASM, PROC1.ASM, and PROC2.ASM to produce the object files MAIN.OBJ, SUB.OBJ, PROC1.OBJ, and PROC2.OBJ.

1.5.2 Adding Object Files to a Library File

The individual object files produced by the assembler can be merged into a library file with the librarian (LIBU8). Saving general-purpose program modules in library files for reuse in other projects can greatly boost development efficiency.

LIBU8 has the following command line syntax.

LIBU8 *library_file* [*operations*] [, [*list_file*][, [*output_library*]]][;]

The *library_file* field specifies the name of the input library file.

The *operations* field specifies operations on the library specified by the *library_file* field. The available operations are add (+), delete (-), replace (%), copy (*), and extract (&).

The optional *list_file* field specifies the name for the list file output.

The optional *output_library* field specifies the name for the output library file.

For further details on the procedures for using the librarian and the options available, see Chapter 8

"LIBU8 Librarian."

■ **Example** ■

```
LIBU8 MODULES +PROC1 +PROC2;
```

This example adds PROC1.OBJ and PROC2.OBJ, two object files produced by the assembler, to the library file MODULES.LIB.

1.5.3 Linking Object Files

The tool for linking object files produced by the assembler is RLU8, which has the following command line syntax.

RLU8 *object_files* [, [*absolute_file*][, [*map_file*][, [*libraries*]]]][;]

The *object_files* field specifies the names of the object and library files to link. The optional *absolute_file* field specifies the name for the output absolute object file.

The optional *map_file* field the name for the output map file.

The optional libraries field specifies the names of the library files to search to resolve unresolved external references.

Options can appear anywhere on the command line.

For further details on the procedures for using the linker and the options available, see Chapter 7 "RLU8 Linker."

■ **Example** ■

```
RLU8 MAIN SUB ,TEST,, MODULES
```

This example links the object files MAIN.OBJ and SUB.OBJ to create a single absolute object file TEST.ABS. The MODULES.LIB in the optional libraries field then tells the linker to resolve any unresolved external references by retrieving the necessary modules from the library file MODULES.LIB.

1.5.4 Converting Absolute Object Files

The absolute object file produced by the linker is in binary format, so must be converted with the object converter (OHU8) to the text-based HEX formats required by PROM writers and other tools for downloading programs to ROM.

OHU8 has the following command line syntax.

OHU8 *object_file* [*hex_file*][;]

The *object_file* field specifies the absolute object file to convert.

The optional *hex_file* field specifies the name for the output HEX file.

Options can appear anywhere on the command line.

OHU8 supports two output formats: Intel HEX and Motorola S2. There are command line options for specifying the output format. The default is Intel HEX.

For further details on the procedures for using the object converter, the options available, the HEX file types available, and the file formats, see Chapter 9 "OHU8 Object Converter."

■ **Example** ■

```
OHU8 TEST ;
```

This example converts the absolute object file TEST.ABS to the HEX file TEST.HEX using the Intel HEX format.

1.5.5 Generating C Source Level Debugging Information

C language source level debugging requires that the programmer direct the C compiler (CCU8) to include C language source level debugging information in its output based on the corresponding C language source code files.

The following shows the procedure for generating C language source level debugging information.

■ **Example** ■

```
CCU8 /TM610001 /SD HELLO.C
CCU8 /TM610001 /SD WORLD.C
RASU8 HELLO /SD
RASU8 WORLD /SD
RLU8 HELLO WORLD STARTUP /CC /SD;
```

The /SD options in the first two lines tell the C compiler to include C language source level debugging information in its output when compiling the C language source code files HELLO.C and WORLD.C to the assembly language source code files HELLO.ASM and WORLD.ASM.

The /SD options in the next two lines tell the assembler to pass this C language source level debugging information on to its output, the object files HELLO.OBJ and WORLD.OBJ.

Similarly, the /SD option in the last line tells the linker to pass this C language source level debugging information from the two object files on to its output, the absolute object file HELLO.ABS, when it links them with the start-up file STARTUP.OBJ.

1.5.6 Generating Assembly Level Debugging Information

Assembly level debugging requires that the programmer direct the assembler (RASU8) to include assembly level debugging information in its output based on the corresponding assembly language source code files.

The following shows the procedure for generating assembly level debugging information.

■ **Example** ■

```
RASU8 HELLO /D
RASU8 WORLD /D
RLU8 HELLO WORLD /D ;
OHU8 HELLO /D;
```

The /D options in the first two lines tell the assembler to include assembly level debugging information in its output when compiling the HELLO.ASM and WORLD.ASM to its output, the object files HELLO.OBJ and WORLD.OBJ.

The /D option in the third line tells the linker to pass this assembly level debugging information from the two object files on to its output, the absolute object file HELLO.ABS, when it links them together.

Finally, the /D option in the last line tells the object converter to pass this assembly level debugging information on to its output, the Intel HEX file HELLO.HEX.

2 Programming Basics

2.1 Writing a Program

This section describes the basic aspects of writing an assembly language source code program.

2.1.1 Sample Program

The discussion that follows makes references to the following assembly language source code file.

```
 1: /************************************
 2:     SAMPLE PROGRAM
 3: ************************************/
 4:     TYPE (M610001)          ; Specify DCL file
 5:     MODEL SMALL, NEAR       ; Specify memory model
 6:     ROMWINDOW 0, 3FFFH      ; Specify ROM window
 7:
 8:     REL_CODE SEGMENT CODE   ; Define segment symbol
 9:     REL_DATA SEGMENT DATA   ; Define segment symbol
10:     STACKSEG 100H           ; Define stack segment
11:
12:     EXTRN DATA: _$$SP       ; Define external symbol
13:
14:     CSEG AT 0:0H            ; Start CODE segment
15:     DW      _$$SP           ; Initialize stack pointer
16:     DW      _START          ; Start reset vectors
17:
18:     RSEG    REL_CODE        ; Start CODE segment
19: _START:
20:     MOV     R0,     #1CH
21:     ST      R0,     0FF00H
22:     MOV     ER0,    #00H
23:     MOV     ER2,    ER0
24:     LEA     OFFSET _BUF
25:     ST      XR0,    [EA+]
26:     ST      XR0,    [EA+]
27:     ST      XR0,    [EA+]
28:     ST      XR0,    [EA+]
29:     BAL     $
30:
31:     RSEG    REL_DATA        ; Start DATA segment
32: _BUF:
33:     DS      10H
34:
35:     END                     ; End program
```

The line numbers down the left edge have been added for use in the following discussion. They do not appear in the actual assembly language source code file itself.

2.1.1.1 Program Components

A program consists of a sequence of source statements written in U8 assembly language. Source statements combine microcontroller instructions, directives, operands, and comments. They fall into three types.

(1) Directives

(2) Instructions

(3) Empty statements

The following describes each type in turn.

2.1.1.1.1 Directives

This type of source statement contains an RASU8 assembler directive. Syntax differs with the directive.

Note, however, that all directives can take a comment at the end. A comment starts with a semicolon (;) and ends with the line feed terminating the line.

The following lines in the sample program code are all directives: lines 4 through 6, lines 8 through 10, line 12, lines 14 through 16, line 18, line 31, line 33, and line 35.

For further details on the syntax for each directive, see Section 5 “Detailed Directive Descriptions”

2.1.1.1.2 Instructions

This type of source statement contains a microcontroller instruction. Instruction statements have four fields. The order of these fields cannot be changed.

An instruction ends with a line feed.

Instructions have the following syntax.

[*label*:]*mnemonic* [*operand_field*][;*comment*]

label is a label, an address for the instruction. If present, it must always be followed by a colon (:). This label has an address in the logical segment to which it belongs.

mnemonic is a microcontroller instruction mnemonic. For further details on instructions, refer to the related documents.

operand_field contains the operands required by the instruction syntax. For further details on these operands, see the related documents. For further details on operand syntax, see Section 4 “Addressing

and Instruction Types."

comment is a comment for the instruction. A comment starts with a semicolon (;) and ends with the line feed terminating the line.

Lines 20 through 29 in the sample program code all contain instructions.

2.1.1.1.3 Empty Statements

This type of source statement contains neither a directive nor an instruction. The following is the syntax.

[*label*:][;*comment*]

The following lines in the sample program code are all empty statements: line 7, line 11, line 13, line 17, line 19, line 30, line 32, and line 34.

2.1.1.1.4 Block Comments

A block comment can span multiple lines. It starts with "/*" and ends with "*/." The assembler ignores everything between these two markers—including line feeds and all character combinations except "*/." Adding these two delimiters turns the enclosed block into a comment.

Block comments can be nested.

```
/* All
/* this */
is treated as a comment. */
```

The entire example above is treated as a comment.

Lines 1 through 3 in the sample program code form a block comment.

2.1.1.2 Specifying Program Start

Each program that you write must start with initialization directives specifying to the assembler such details as the target microcontroller on which the program will run, which memory model it will use, and whether it will use a ROM window.

Lines 4 through 6 in the sample program code contain the following assembler initialization directives.

TYPE

This directive specifies the base name of the DCL file for the target microcontroller. It must appear in each source code file.

MODEL

This directive specifies the data model and, as necessary, the memory model.

ROMWINDOW

This directive specifies the starting and end addresses for the ROM window. If the program does not use a ROM window, use the NOROMWIN directive instead.

Use this directive to specify the ROM window if one is needed.

2.1.1.3 Defining Reset Vectors

Looking carefully at the absolute CODE segment in lines 14 through 16 in the sample program code reveals that the program initializes the first two words starting at offset 0 in physical segment #0. These addresses represent the reset vector region. These reset vector definitions must appear in one of the files for the program. Failure to include them can lead to erratic operation.

2.1.1.4 Specifying Program End

Line 35 in the sample program code contains the END directive. This directive specifies to the assembler that the program ends at that point. Nothing beyond that line is assembled. If the program does not contain an END directive, the assembler processes everything through to the end of the file.

2.2 Memory Spaces

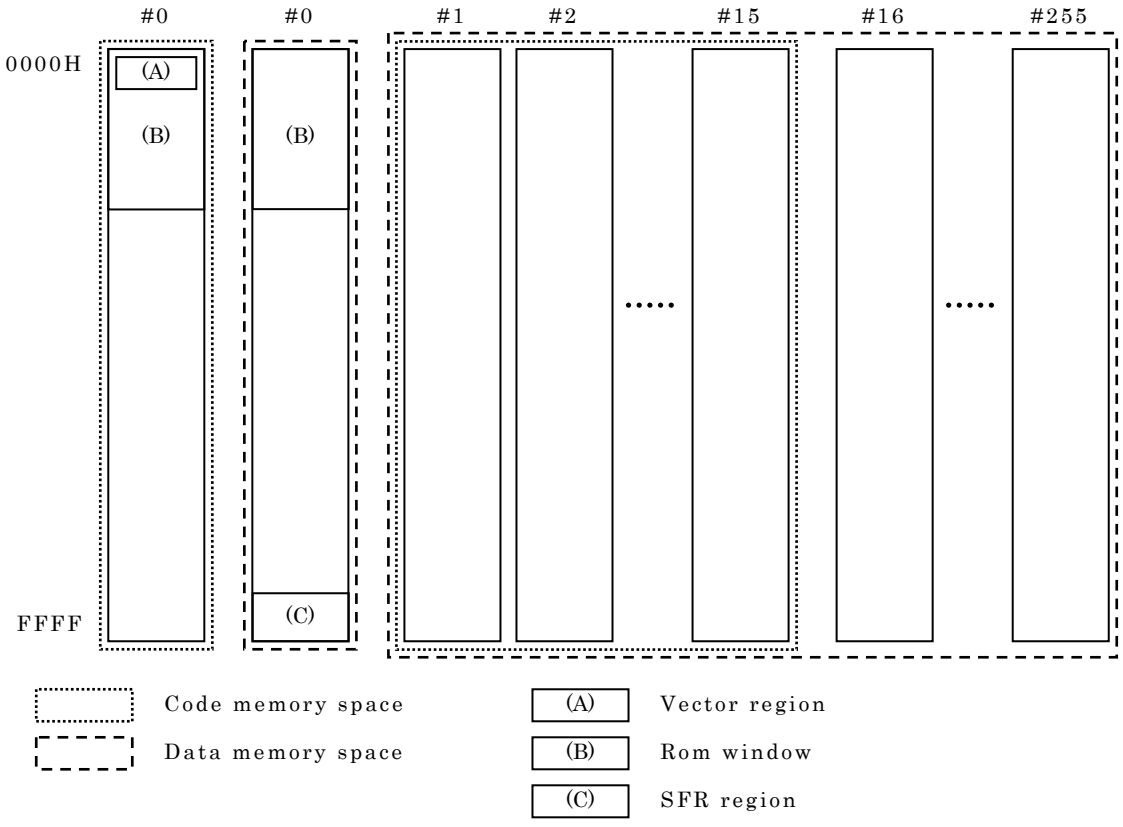
This Section describes the memory spaces supported by the nX-U8 architecture.

2.2.1 Overview of Memory Spaces

The nX-U8 architecture features a 1-megabyte code memory space and a 16-megabyte data memory space. The primary use for the code memory space is holding the machine code necessary for program execution; that for the data memory space, holding initialization data and providing temporary storage during execution.

Each memory space is divided into physical segments of 64K bytes each. Physical segment #0, however, has a different memory space structure from the others, #1 and higher.

The following gives an overview of the nX-U8 memory spaces.



Where physical segment #0 differs is in providing independent code and data memory spaces. Physical segments #1 to #15 are shared and may be used either as code memory space or as data memory space. Physical segments #16 to #255 are only available as data memory space.

To switch physical segments, the user program must manipulate the appropriate segment register: the code segment register (CSR), which specifies the current physical segment for code memory, or the data segment register (DSR), its data memory counterpart.

The 4-bit CSR register offers a choice of 0 to 15; the 8-bit DSR register, 0 to 255. The supported address ranges, therefore, are 0:0000H to F:FFFFH or 1 megabyte for the code memory space and 0:0000H to FF:FFFFH or 16 megabytes for the data memory space.

■ Aside ■

This manual uses the following notation for representing a complete address consisting of a physical segment address and an offset within that physical segment.

segment:*offset*

where *segment* is the physical segment address and *offset*, the offset.

The notation 1:2345H, for example, represents the address at offset 2345H in physical segment #1.

2.2.2 Special Regions

Physical segment #0 also contains special regions. The code memory space has the vector region and the ROM window; the data memory space, the ROM window and the SFR region.

2.2.2.1 Vector Region

This special region contains vectors, words holding address pointers to the entry points for the routines for processing resets and interrupts. It is further subdivided into the following three regions.

Vector Type	Description
Resets	This region holds the entry points for processing resets—that is, the initial value for the stack pointer at address 0 and the reset routine entry point at address 2. Failure to properly initialize these vectors can lead to erratic operation.
Hardware interrupts	This region holds the entry points for processing hardware interrupts. If the program does not use these interrupts, this region is available for normal program code.
Software interrupts	This region holds the entry points for processing software interrupts. If the program does not use these interrupts, this region is available for normal program code.

For further details on the vector region, refer to the hardware manual.

2.2.2.2 ROM Window Region

This special region provides access to a portion of physical segment #0 in the code memory space using memory access instructions. The ROM window function provides read access to this region from the same addresses in the physical segment #0 in the data memory space if the corresponding region is not mapped to internal RAM. One application is accessing read-only data (table data).

Note, however, that actually using a ROM window requires hardware setup and control.

This ROM window in the code memory space is for holding table data and program code. Note that the ROM window in the data memory space is treated as a window into the code memory space, so you cannot assign read/write data there.

2.2.2.3 SFR Region

This special region contains the special function registers (SFRs) controlling peripheral functions. You cannot assign data there.

2.3 Address Spaces

These are logical spaces provided to assist the assembler and the linker in assigning regions for program code and data. They are the spaces for relocating logical segments (described below in the next Section) and for use as memory addressing targets.

The following Table lists these address spaces.

Address Space	Description	Address Granularity
CODE	Code memory space. Specifically, the space for program code.	Byte
DATA	Data memory space except for the ROM window and regions with nonvolatile memory. Specifically, the space for RAM data.	Byte
BIT	Data memory space except for the ROM window and regions with nonvolatile memory. Specifically, the space for RAM data.	Bit
NVDATA	Nonvolatile memory regions in the data memory space except for any overlap with the ROM window. Specifically, the space for nonvolatile data.	Byte
NVBIT	Nonvolatile memory regions in the data memory space except for any overlap with the ROM window. Specifically, the space for nonvolatile data.	Bit
TABLE	ROM window in the code memory space and physical segments #1 and higher in the data memory space. Specifically, the space for read-only data—in other words, table data.	Byte

2.4 Logical Segments

The nX-U8 architecture has six address spaces: CODE, DATA, BIT, NVDATA, NVBIT, and TABLE. An nX-U8 assembly language user program must, therefore, specify to the assembler and the linker which program portions go in which address spaces using what are called logical segments representing a clustering into a single range of contiguous addresses.

All nX-U8 program source code must belong these logical segments because the linker uses them to decide where to assign them in the memory spaces.

The following summarizes these logical segments.

Logical Segment	Description
CODE	Logical segment for CODE address space
DATA	Logical segment for DATA address space outside the SFR region
BIT	Logical segment for BIT address space outside the SFR region
NVDATA	Logical segment for NVDATA address space
NVBIT	Logical segment for NVBIT address space
TABLE	Logical segment for TABLE address space

2.4.1 Specifying Logical Segments

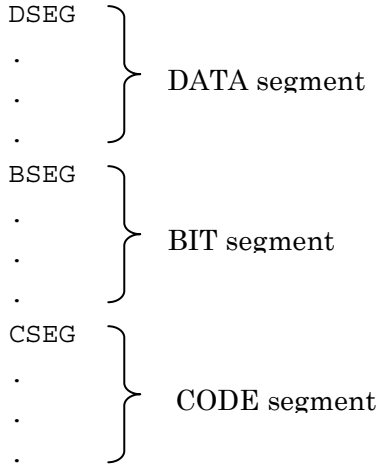
Logical segments represent a clustering into a single range of contiguous addresses in a particular address space. Every source statement in an nX-U8 assembly language user program belongs to a logical segment.

The following segment definition directives are available to specify the logical segments for each part of the program.

Logical Segment	Segment Definition Directive for Absolute Segment	Segment Definition Directive for Relocatable Segment
CODE	CSEG	RSEG
DATA	DSEG	RSEG
BIT	BSEG	RSEG
NVDATA	NVSEG	RSEG
NVBIT	NVBSEG	RSEG
TABLE	TSEG	RSEG

As the Table shows, there are two types of directive for defining logical segments. We shall postpone discussion of the RSEG directive until later and give an example using the absolute directives CSEG, DSEG, and BSEG.

■ **Example** ■



In this example, the DSEG directive starts a DATA segment. Later, a BSEG directive starts a BIT segment. Finally, a CSEG directive starts a CODE segment.

As the above example illustrates, one logical segment definition remains in effect until the next. This contiguous block of source code forms a logical segment in the program. Each source code statement always belongs to a logical segment.

A logical segment is assigned within a single physical segment. It cannot be defined to span multiple physical segments.

Note also that logical segments cannot be assigned to the SFR region.

2.4.2 Assigning Source Code to Logical Segments

Assembly language user programs must always keep these logical segments in mind. In other words, source code for the CODE address space must start with a CODE segment directive. Similarly, source code for the TABLE, DATA, BIT, NVDATA, and NVBIT address spaces must start with the corresponding segment directive. For further details on address spaces, see Section 2.3 “Address Spaces.”

The following summarizes the primary source code types for each logical segment.

Logical Segment	Primary Source Code Types
CODE	Microcontroller instructions. DW directives initialized to specified numerical values. GJMP and GBcond directives subsequently converted to the optimal branch instruction.
TABLE	DB and DW directives initialized to specified numerical values.
DATA	DS directives reserving memory for data in bytes.
BIT	DBIT directives reserving memory for data in bits.
NVDATA	DB and DW directives initialized to specified numerical values. DS directives reserving memory for data in bytes.
NVBIT	DBIT directives reserving memory for data in bits.

2.4.3 Absolute vs. Relocatable Segments

The assembler divides logical segments into absolute and relocatable based on the tool that fixes the address. An absolute segment is one for which the assembler fixes the address. Otherwise, the logical segment becomes a relocatable segment, and the linker decides its address.

There are directives for defining each logical segment type.

2.4.3.1 Absolute Segments

An absolute segment is a logical segment for which the assembler fixes the address. The assembler manages them by physical segment. The programmer can, therefore, specify the following when defining the absolute segment.

(1) Starting address for the absolute segment

(2) Physical segment address for the absolute segment

If the above are not specified, the absolute segment inherits previous settings. For further details on this inheritance process, see the Section 5.4 “Absolute Segment Definition Directives.”

If a program has multiple absolute segments with the same physical segment address, and these absolute segments do not have leading addresses specifications, these absolute segments from a single absolute segment.

Anything at the start of the program before the first logical segment definition becomes an absolute

segment in the CODE address space. If the program has no logical segment definitions, therefore, the entire program fits into this segment. This segment starts at offset 0 in physical segment address #0.

Absolute segments have the following names based on the address space to which they belong.

Absolute Segment	Description
Absolute CODE segment	Absolute segment belonging to the CODE address space
Absolute DATA segment	Absolute segment belonging to the DATA address space
Absolute BIT segment	Absolute segment belonging to the BIT address space
Absolute NVDATA segment	Absolute segment belonging to the NVDATA address space
Absolute NVBIT segment	Absolute segment belonging to the NVBIT address space
Absolute TABLE segment	Absolute segment belonging to the TABLE address space

The following directives define absolute segments belonging to each address space.

Directive	Description
CSEG	Defies an absolute segment in the CODE address space
DESG	Defies an absolute segment in the DATA address space
BSEG	Defies an absolute segment in the BIT address space
NVSEG	Defies an absolute segment in the NVDATA address space
NVBSEG	Defies an absolute segment in the NVBIT address space
TSEG	Defies an absolute segment in the TABLE address space

■ Example ■

```
        CSEG    #1 AT 100H  ; First absolute CODE segment definition
        NOP

        DSEG    #0          ; First absolute DATA segment definition
LABEL7: DS      2

        CSEG    #1          ; Second absolute CODE segment definition
        MOV     ER0, #0

        DSEG                ; Second absolute DATA segment definition
LABEL8: DS      2
```

This example contains four absolute segment definitions. The first absolute CODE segment definition

specifies a starting address of 100H in physical segment address #1. The second absolute CODE segment definition specifies only the physical segment address (#1). Because it does not specify a starting address, it inherits one from the first absolute CODE segment belonging to the same physical segment. As a result, the MOV ER0, #0 instruction follows at the address after the NOP instruction.

The first absolute DATA segment definition does not specify a starting address—only the physical segment address. The second absolute DATA segment definition specifies neither a starting address nor a physical segment address. It therefore inherits address the from the first absolute DATA segment.

2.4.3.2 Relocatable Segments

A relocatable segment is a logical segment to which the assembler cannot assign an address. The linker decides.

The assembler manages relocatable segments with segment symbols. If a source code file defines multiple relocatable segments with the same segment symbol, the assembler assigns these relocatable segments together in memory. It then assigns relocatable segments in the order in which they are defined in the file.

If the same segment symbol appears in relocatable segments from different source code files, the linker merges these relocatable segments before assigning them to memory. The value assigned to the segment symbol is the starting address for the merged segment. These merged relocatable segments are called partial segments. For further details on deciding the relocatable segment address and link order, see the Section 7.5.3 “Linking Segments.”

Relocatable segments have the following names based on the address space to which they belong.

Relocatable Segment	Description
Relocatable CODE segment	Relocatable segment belonging to the CODE address space
Relocatable DATA segment	Relocatable segment belonging to the DATA address space
Relocatable BIT segment	Relocatable segment belonging to the BIT address space
Relocatable NVDATA segment	Relocatable segment belonging to the NVDATA address space
Relocatable NVBIT segment	Relocatable segment belonging to the NVBIT address space
Relocatable TABLE segment	Relocatable segment belonging to the TABLE address space

The following directive defines a relocatable segment.

Directive	Description
RSEG	Defines a relocatable segment

The RSEG operand specifies a segment symbol defined with a SEGMENT directive specifying the address space to which the relocatable segment belongs.

■ **Example** ■

```
CODESEG2    SEGMENT CODE    #1  ; Define segment symbol CODESEG2
DATASEG2    SEGMENT DATA        ; Define segment symbol DATASEG2
            RSEG    CODESEG2    ; Define relocatable CODE segment
            NOP
            RSEG    DATASEG2    ; Define relocatable DATA segment
LABEL9:     DS      2
            RSEG    CODESEG2    ; Define relocatable CODE segment
            MOV     ER0, #0
```

The above example defines two segment symbols (CODESEG2 and DATASEG2), which it then uses to define relocatable segments. The former definitions use SEGMENT directives; the latter, RSEG directives.

A segment symbol definition specifies the address space for assigning relocatable segments with that segment symbol.

A segment symbol definition can also specify the physical segment address. The CODESEG2 definition in this example specifies both the address space (CODE) and the physical segment address (#1). The DATASEG2 definition, in contrast, specifies only the address space (DATA). It does not specify a physical segment address.

This example uses the same segment symbol (CODESEG2) for two relocatable CODE segments. The linker therefore assigns these relocatable segments together in memory. As a result, the MOV ER0, #0 instruction follows at the address after the NOP instruction.

As this example illustrates, the programmer cannot specify an absolute address for assigning the relocatable segment. The reason is that relocatable segments are for writing programs that are not affected by the absolute addresses assigned to logical segments. There is, however, an RLU8 command line option for specifying the absolute address for assigning the relocatable segment.

2.4.4 Physical Segment Attributes

Of the two types of logical segments, absolute segments have their assignment addresses specified from the start, so their physical segment addresses are automatically determined. This is not always the case for relocatable segments, however. Logical segments therefore have physical segment attributes for specifying the status of this physical segment address determination.

The following lists these physical segment attributes.

Physical Segment Attribute	Description
#*n*	This value indicates that the assembler was able to determine the physical segment address.
ANY	This special value indicates that the relocatable segment is missing a physical segment address. The linker must decide.

Physical segment attributes are not limited to logical segments. Communal symbols, external symbols, and others have them too.

2.4.5 Usage and Segment Types

The usage type is an attribute indicating the intended purpose for a value.

The following usage types are available.

Usage Type	Description
NUMBER	This indicates that the value represents a numerical value.
CODE	This indicates that the value represents an address in the CODE address space.
DATA	This indicates that the value represents an address in the DATA address space.
BIT	This indicates that the value represents an address in the BIT address space.
NVDATA	This indicates that the value represents an address in the NVDATA address space.
NVBIT	This indicates that the value represents an address in the NVBIT address space.
TABLE	This indicates that the value represents an address in the TABLE address space.
TBIT	This indicates that the value represents an address in the TBIT address space.
NONE	This indicates that the value represents an address, but the address space is indeterminate.

The usage type NUMBER indicates a numerical value, so giving a value this usage type is exactly the

same as declaring the value as a numeric type.

The usage types CODE, DATA, BIT, NVDATA, NVBIT, and TABLE are also called segment types, the term which this manual uses for the attribute specifying the address space type for an address. The usage type indicates the intended purpose for numerical values and addresses.

Usage types play an important role in address checking. Many instructions and directives take operands, but impose limits as to the type of values that can appear in those positions. The assembler therefore compares the usage type of these operands against the predetermined possibilities.

2.4.6 Address Ranges Available for Assigning Segments

The address range available for a logical segment depends on the segment type and the type of memory installed. The assembler checks whether absolute segments are within the corresponding address ranges available. The linker checks whether relocatable segments are within the corresponding address ranges available as it fits them into free regions.

The following lists the address ranges available for assigning segments of each type.

Logical segment	Physical segment range	Offset range
CODE	#0 to #15	Ranges specified for the ROM and nonvolatile memory in the corresponding physical segment.
DATA / BIT	#0 to #255	Ranges specified for the RAM in the corresponding physical segment. Note, however, that physical segment #0 excludes any ranges that overlap with the ROM window, nonvolatile memory regions, or the SFR region.
NVDATA / NVBIT	#0 to #255	Ranges specified for the nonvolatile memory in the corresponding physical segment. Note, however, that physical segment #0 excludes any ranges that overlap with the ROM window, on the SFR region.
TABLE	#0 to #255	Ranges specified for the ROM in the corresponding physical segment. Note, however, that physical segment #0 limits addresses to the ROM window.

Both the assembler and the linker treat regions with no memory installed as off limits for logical segment assignment. Because the DCL file lists only the memory ranges built into the target microcontroller, a

user application system using external memory must specify the external memory ranges with the appropriate assembler and linker command line options.

2.4.7 Special Relocatable Segments

The following relocatable segments receive special treatment that makes them different from normal ones.

(1) Stack segment

(2) Dynamic segments

The following describes each type in turn.

2.4.7.1 Stack Segment

This special relocatable segment contains a stack region. It must be assigned to physical segment #0 in the data memory space.

The STACKSEG directive defines a stack segment. The sole operand specifies the stack size.

The stack segment is treated as a relocatable DATA segment. Its starting and end addresses are therefore not available directly. They are, however, available through symbol references. Referencing the symbol $STACK, the segment name for the stack segment, gives the starting address. The assembler automatically defines it when the program defines the stack segment.

Referencing the symbol _$$SP yields the initial value for the stack pointer—that is, one past the last address in the stack segment. Note, however, that this symbol is not defined automatically. Before it can be referenced, the program must declare with an EXTRN directive.

The nX-U8 architecture initializes the stack pointer from address 0 in the reset vector region. The following is sample code for defining the stack segment and specifying the initial value for the stack pointer.

```
STACKSEG  200H
EXTRN DATA NEAR:_$$SP
    CSEG  AT  00H
    DW  _$$SP
```

The above example defines a stack segment of 200H bytes. It then initializes address 0 in the reset vector region with the initial value for the stack pointer. If the linker, assigns the stack segment to the address 0:8000H , for example, _$$SP has the value 8200H.

Unlike normal relocatable segments, the stack segment allows the programmer to change the segment size at link time with the linker command line option /STACK(*stack_size*), where *stack_size* is the new size.

2.4.7.2 Dynamic Segments

These special relocatable segments, intended for use with the dynamic memory allocation functions of higher-level languages, do not acquire a size at assembly. Instead, the linker assigns all logical segments to the appropriate memory spaces and then assigns the largest data memory regions remaining to dynamic segments.

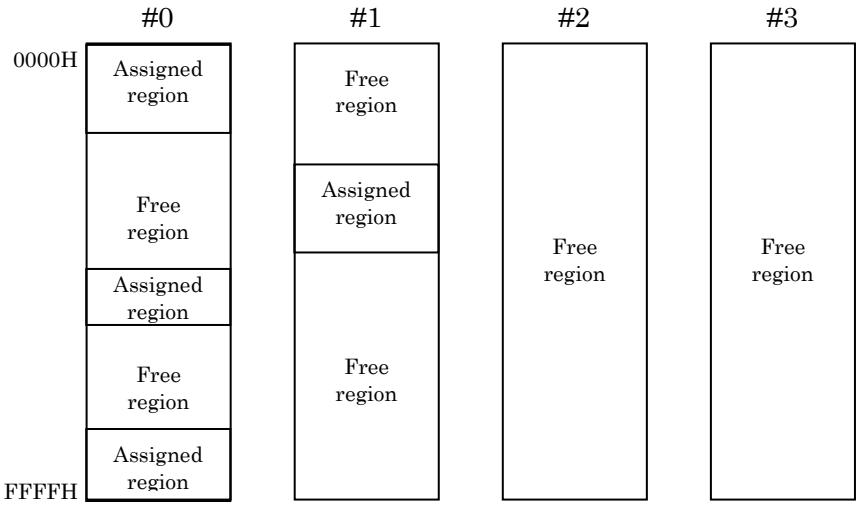
Note, however, that a dynamic segment cannot span multiple physical segments, so the maximum size is 64K bytes.

There can be a dynamic segment for each physical segment present.

The following are examples of dynamic segment definitions.

```
dynamic_0   segment data #0 dynamic
dynamic_2   segment data #2 dynamic
dynamic_any segment data any dynamic
```

In these examples, dynamic_0 is a dynamic segment assigned to physical segment #0, dynamic_2 a dynamic segment assigned to physical segment #2, and dynamic_any a dynamic segment assigned to a physical segment chosen by the linker.



Data Memory Space

Consider, for example, the above set of free regions after all logical segments other than the dynamic ones have been assigned. Furthermore, let us assume that the free regions all contain RAM. If so, the linker assigns dynamic_0 to free region (A), dynamic_2 to free region (E), and dynamic_any to free region (F).

2.5 Location Counters

The assembler maintains a counter, the location counter, that always holds the logical segment address for the instruction that it is currently assembling.

Relocatable segments each have their own location counter. Absolute segments use the location counter for the current physical segment in the current address space.

2.5.1 Initializing Location Counters

2.5.1.1 Initializing Absolute Segment Location Counters

There is a separate absolute segment location counter for each physical segment in the address space. When the assembler loads, it initializes these location counters to their default offsets (see Table below). Each absolute segment in a physical segment uses that default as its starting address.

The following lists the initial values for absolute segment location counters.

Segment Type	Initial Offset for Absolute Segments
CODE	Starting address for ROM installed in the corresponding physical segment.
DATA	Starting address for RAM installed in the corresponding physical segment. Note, however, that physical segment #0 uses the starting address for built-in RAM.
BIT	Starting (bit) address for RAM installed in the corresponding physical segment. Note, however, that physical segment #0 uses the starting address for built-in RAM.
NVDATA	Starting address for nonvolatile memory installed in the corresponding physical segment.
NVBIT	Starting (bit) address for nonvolatile memory installed in the corresponding physical segment.
TABLE	Starting address for ROM installed in the corresponding physical segment.

2.5.1.2 Initializing Relocatable Segment Location Counters

Relocatable segments each have their own location counter. The assembler initializes the corresponding location counter to zero when the program defines the segment symbol.

2.5.2 Modifying Location Counters

The following microcontroller instructions and assembler directives modify the contents of the corresponding location counter.

Overt starting address in absolute segment definition

Specifying a starting address in an absolute segment definition resets the location counter to that address.

Microcontroller instructions

The CODE segment location counter increments by the instruction length in bytes after each instruction.

GJMP and GBcond directives

The CODE segment location counter increments by the instruction length in bytes after the assembler converts these directives to the optimal branch instruction.

DS directive

The segment (DATA, TABLE, NVDATA, or CODE) location counter increments by the number specified by the operand.

DBIT directive

The segment (BIT or NVBIT) location counter increments by the number specified by the operand.

DB and DW directives

The segment (CODE, TABLE, or NVDATA) location counter increments by the number of bytes specified by the operand.

ORG directive

This directive sets the segment location counter to the value specified by the operand.

2.5.3 Referencing Location Counters

The dollar sign ($) references the current location counter contents for the logical segment to which the source code belongs. For this reason, the dollar sign is sometimes called the location counter symbol.

2.6 Memory Models

The nX-U8 architecture provides hardware control over the number of physical segments accessible as code memory. The assembler uses the concept of memory model to support these hardware specifications.

The memory model limits the number of physical segments accessible as code memory and the instructions available regardless of the DCL file contents.

The assembler command line options /MS and /ML or the MODEL directive inside the source code file specify the memory model. Note that these only specify the memory model to the assembler. They do not actually configure the hardware for the memory model. That requires hardware setup and control.

The assembler defaults to the SMALL model.

The following lists the range of physical segments accessible with each memory model.

Memory Model	Description
SMALL	Only physical segment #0 in the code memory space is available. Specifying #1 or higher for the physical segment attributes with a SEGMENT directive produces an error message. Specifying #1 or higher for the physical segment portion of a branch target address produces an error message.
LARGE	Physical segments #0 to #15 in the code memory space are all available.

The memory model must match for all modules in the program. Attempting to link modules with different memory models causes the linker to abort with an error message.

2.7 Data Models

For the data memory space, there is no hardware function controlling the number of physical segments accessible. Preceding a memory access instruction with the appropriate code for loading DSR is all it takes to access physical segment #1 or higher in the data memory space. The nX-U8 assembly language specifications therefore provide addressing specifiers for specifying to the assembler whether to insert this code for such memory accesses.

The following summarizes this addressing specifiers.

Addressing Specifier	Description
NEAR	This specifier suppresses the code for loading DSR, limiting data memory access to physical segment #0.
FAR	This specifier inserts code for loading DSR immediately before each memory access instruction, removing all physical segment limits on memory access.

Adding an addressing specifier to a memory access instruction forces the assembler to generate the corresponding instruction code. Overt addressing specifiers are not always necessary, however. In the absence of an addressing specifier, the assembler assumes the one specified by the data model, inserting or skipping the DSR code accordingly.

There are two data models: NEAR and FAR.

NEAR Model

For most memory access instructions without overt addressing specifiers, this model assumes NEAR. The assembler overrides this and uses FAR, however, for memory access instructions whose operands are expressions without forward references and have the physical segment attribute ANY.

For segment and communal symbols without overt addressing specifiers, the assembler assigns the physical segment attribute #0 to assign them to physical segment #0.

For external symbols, the assembler assumes that the corresponding relocatable symbols are assigned to physical segment #0. The linker issues an error message, however, if it discovers that such a relocatable symbol is, in fact, not assigned to physical segment #0.

FAR Model

For most memory access instructions without overt addressing specifiers, this model assumes FAR. The assembler overrides this and uses NEAR, however, for memory access instructions whose operands are expressions without forward references and have the physical segment attribute #0.

For segment and communal symbols without overt addressing specifiers, the assembler assigns the

physical segment attribute ANY.

For external symbols, the assembler assumes that the corresponding relocatable symbols have the physical segment attribute ANY.

The assembler command line options /DN and /DF or the MODEL directive inside the source code file specify the data model.

```
TYPE (M610001)
MODEL NEAR
REL_DATA_SEG SEGMENT DATA
    RSEG    REL_DATA_SEG
VAR1:
    DS      2

    CSEG AT 1000H
    MOV     ER0, #00H
    ST      ER0, VAR1  ; Equivalent to ST ER0, NEAR VAR1
```

The above sample program code specifies NEAR for the data model, so the assembler sets the physical segment attribute to #0 to assign the segment REL_DATA_SEG to physical segment #0. Then, when the memory access instruction accesses the variable VAR1 in that segment, the assembler knows that the segment will always be assigned to physical segment #0, so uses NEAR access.

```
TYPE (M610001)
MODEL FAR
REL_DATA_SEG SEGMENT DATA
    RSEG    REL_DATA_SEG
VAR1:
    DS      2

    CSEG AT 1000H
    MOV     ER0, #00H
    ST      ER0, VAR1  ; Equivalent to ST ER0, FAR VAR1
```

The above sample program code specifies FAR for the data model, so the assembler sets the physical segment attribute to ANY to assign the segment REL_DATA_SEG to physical segment #1 or higher. Then, when the memory access instruction accesses the variable VAR1 in that segment, the assembler knows that the segment could be outside physical segment #0, so uses FAR access.

The data model defaults to NEAR.

■ Aside ■

With the NEAR model, a memory access instruction with a forward symbol reference in an operand sometimes produces an error message. The following is an example.

```
    TYPE (M610001)
    MODEL   NEAR
BWD_FAR EQU 1:8000H
    L       R0, FWD_FAR     ; Produces an error message
    L       R1, BWD_FAR     ; Treated as L R1, FAR BWD_FAR
FWD_FAR EQU 1:8000H
```

The FWD_FAR symbol represents a forward reference; BWD_FAR, a backward one. Both definitions specify the same address value (1:8000H), however.

With the NEAR model, the assembler assumes NEAR access for such forward symbol references. As a result, when it subsequently discovers that the definition is not in physical segment #0, it flags the forward symbol reference with an error message.

2.8 Specifying ROM Window Range

The programmer can specify the address range for the ROM window in the user source program. The ROMWINDOW directive tells the assembler not only that the program uses a ROM window, but also specifies the starting and end addresses for the ROM window. The NOROMWIN directive, in contrast, just says that the program does not use a ROM window.

Note, however, that actually using a ROM window requires more than just supplying this directive (ROMWINDOW or NOROMWIN). It also requires hardware setup and control.

Neither directive is absolutely necessary, but skipping both interferes with proper address checking for physical segment #0 in the data memory space.

Neither Directive

The assembler assumes that the program uses a ROM window, but does not know the address range, so cannot properly perform range checks on addresses in physical segment #0 because the only address ranges that it knows in the data memory space are those for the built-in RAM and SFRs. The result, therefore, is a warning message from the assembler for each address outside these two regions.

ROMWINDOW Directive

This directive specifies the address range for the ROM window, so the assembler can apply full range checking. Note, however, that the address range must be identical across all link modules. The linker aborts with an error message if modules use different ranges.

NOROMWIN Directive

This directive specifies that the program does not use a ROM window and thus tells the assembler that there is no TABLE address space in physical segment #0. Allocating a region of type TABLE in physical segment #0 then produces an error message.

Linking modules not using a ROM window (NOROMWIN directive) with ones that do (ROMWIN or neither directive) is not allowed, so the linker aborts with an error message.

3 Program Components

3.1 Program Elements

The elements available to assembly language programs are the character set, constants, symbols, and the location counter symbol ($).

The rest of this Section describes these topics individually.

3.1.1 Character Set

The characters available to the assembly language programmer fall into the following groups.

1. Letters, digits, underscore, question mark, and dollar sign
2. Whitespace characters
3. Line feed and carriage return
4. Special characters
5. Operators
6. Escape sequences
7. Double-byte characters

Note, however, that character constants, string constants, and comments are limited to the characters expressed by the single-byte codes 00H to 0FFH.

3.1.1.1 Letters, Digits, Underscore, Question Mark, and Dollar Sign

Programs can use upper and lower case letters, decimal digits, the underscore (_), the question mark (?), and the dollar sign ($).

The following is a summary.

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

3.1.1.2 Whitespace Characters

The whitespace characters space (20H) and tab (09H) serve to separate adjoining elements in the source code. Sequences of these two characters are functionally equivalent to a single space.

3.1.1.3 Line Feed and Carriage Return

The line feed (0AH) indicates the end of a source code line.

The carriage return (0DH) has no syntactic meaning. The assembler simply skips carriage returns.

3.1.1.4 Special Characters

Special characters modify the meaning of elements before and after themselves.

The following lists these special characters.

Character	Description
#	Specifies immediate addressing. Specifies the physical segment address.
,	Separates operands.
:	Specifies a label. Separates the usage type and symbol in EXTRN directives. Separates the physical segment address and offset in an address constant.
;	Starts a comment.
[ ]	Specifies register indirect addressing.
'	Specifies the start or end of a character constant.
"	Specifies the start or end of a string constant.

3.1.1.5 Operators

Operators consist of a single character or a combination of characters.

For further details on operator function, see Section 3.2.2 “Operators.”

The following operators are available.

(	)	.	!	~	::
+	-	*	/	%	
<<	>>				
<	<=	>	>=	==	!=
&	^	|	&&	||	
BYTE1	BYTE2	BYTE3	BYTE4	WORD1	WORD2

SEG	OFFSET	BPOS	SIZE
OVL_SEG	OVL_OFFSET	OVL_ADDRESS	

3.1.1.6 Escape Sequences

The tools in the MACU8 assembler package support the following C-like escape sequences.

Notation	Description
\nnn	An octal value expressed as one to three octal digits. This value must be between 0 and 377.
\xnn , \Xnn	A hexadecimal value expressed as one or two hexadecimal digits. This value must be between 0 and FFH.
\a	ASCII bell (07H)
\b	ASCII backspace (08H)
\f	ASCII form feed(0CH)
\n	ASCII line feed (0AH)
\r	ASCII carriage return (0DH)
\t	ASCII tab (09H)
\v	ASCII vertical tab (0BH)
char	If char is an ASCII character other than the above (a, b, f, n, r, t, and v), this is equivalent to the single-byte code for that character.

■ Examples ■

The following are examples of escape sequences. The second column gives the equivalent in hexadecimal.

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

3.1.1.7 Double-Byte Characters

Programs can use kanji and other double-byte characters. The assembler can handle both Shift JIS (SJIS) and EUC double-byte character encodings.

Adding the /KE or /KEUC option to the command line switches the assembler from the default Shift JIS (SJIS) encoding for double-byte characters to the EUC encoding. Both encodings assign 2-byte codes to Japanese “full width” characters, but use different sets of code points. The following is the assembler's “good enough for government work” definition of valid byte combinations.

	Shift JIS (SJIS) encoding	EUC code encoding (/KE or /KEUC specified)
First byte of double-byte character	81H to 9FH 0E0H to 0FCH	0B0H to 0F4H 0A1H to 0A8H
Second byte of double-byte character	40H to 0FCH	0A0H to 0FEH

Double-byte characters can only appear in the following source code positions.

1. String constants
2. Comments
3. Block comments

3.1.2 Constants

A constant is a numerical value, character, or string that retains an unchanging value throughout program execution. This assembler package recognizes the following type of constants: integer constants, address constants, character constants, and string constants.

The rest of this Section describes these types individually.

3.1.2.1 Integer Constants

■ Syntax ■

ddigits

*hdigits*H

*ddigits*D

*odigits*O

*odigits*Q

*bdigits*B

■ Description ■

An integer constant is a 32-bit integer expressed using binary, octal, decimal, or hexadecimal notation. A radix specifier following the numerical value specifies the number base for the integer constant. If there is no specifier, the number base defaults to decimal.

The fields *hdigits*, *ddigits*, *odigits*, and *bdigits* represent valid sequences of hexadecimal, decimal, octal, and binary digits, respectively. The first character must be a digit between 0 and 9, however, to distinguish the integer constant from a symbol. If a hexadecimal representation starts with a letter, therefore, prefix it with 0.

For program legibility, underscores (_) can be inserted into the text strings representing the integer constant—except for the very beginning.

The following summarizes the radix specifiers and characters used by each number base available.

Radix Specifier(s)	Description	Valid Digits
H, h	Hexadecimal	0123456789ABCDEFabcdef_
D, d	Decimal	0123456789_
O, o, Q, q	Octal	01234567_
B, b	Binary	01_

As the Table indicates, radix specifiers and hexadecimal constants are not case sensitive. They accept upper or lower case letters or both.

■ Example ■

The following shows some hexadecimal, decimal, octal, and binary representations for decimal 256.

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

Prefixing the integer constant's representation with any number of zeros has no effect whatsoever on the meaning (value) of the integer constant. The following are further examples of valid representations for decimal 256.

	Representation
Hexadecimal	00100H
Decimal	0256 00256D
Octal	000400O 00400Q
Binary	000100000000B

Finally, adding underscores (_) produces even more variants.

	Representation
Hexadecimal	1_ 00H 1_00_H
Binary	1_00000000B 1_0000_0000_B

3.1.2.2 Address Constants

■ Syntax ■

integer_constant1: *integer_constant2*

■ Description ■

An address constant is a direct representation of an address in an address space. It consists of two fields: a physical segment address and an offset. The first field, *integer_constant1*, is an integer constant

representing a physical segment address, and must have a value between 0 and 0FFH. The second field, *integer_constant2*, is an integer constant representing an offset, and must have a value between 0 and 7FFFFH.

The two fields are joined with a colon (:). Note that whitespace characters are not allowed before or after the colon.

By itself, an address constant does not specify the address space type. The assembler decides from the context—that is, the microcontroller instruction or assembler directive in which the address constant appears.

■ Example 1 ■

The following is an example of an address constant for the address at offset 1000H in physical segment address 2 in the DATA address space.

```
        L       R0, 2:1000H
D_ADR   DATA    2:1000H
```

■ Example 2 ■

The following is an example of an address constant for the address at offset 1000H in physical segment address 2 in the CODE address space.

```
        B       2:1000H
C_ADR   CODE    2:1000H
```

3.1.2.3 Character Constants

■ Syntax ■

'*char*'

■ Description ■

A character constant is converted to the single-byte code for the specified character. One possibility for the char field is a character with a single-byte code; another, an escape sequence representing such a code.

Finally, if there is no character and escape sequence, the character constant defaults to the value 0H.

Double-byte characters are not allowed because their codes require two bytes.

■ Example ■

The following are examples of character constants. The second column gives the equivalent in hexadecimal.

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

3.1.2.4 String Constants

■ Syntax ■

"*characters*"

■ Description ■

A string constant, used for such purposes as initializing code memory, is a sequence of characters enclosed in double quotation marks (“). The characters field specifies this sequence, which can contain characters with single-byte codes, double-byte characters, and escape sequences for either. The only restriction is that the total length of the equivalent hexadecimal codes must not exceed 255 bytes.

■ Example ■

The following is an example of a DB directive using a string constant as an operand. The comments give the equivalent hexadecimal codes.

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

3.1.3 Symbols

Symbol are names representing the following.

1. Numerical values
2. Addresses
3. Relocatable segments
4. Instructions
5. Directives
6. Registers
7. Register addresses
8. Operators
9. Addressing types
10. Special operands for instructions
11. Special operands for directives
12. Macros

There are two types of symbols: user symbols defined by the programmer and reserved words predefined by the assembler package. Symbols representing numerical values and address come in both types. Those representing relocatable segments and macros are always user symbols. The rest are all reserved words.

A symbol is a sequence of 1 to 32 characters from the following set: letters, digits, underscore (_), question mark (?), and dollar sign ($). To distinguish symbols from integer constants, however, the first character cannot be a digit. Any characters beyond the first 32 are ignored. A symbol can be defined only once per source code file—unless SET directives are used for all. Attempting to redefine a symbol within the same source code file produces an error message.

3.1.3.1 User Symbols

A user symbol is one defined by the programmer in the program. A reserved word cannot redefined as a user symbol.

For user symbols, the /CD and /NCD options specify whether the assembler distinguishes between upper and lower case in the letters making up the symbol. Specifying the /CD option, the default, distinguishes between upper and lower case—that is, two symbols are the same only if they agree in both spelling and case. Specifying the /NCD option symbol, on the other hand, forces all letters to upper case, so that only the spelling need match.

User symbols are defined as labels, with a command line option, or with a directive. The following summarizes the methods for defining each type of user symbol.

User Symbol	Definition Method(s)
User symbol representing a numerical value	EQU, SET, and EXTRN directives
User symbol representing an address	Label or one of the following directives: EQU, SET, CODE, TABLE, TBIT, DATA, BIT, NVDATA, NVBIT, COMM, and EXTRN
User symbol representing a relocatable segment	SEGMENT directive
User symbol representing a macro	DEFINE directive or the assembler's /DEF command line option

Symbol definitions associate text strings with symbols representing macros and values with all others. Symbols representing numerical values have as their value that number; ones representing addresses, that address; ones representing relocatable segments, the starting address for the region to which that relocatable segment is assigned.

■ Example 1 ■

The following are examples of definitions for user symbols representing numerical values.

These examples define SYMEQU1, SYMSET1, and SYM_EXT_NUM1 as user symbols representing numerical values.

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

■ Example 2 ■

The following are examples of definitions for user symbols representing addresses.

```
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
```

These examples define EXTINT0, EXTINT1, SYMDAT1, SYMBIT1, SYMSET2, SYM_COMM_DAT1, and SYM_EXT_BIT1 as user symbols representing addresses.

■ **Example 3** ■

The following shows examples of definitions for user symbols representing relocatable segments.

This example defines MAINCOD, TABLE1, and COMBUF1 as user symbols representing relocatable segments.

```
MAINCOD       SEGMENT       CODE #0
       RSEG   MAINCOD
       .
       .
       .
TABLE1 SEGMENT       TABLE #1
       RSEG   TABLE1
       DW     0000H
       DW     0001H
       .
       .
       .
DATBUF1       SEGMENT       DATA 2 ANY
       RSEG   DATBUF1
       DS     2
```

■ **Example 4** ■

The following is an example of a definition for a user symbol representing a macro.

This example defines MCRSYM as a user symbol representing a macro.

```
DEFINE MCRSYM "MACRO BODY"
```

The rest of this Section divides user symbols into types based on the program contexts in which symbols of that type can appear.

Note, however, that the following discussion ignores symbols representing macros.

The assembler divides user symbols into absolute and relocatable based on the tool that fixes the value. An absolute symbol is one for which the assembler fixes the value. Otherwise, the user symbol becomes a relocatable symbol, and the linker decides its value.

3.1.3.1.1 Absolute Symbols

Absolute symbols represent numerical values or addresses. They are defined with the following methods.

1. Local symbol definition directives (EQU, SET, CODE, DATA, BIT, NVDATA, NVBIT, TABLE, and TBIT) with a constant expression as the operand
2. Labels definitions inside absolute segments

■ Example ■

The following gives examples of absolute symbol definitions.

This example defines SYMCOD, SYMDAT, SYMBIT, SYMEQU, SYMSET, DATALABEL, and CODELABEL as 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
```

3.1.3.1.2 Relocatable Symbols

Relocatable symbols have the following types.

1. Simple relocatable symbols
2. Segment symbols
3. Communal symbols
4. External symbols

This Section describes each type in turn.

(1) Simple Relocatable Symbols

Simple relocatable symbols are symbols representing the addresses of relocatable segments within the same source code file. Note, however, that symbols representing relocatable segments (segment symbols) are not simple relocatable symbols.

Simple relocatable symbols belong to relocatable segments and are defined with the following methods.

1. Labels definitions inside relocatable segments. Such symbols belong to the relocatable segment in which the label is defined.
2. Local symbol definition directives (EQU, SET, CODE, DATA, BIT, NVDATA, NVBIT, TABLE, and TBIT) with an expression using a simple relocatable symbol as the operand. Such symbols belong to the same relocatable segment as the simple relocatable symbol.

■ **Example** ■

The following shows examples of definitions for simple relocatable symbols.

```
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
```

This example defines the simple relocatable symbols LBUF, HBUF, START, SIMCOD, SIMDAT, SIMBIT, SIMEQU, and SIMSET. DATSEG and CODSEG are symbols representing relocatable segments, so are not simple relocatable symbols.

(2) Segment Symbols

Segment symbols are symbols representing relocatable segments. They are defined with the SEGMENT directive. Using a segment symbol as the operand in an RSEG directive defines a relocatable segment with the segment symbol as its name. For this reason, segment symbols are sometimes called segment names.

Segment symbols have as their value the starting address of the region to which the relocatable segment is assigned. To reference a segment symbol in another source code file, the current file must define that segment symbol with a SEGMENT directive.

■ **Example** ■

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

This example defines the segment symbols CHARBUF and SUB1.

(3) Communal Symbols

Communal symbols represent the starting addresses of data regions shared by multiple source code files.

The linker takes the largest of the size operands from these multiple communal symbol definitions, reserves that amount of memory, and sets the communal symbol value to the starting address of that region.

If the communal symbol is defined in only one file, the size operand from that sole definition is the largest, so the linker reserves that amount of memory, and sets the communal symbol value to the starting address of that region.

Communal symbols are defined with the COMM directive.

■ Example ■

The following is an example of a communal symbol definition.

```
COMMSYM       COMM   DATA 10H
```

This example defines the communal symbol COMMSYM for an area in the DATA address space. The size of this area is a minimum of 10H bytes.

(4) External Symbols

External symbols reference public and communal symbols defined in other source code files.

External communal symbols are defined with the EXTRN COMM directive.

For further details on public symbols, see Section 3.1.3.2 “Local vs. Public Symbols” below.

■ Example ■

The following are examples of external symbol definitions.

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

These examples define EXTSYM1 and EXTSYM2 as external symbols representing data addresses, EXTSYM3 as an external symbol representing a numerical value, and EXTSYM4 as an external symbol representing a code address.

3.1.3.2 Local vs. Public Symbols

In the absence of any declaration to the contrary, absolute and simple relocatable symbols can only be referenced in the source code file in which they are defined. For this reason, they are called local symbols.

The PUBLIC directive converts local symbols into public symbols that can be referenced from other source code files as well.

Limiting the lexical scope of local symbols to the source code files in which they are defined is important to keeping those files—and the functionality in each—independent. Different files can thus use the same names for local symbols and not worry about overlap. Otherwise, name management would become an administrative nightmare.

■ Example ■

The following gives an example of a public symbol definition.

```
PUBLIC  SYMEQU  DATABUF1     ; Make symbol public

SYMEQU  EQU     1

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

This example defines SYMEQU as an absolute symbol and DATABUF1 as a simple relocatable symbol, so the two symbols would both normally be local symbols. This example, however, makes them public with the PUBLIC directive.

3.1.3.3 Referencing User Symbols

User symbols, once defined, can be referenced in the operands in instructions and directives in the same source code file.

There are two types of references: backward references to symbols defined earlier in the source code file and forward references in the opposite direction.

Operands in microcontroller instructions can reference user symbols with either backward or forward references. Those in directives can always use backward references, but forward references are sometimes not allowed. For further details on these restrictions, see the individual directive descriptions in Section 5 “Detailed Directive Descriptions.”

■ **Example** ■

This example uses both backward and forward references in the operands for two microcontroller instructions (B and MOV) and two directives (DW and ORG).

```
        MOV     ER0, #FORWARD_SYM
        DW      FORWARD_VALUE

BACKWARD_VALUE  EQU  10H
BACKWARD_SYM    EQU  28H
BACKWARD_LABEL:
        B       FORWARD_LABEL
        ORG     FORWARD_VALUE       ; error

        ORG     BACKWARD_VALUE
        DW      BACKWARD_VALUE
        MOV     ER0, #BACKWARD_SYM
        B       BACKWARD_LABEL
FORWARD_LABEL:
FORWARD_SYM   EQU    30H
FORWARD_VALUE EQU    20H
```

The backward reference symbols are BACKWARD_VALUE, BACKWARD_SYM, and BACKWARD_LABEL; the forward reference ones, FORWARD_LABEL, FORWARD_SYM, and FORWARD_VALUE. Operands in all microcontroller instructions and the DW directive can reference user symbols with either backward or forward references. The ORG directive, however, does not allow forward references in its operand. The ORG FORWARD_VALUE line in the source code therefore produces an error message.

3.1.3.4 Referencing User Symbols in Multiple Source Code Files

Referencing the same user symbol from multiple source code files uses the following symbol types.

1. Public symbols
2. External symbols
3. Communal symbols
4. Segment symbols

For further details on the procedures for using these symbols, see Section 5.9 “Linkage Control Directives.”

3.1.3.5 Macro Symbols

Macro symbols are markedly different from other user symbols. First, they have text strings in their definitions, not values as is normally the case with other user symbols. Secondly, they cannot be made public for referencing them from a source code file other than the one in which they are defined. The biggest difference, however, is that the significant part from the assembly language standpoint is not the macro symbol itself, but the text string associated with it.

For these reasons, therefore, this manual clearly distinguishes macro symbols from other user symbols. When it talks about user symbols, it almost always means user symbols other than macro symbols.

3.1.3.6 Reserved Words

Reserved words are symbols predefined by the assembler package. They consist of the following types.

1. Instructions
2. Directives
3. Registers
4. Operators
5. SFR symbols
6. Addressing specifiers
7. Special operands for instructions
8. Special operands for directives

SFR symbols are similar to user symbols in that the /CD and /NCD options specify whether the assembler distinguishes between upper and lower case in the letters making up the symbol. All other reserved words are case insensitive.

Certain reserved words are polymorphic—that is, have different meanings depending on the syntactic context.

Certain reserved words are defined only for microcontrollers with a specific core. For other cores, the symbols are available for definition as user symbols.

Appendix B “Reserved Word List” lists all reserved words other than SFR symbols as well as their applications and applicable CPU cores.

The rest of this Section describes reserved words by the above types.

3.1.3.6.1 Instructions

These reserved words represent the microcontroller instructions recognized by the assembler.

For further details on instruction functions, refer to the related documents.

3.1.3.6.2 Directives

These reserved words represent the directives recognized by the assembler.

For further details on directive functions, see Section 5 “Detailed Directive Descriptions.”

3.1.3.6.3 Registers

These reserved words represent the registers recognized by the assembler as operands in microcontroller instructions.

For further details on using registers in operands, see Section 4.1 “Addressing Syntax.”

The following reserved words represent registers.

R0	R1	R2	R3	R4	R5	R6	R7
R8	R9	R10	R11	R12	R13	R14	R15
ER0	ER2	ER4	ER6	ER8	ER10	ER12	ER14
XR0	XR4	XR8	XR12	QR0	QR8		
CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7
CR8	CR9	CR10	CR11	CR12	CR13	CR14	CR15
CER0	CER2	CER4	CER6	CER8	CER10	CER12	CER14
CXR0	CXR4	CXR8	CXR12	CQR0	CQR8		
BP	DSR	EA	ECSR	ELR	EPSW	FP	LR
SP	PC	PSW					

3.1.3.6.4 Operators

These reserved words represent operators used in expressions.

For further details on operator function and usage, see Section 3.2.2 “Operators.”

The following reserved words represent operators.

BYTE1	BYTE2	BYTE3	BYTE4	WORD1	WORD2
SEG	OFFSET	BPOS	SIZE		
OVL_SEG	OVL_OFFSET	OVL_ADDRESS			

3.1.3.6.5 SFR Symbols

These reserved words represent addresses specific to individual target microcontrollers—names for registers in the SFR region and names for bits in those registers, in particular. They are defined in DCL files. The assembler treats them as absolute symbols.

■ Example ■

The following source code fragment references the SFR symbols P0CON0 and P00 in configuring the

port 0.0 pin for high-impedance input and then testing the input from that pin.

```
L       R0, P0CON0
OR      R0, #03H
ST      R0, P0CON0
BTST    P00
```

SFR symbols differ from other reserved words in that the /CD and /NCD options specify whether the assembler distinguishes between upper and lower case in the letters making up the symbol. Specifying the /CD option, the default, distinguishes between upper and lower case—that is, two symbols are the same only if they agree in both spelling and case. Specifying the /NCD option symbol, on the other hand, forces all letters to upper case, so that only the spelling need match.

3.1.3.6.6 Addressing Specifiers

These reserved words control memory access. For further details on operands using addressing specifiers, see Section 4.1.3 “Data Memory Addressing.”

The following reserved words represent addressing specifiers.

```
NEAR        FAR
```

3.1.3.6.7 Special Operands for Instructions

These reserved words represent such things as branch conditions in conditional branch instructions and PSW flag names in instructions, so have meanings completely different from addressing.

For further details on their meanings and the procedures for using them, refer to the U8 Core Instruction Manual.

The following reserved words represent special operands for instructions.

```
GT          GE          LT          LE          PS          NS
EQ          NE          ZF          NZ          CY          NC
GTS         GES         LTS         LES         OV          NV
AL
```

3.1.3.6.8 Special Operands for Directives

These reserved words impart special meanings to directive operands.

For further details on their meanings and the procedures for using them, see Section 5 “Detailed Directive Descriptions.”

```
CODE        DATA        BIT         NVDATA      NVBIT       TABLE
DYNAMIC     NVRAM       UNIT        WORD        DUP
NEAR        FAR         SMALL       LARGE
```

3.1.4 Location Counter Symbol

■ Syntax ■

$

■ Description ■

The location counter symbol references the current location counter contents for the logical segment to which the source code belongs.

For further details on the location counter, see Section 2.5 “Location Counters.”

3.2 Operators and Expressions

Operands in instructions and directives can use expressions—that is, constants, addresses, symbols representing numerical values, etc. joined together with operators. The assembler evaluates an expression in the source code to a value. The programmer can thus code operands as expressions reflecting the true meaning behind their values instead of just specifying the values themselves.

This Chapter starts by discussing attributes and values, two basic concepts behind expressions in assembly language. It then moves on to functional descriptions of the operator types available with the assembler, classification of expressions into types, and expression evaluation.

3.2.1 Basic Concepts

3.2.1.1 Why Expressions Have Attributes

Constants, symbols, and other elements representing values have various attributes. Labels have attributes indicating where they appear in the source code—whether they represent an address in the CODE address space and whether they are in a relocatable segment, for example.

The assembler not only divides the values resulting from expression evaluation into the same numeric and address types as constants and symbols, but also assigns them a usage type, an attribute indicating the intended purpose for a value. These attributes depend on such factors as the operator types appearing in the expression and the types for the constants, symbols, and other operands. In other words, the assembler manages not only a value for the expression, but also a meaning.

The above description is all very theoretical and vague, so here are some actual examples to illustrate how simple this concept is—and how natural.

■ Example 1 ■

```
L       ER0, TBL+2
```

If TBL is an address in the DATA address space, the expression TBL+2 in this example represents the address two bytes past that address. We can also say that it represents an address in the DATA address space.

■ Example 2 ■

```
MOV     R0, #END_ADR-START_ADR
```

If START_ADR and END_ADR are the starting and end addresses for a buffer in the DATA address space, the expression END_ADR-START_ADR in this example represents the size of that buffer. Although the operands are both addresses, the result represents a numerical value.

■ **Example 3** ■

```
SB      D_ADR.4
```

If D_ADR is an address in the DATA address space, the expression D_ADR.4 in this example represents bit 4 in the byte at that address. We can also say that the expression represents an address in the BIT address space.

The above examples should illustrate the following two reasons for giving expressions attributes.

1. They allow the assembler to check whether expressions appearing in the operands of instructions and directives are being used properly.
2. They allow the assembler to check whether expressions make sense.

The first is particularly important because the assembler can then issue a warning message to flag operands when they appear in instruction and directive contexts where they normally should not and thus signal to the programmer that there may be a typo or other error.

The following are examples of such suspect expressions.

■ **Example 4** ■

```
L       ER0, CODE_ADR
```

The second operand in an L instruction must be an address in the data memory space, so CODE_ADR is clearly being used incorrectly if it represents an address in the code memory space. The assembler therefore flags this source code with a warning message.

■ **Example 5** ■

```
MOV     R0, END_ADR+START_ADR
```

If END_ADR and START_ADR are both addresses in the data memory space, the expression END_ADR+START_ADR in this example represents the sum of two addresses, an arithmetic operation producing a meaningless result. Although it is theoretically possible that a programmer might deliberately write this sort of expression, we can also say that it is far more likely that the expression contains an error. The assembler therefore issues a warning message here as well.

The above should illustrate how important assigning attributes to expressions and imposing certain limits on expression syntax can be to boosting program safety and reliability.

3.2.1.2 Constant vs. Relocatable Expressions

Constant expressions can be fully evaluated because they consist solely of integer constants, absolute symbols, and other elements whose values are known to the assembler. They have all operators available to them. Relocatable expressions, in contrast, contain one or more relocatable symbols, so, in general, can only be partially evaluated. The assembler therefore sends descriptions of these relocatable expressions to the object file for the linker to resolve.

The following are examples of relocatable expressions.

■ Example 1 ■

```
        EXTRN  DATA:GL_TBL
        L      ER0, GL_TBL+2
```

This example adds to the external symbol GL_TBL. The assembler does not know the value of GL_TBL, so cannot evaluate the expression GL_TBL+2.

■ Example 2 ■

```
GL_TBL COMM   DATA 10H
        SB     GL_TBL.4
```

This example references bit 4 in an address in the data memory space indicated by the communal symbol GL_TBL. As was the case with the first example, the assembler does not know the value of GL_TBL, so cannot evaluate the expression GL_TBL.4.

These relocatable symbols can only use a strictly limited number of arithmetic operators and there are additional syntactic restrictions as well.

For further details on relocatable expression syntax and restrictions, see Section 3.2.3 “Expression Types.”

3.2.2 Operators

This Section describes the function of the operators available to the assembly language programmer.

The assembler provides the following types of operators.

1. Arithmetic operators
2. Logic operators
3. Bit operators
4. Relational operators
5. Dot operator
6. Special operators

There are unary operators and binary ones. Unary operators have a single operand expression on their right side; and binary operators, one on both sides.

In the following description, TRUE is any numerical value other than 0; FALSE, 0.

The dummies *expression*, *expression1*, and *expression2* used in syntax descriptions represent expressions.

3.2.2.1 Arithmetic Operators

These operators are for general arithmetic.

Operator	Syntax	Description
+	*expression1* + *expression2*	Addition (binary plus)
	+ *expression1*	Unary plus
-	*expression1* - *expression2*	Subtraction (binary minus)
	- *expression1*	Negation (unary minus)
*	*expression1* * *expression2*	Multiplication
/	*expression1* / *expression2*	Division
%	*expression1* % *expression2*	Modulo (remainder after dividing *expression1* by *expression2*)

■ Examples ■

```
VALUE  EQU    30H
       ADD    R0, #VALUE+1
BUFSIZE       EQU    1024H
       DS     BUFSIZE*4
```

This example uses the addition (binary plus) operator in an operand for the ADD instruction and the multiplication operator in the operand for the DS directive.

3.2.2.2 Logical Operators

These operators derive a truth value from the two operand expressions on their left and right sides or a single operand expression on their right sides.

Operator	Syntax	Description
&&	*expression1* && *expression2*	1 if both operands are TRUE; 0 otherwise.
||	*expression1* || *expression2*	1 if either operand is TRUE; 0 otherwise.
!	!*expression*	0 if the operand is TRUE; 1 otherwise.

■ **Example** ■

```
SW1  EQU  0
SW2  EQU  2

IF  SW1&&SW2
   BUSIZE   EQU  1024
ELSE
   BUSIZE   EQU  2048
ENDIF
```

This example uses the logical AND operator (&&) in the operand of an IF directive for conditional assembly.

3.2.2.3 Bitwise Operators

These operators perform logic arithmetic on the bits in expression results.

Operator	Syntax	Description
&	*expression1* & *expression2*	Bitwise AND
|	*expression1* | *expression2*	Bitwise OR
^	*expression1* ^ *expression2*	Bitwise exclusive OR
<<	*expression1* << *expression2*	Left shift of *expression1* by the number of bit positions specified by *expression2*, filling vacated bits at the lower end with zero
>>	*expression1* >> *expression2*	Right shift of *expression1* by the number of bit positions specified by *expression2*, filling vacated bits at the upper end with zero
~	~ *expression*	One's complement

■ **Example** ■

```
SCB_MSK       EQU    1111_1000B
BCB_MSK       EQU    1100_1111B
       AND    R0, #BCB_MSK & SCB_MSK
```

This example uses the bitwise AND operator (&) in an operand for an AND instruction.

3.2.2.4 Relational Operators

These operators compare the values of two expressions yielding 1 if the condition is satisfied and 0 if it

is not.

Relational operators are limited to use with constant expressions.

Comparing two expressions representing addresses compares the physical segment address as well.

Operator	Syntax	Description
>	*expression1* > *expression2*	1 if *expression1* is greater *expression2*; 0 otherwise.
>=	*expression1* >= *expression2*	1 if *expression1* is greater than or equal to *expression2*; 0 otherwise.
<=	*expression1* < *expression2*	1 if *expression1* is smaller than *expression2*; 0 otherwise.
<=	*expression1* <= *expression2*	1 if *expression1* is smaller than or equal to *expression2*; 0 otherwise.
==	*expression1* == *expression2*	1 if *expression1* is equal to *expression2*; 0 otherwise.
!=	*expression1* != *expression2*	1 if *expression1* is not equal to *expression2*; 0 otherwise.

■ Example ■

```
IF      VALUE1 >= VALUE2
        .
        .
        .
ENDIF
```

This example uses the greater than or equal to (>=) operator in the operand of an IF directive for conditional assembly.

3.2.2.5 Dot Operator

This operator calculates a bit address from a data address and a bit offset.

■ Syntax ■

expression1.*expression2*

■ Descripton ■

The *expression1* field specifies the data address; the *expression2* field, the bit offset. The above syntax produces the same value as the following expression.

((*expression1* << 3) + *expression2*)

The assembler treats the dot operator (.) in much the same way as the arithmetic operators. In particular, note that it does not apply range checking to either operand.

■ Example ■

The following is an example using the dot operator.

This example defines user symbols DATSYM1 and DATSYM2 with usage type DATA and user symbol EXTNUM1 with usage type NUMBER and then uses them in the operands for SB instructions to access bit 0 in the corresponding bytes in data memory.

```
DATSYM1       DATA   8000H
EXTRN  NUMBER:EXTNUM1

       DSEG   #0 AT 8800H
DATSYM2:
       DS     1

       CSEG
       SB     DATSYM1.0
       SB     DATSYM2.0
       SB     EXTNUM1.0
```

3.2.2.6 Address Operator

This operator calculates an address from a physical segment address and an offset.

■ Syntax ■

expression1::*expression2*

■ Description ■

The *expression1* field specifies the physical segment address. If it is of type address instead NUMBER, only the physical segment address portion is used.

The *expression2* field specifies the offset. If it is of type address instead NUMBER, only the offset portion is used.

■ Example ■

The following gives examples of :: operator usage. The first L instruction accesses offset FFFFH in the same physical segment as DATSYM; the second, the same offset as DATSYM in physical segment address #1.

```
        DSEG
DATASYM:        DS      10H

        CSEG
        L       R0,   DATASYM::0FFFFH
        L       R1,   1::DATASYM
```

3.2.2.7 Special Operators

These operators perform such operations as extracting bit sequences for their operand expressions and determining the address to which a segment symbol is actually assigned.

The following special operators are available.

Operator	Syntax	Description
BYTE1	BYTE1 *expression*	Equivalent to (*expression* & 0FFH)
BYTE2	BYTE2 *expression*	Equivalent to ((*expression* >> 8) & 0FFH)
BYTE3	BYTE3 *expression*	Equivalent to ((*expression* >> 16) & 0FFH)
BYTE4	BYTE4 *expression*	Equivalent to ((*expression* >> 24) & 0FFH)
WORD1	WORD1 *expression*	Equivalent to ((*expression*) & 0FFFFH)
WORD2	WORD2 *expression*	Equivalent to ((*expression* >> 16)& 0FFFFH)
SEG	SEG *expression*	Physical segment address portion of *expression* of type address
OFFSET	OFFSET *expression*	Offset portion of *expression* of type address
SIZE	SIZE *segment_symbol*	Size of segment *segment_symbol*
BPOS	BPOS *expression*	Bit offset (lowest three bits) from bit *expression* of type address
OVL_SEG	OVL_SEG *segment_symbol*	Physical segment address portion of actual address for *segment_symbol*
OVL_OFFSET	OVL_OFFSET *segment_symbol*	Offset portion of actual address for *segment_symbol*
OVL_ADDRESS	OVL_ADDRESS *segment_symbol*	Actual starting address *segment_symbol*

Unlike other operators, which are generally found in most assemblers, these special operators are for use in effectively exploiting unique features of the nX-U8 instruction set. This tight link to the hardware imposes certain limitations on their operand expressions.

The operators BYTE1 to BYTE4, for example, extract 8-bit portions of the specified 32-bit value. The operators WORD1 and WORD2 perform a similar function at the 16-bit level. Applying these operators to expressions of type address, however, produces a warning message.

■ Example 1 ■

The following gives examples using the operators BYTE1 to BYTE4, WORD1, and WORD2.

```
VALUE  EQU    12345678H

       CSEG
       MOV    R0, #BYTE1 VALUE           ;78H
       MOV    R1, #BYTE2 VALUE           ;56H
       MOV    R2, #BYTE3 VALUE           ;34H
       MOV    R3, #BYTE4 VALUE           ;12H

       TSEG
       DW     WORD1 VALUE          ;5678H
       DW     WORD2 VALUE          ;1234H
```

The SEG operator extracts the physical segment number from an expression of type address, so cannot be used with expressions of type NUMBER.

The OFFSET operator extracts the offset from an expression of type address, converting an address to a numerical value. Applying this operator to an expression that is already a numerical value, however, produces a warning message.

■ Example 2 ■

This example uses the SEG and OFFSET operators to obtain the physical segment address and segment starting address for the relocatable segment DATA_SEG.

```
DATA_SEG      SEGMENT       DATA
       MOV    R2,  #SEG DATA_SEG         ; Physical segment address
       MOV    ER0, #OFFSET DATA_SEG      ; Offset
```

The BPOS operator extracts the bit offset from a bit address, providing access to a particular bit position in a contiguous data region.

expression & 7

The operator has the same effect as using the following expression. Applying this operator to an expression that does not have usage type BIT, NVBIT, or TBIT produces a warning message.

■ Example 3 ■

This example uses the BPOS operator to access the same bit position as FLG1 in the byte D_TBL.

```
       BSEG
FLG1:  DBIT   1

       DSEG #0
D_TBL: DS     10H

       CSEG
       MOV    R10, #BYTE1 D_TBL
       MOV    R11, #BYTE2 D_TBL
LOOP:
       L      R0, [ER10]
       SB     R0. (BPOS FLG1)      ; Get bit position
       .
       .
       .
```

The SIZE operator returns the size of the specified relocatable segment, so can only be used with segment symbols.

■ Example 4 ■

This example uses the SIZE operator to determine the size of the relocatable segment DATA_SEG.

```
DATA_SEG      SEGMENT       DATA WORD
       DW     SIZE DATA_SEG ; Get segment size
```

The OVL_SEG, OVL_OFFSET, and OVL_ADDRESS operators extract portions of the address at which the overlay function saves the specified segment symbol, so can only be used with segment symbols.

■ Example 5 ■

This example uses the OVL_SEG and OVL_OFFSET operators to load the address at which the program saves a segment into QR0.

```
        CSEG
        MOV     R8, #OVL_SEG CODE_SEG2
        LEA     OVL_OFFSET CODE_SEG2
LOOP2:
        L       QR0, R8:[EA+]
        .
        .
        .
CODE_SEG2 SEGMENT    CODE NVRAM
        RSEG    CODE_SEG2
START2:
        .
        .
        .
```

■ Example 6 ■

This example uses the OVL_ADDRESS operator to load the address at which the program saves a segment into ER2.

```
        CSEG
        MOV     ER0, #0
LOOP1:
        L       ER2, OVL_ADDRESS CODE_SEG1[ER0]
        .
        .
        .
CODE_SEG1 SEGMENT    CODE NVRAM
        RSEG    CODE_SEG1
START1:
        .
        .
        .
```

3.2.3 Expression Types

The assembler divides expressions into the following three broad categories based on its ability to determine the final value.

1. Constant expressions
2. Relocatable expressions
3. Simple relocatable expressions

Constant expressions are ones that can be fully evaluated by the assembler. Relocatable expressions, in contrast, must await evaluation by the linker. They can, within the limits imposed by the syntax, use all relocatable symbols.

Simple relocatable expressions are relocatable expressions that evaluate to a known offset from a

segment base. The only relocatable symbols that they can use are simple relocatable symbols.

■ Example ■

```
ABS_DATA      DATA   1000H
EXTRN  DATA:EXT_DATA

DATA_SEG      SEGMENT       DATA WORD
       RSEG   DATA_SEG
D_TBL: DS     10H

       CSEG
       L      R0, ABS_DATA+10H
       L      R1, EXT_DATA+10H
       L      R2, D_TBL+10H
```

This example looks at the expressions forming the second operands in the three L instructions.

The first, ABS_DATA+10H, is a constant expression. The assembler knows that the value of ABS_DATA is 1000H, so fully evaluates the expression to 1010H.

The second, EXT_DATA+10H, is a relocatable expression. The assembler does not know the value of the external symbol representing the address EXT_DATA, so cannot perform the addition.

The third, D_TBL+10H, is a simple relocatable expression. The assembler does not have a value for D_TBL, but knows that the address is at offset 0 relative to the segment base DATA_SEG and thus that the expression evaluates, at least in this program, to an offset of 10H.

The reason for distinguishing these three types has to do with the limits that instructions and directives impose on expressions in their operands. Expressions fall into three types in terms of the freedom with which they fit into these contexts.

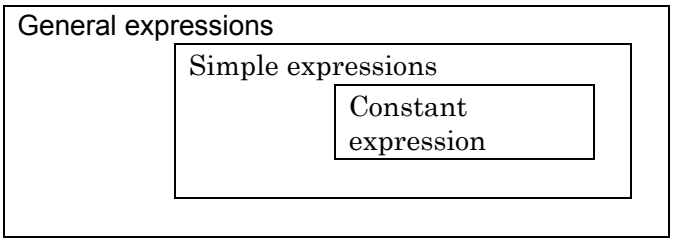
1. Constant expressions
2. Simple expressions
3. General expressions

Constant expressions, as already mentioned above, are ones that can be fully evaluated by the assembler. They have the least freedom.

Simple expressions add simple relocatable expressions to constant expressions, so evaluate to at least a known offset from a segment base.

General expressions cover constant expressions and relocatable ones. In other words, they represent all syntactically possible expressions, so thus have the greatest freedom.

The following illustrates the relationships between constant expressions, simple expressions, and general expressions.



Basic instructions by and large accept general expressions for their operands. Directives, on the other hand, restrict most operands to constant expressions or, at best, simple expressions. The reasons for these limitations lie in the way that the assembler processes each particular directive. Specific examples appear in Section 3.2.3.4 “Restrictions on Expression.”

For further details on the expressions that can appear in instruction and directive operands, see Sections 4 “Addressing and Instruction Types” and Section 5 “Detailed Directive Descriptions,” respectively.

The rest of this Section describes these three expression types individually.

3.2.3.1 Constant Expressions

This group consists of expressions that can be fully evaluated by the assembler—that is, integer constants, character constants, absolute symbols, and, within absolute segments, the location counter symbol as well as combinations thereof with operators. They have all operators available to them.

Constant expressions generally cannot contain relocatable symbols for the simple reason that the assembler must know their final values. There are, however, exceptions where expressions can contain relocatable symbols and still qualify as constant expressions.

1. Subtraction involving two simple relocatable symbols in the same relocatable segment
2. Subtraction involving two simple relocatable symbols with the dot operator (.) and in the same relocatable segment
3. An expression applying first the dot operator (.) to a relocatable symbol and then the BPOS operator to the result
4. An expression applying the SEG operator to a relocatable symbol with a physical segment attribute other than ANY

■ Example ■

The following is an example using a variety of constant expressions.

```
ABSSYM1   EQU      100H

DATSEG4   SEGMENT  DATA #2
          RSEG     DATSEG4
LABEL1:
          DS       2
LABEL2:
          TSEG
          DW       100H                        ;100H
          DW       'A'                         ;41H
          DW       ABSSYM1                     ;100H
          DW       1+2                         ;3H
          DW       +(1+2*ABSSYM1)              ;201H
          DW       100H*'A'                    ;4100H
          DW       (LABEL2-LABEL1)             ;2H
          DW       (LABEL2.0-LABEL1.0)         ;10H
          DW       BPOS(LABEL1.3)              ;3H
          DW       SEG LABEL2                  ;2H
```

The DW directive operands in the above code fragment are all constant expressions. The comments give the resulting values.

3.2.3.2 Simple Expressions

This group consists of expressions that do not contain any relocatable symbols other than simple relocatable symbols. Note that the location counter symbol qualifies as a simple relocatable symbol within absolute segments. This group includes constant expressions.

The following is the syntax for simple expressions.

Expression	Definition
Simple expression	constant expression
	| simple relocatable expression
Simple relocatable expression	Simple relocatable symbol
	| (simple relocatable expression)
	| + simple relocatable expression
	| simple relocatable expression + constant expression
	| constant expression + simple relocatable expression
	| simple relocatable expression – constant expression
	| simple relocatable expression.constant expression
	| OFFSET simple relocatable expression

The vertical bars (|) in the above syntax definition indicate a list of mutually exclusive choices, one of which must always appear.

The dot and OFFSET operators cannot be applied to a simple expression that is not a constant expression and contains the dot operator.

The assembler cannot fully evaluate a simple expression that contains a simple relocatable expression. The linker decides the final value.

■ Example ■

The following are examples of simple expressions that are not constant expressions.

```
DATSEG5         SEGMENT         DATA
        RSEG    DATSEG5
        DS      2
LABEL3:
        ORG     LABEL3
```

The following are examples of simple expressions using the symbols defined above.

```
        (LABEL3)
        +LABEL3
        LABEL3-1
        LABEL3.4
        OFFSET LABEL3
```

3.2.3.3 General Expressions

This group adds expressions containing segment, external, or communal symbols to simple expressions.

The following is the syntax for general expressions.

Expression	Definition
General expression	constant expression
	| relocatable expression
	| relocatable arithmetic 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 arithmetic expression	BYTE1 relocatable expression
	| BYTE2 relocatable expression
	| BYTE3 relocatable expression
	| BYTE4 relocatable expression
	| WORD1 relocatable expression
	| WORD2 relocatable expression
	| SEG relocatable expression
	| BPOS relocatable expression
	| SIZE relocatable expression
	| OVL_SEG segment symbol
	| OVL_OFFSET segment symbol
	| OVL_ADDRESS segment symbol
	| (relocatable arithmetic expression)

The vertical bars (|) in the above syntax definition indicate a list of mutually exclusive choices, one of

which must always appear.

The syntax for relocatable expressions follows for general expressions.

The dot and special operators cannot be applied to a relocatable expression that contains the dot operator.

The assembler cannot evaluate a relocatable expression. The linker decides the final value.

■ Examples ■

The following are examples of general expressions that are neither constant expressions nor simple expressions.

```
SEGSYM SEGMENT DATA
COMMSYM COMM DATA 2
EXTRN  DATA:EXTSYM BIT:BITSYM
```

The following are examples of general expressions using the symbols defined above.

```
EXTSYM
(COMMSYM)
+EXTSYM
COMMSYM-1
EXTSYM.1
(EXTSYM+1).1
EXTSYM.0+10.0
HIGH EXTSYM
LOW(COMMSYM)
SEG (+EXTSYM)
OFFSET (COMMSYM-1)
PAGE SEGSYM
LREG (COMMSYM)
BPOS BITSYM
SIZE SEGSYM
```

3.2.3.4 Restrictions on Expression

There are restrictions on the positions in which each expression type can appear and on the use of forward references in user symbols in expressions.

These restrictions fall into the following types.

(1) ORG directive restrictions on operands

In an absolute segment, the ORG directive operand must be a constant expression so that the assembler can determine the address for the operand.

Relocatable segments, in contrast, allow simple expressions in ORG directive operands. Note, however,

that any simple relocatable symbols in the simple expression must belong to the current relocatable segment. The reason is that the assembler must determine at least an address relative to the segment if not the full address for the relocatable segment. Determining this relative address determines the size of the relocatable segment, which the linker subsequently uses to assign the logical segment to memory.

■ Example ■

```
        TYPE(M610001)
XCODSEG         SEGMENT         CODE
        RSEG    XCODSEG
XLABEL:
        .
        .
        .

CODESEG         SEGMENT         CODE
        RSEG    CODESEG
        ORG     10H
        .
        .
        .
LABEL:
        .
        .
        .
        ORG     LABEL+100H
        .
        .
        .
        ORG     XLABEL          ; error
        .
        .
        .
```

The relocatable segment CODESEG contains three ORG directives with the following operands: the constant expression 10H, the simple expression LABEL+100H, and the simple expression XLABEL. The last produces an error message because XLABEL is a simple relocatable symbol from outside the current relocatable segment.

(2) Local symbol definition directive restrictions on operands

EQU, CODE, and other directives defining local symbols accept simple expressions as operands. Note, however, that they do not accept general expressions that are not simple expressions.

These directive do not accept forward references in user symbols in their operands.

(3) Restrictions on operands by other directives

Directives other than those in (1) and (2) above also sometimes impose restrictions on expressions in their operands. For further details on these restrictions, see the detailed directive descriptions in Section 5 “Detailed Directive Descriptions.”

(4) Microcontroller instruction restrictions on operands

Microcontroller instructions accept only constant expressions for the shift size operands in rotate shift instructions and the bit position in bit addressing. The other addressing types, however, accept general expressions.

3.2.4 Expression Evaluation

3.2.4.1 Operator Precedence

Operator precedence determines the order of expression evaluation. Operators with higher precedence are evaluated first; those with the same precedence, in the order in which they appear in the expression from left to right.

The following lists operators by precedence. The higher the precedence, the lower the priority number.

Priority Number	Operator
1	( ) OVL_ADDRESS OVL_SEG OVL_OFFSET
2	.
3	! ~ Unary + Unary – BYTE1 BYTE2 BYTE3 BYTE4 WORD1 WORD2 SEG OFFSET BPOS SIZE
4	* / %
5	Binary + Binary –
6	<< >>
7	< <= > >=
8	== !=
9	&
10	^
11	|
12	&&
13	||

■ Examples ■

```
LABEL  DATA   200H

       L      R0, LABEL+2*8
       SB     (LABEL+2).7
       SB     LABEL+2.7
```

This example defines an absolute symbol, LABEL, with usage type DATA. Later there are three instructions with operands containing expressions using this symbol.

The first has as its second operand the value 210H. The second has as its only operand bit 7 in the data memory byte at the address LABEL+2. What is important here is that the third does not have the same operand as the second. The dot operator (.) has higher precedence than the binary + operator, so the operand is evaluated as LABEL+(2.7).

3.2.4.2 Evaluating an Expression's Numerical Value

The assembler treats the physical segment number portion of an expression representing an address as an unsigned 8-bit integer. It treats all other numerical values as unsigned 32-bit integers—including those used in evaluating expressions. It evaluates an expression's numerical value in the order specified by the operator precedence above. There are range checks for each operand and each result along the way.

3.2.4.3 Evaluating an Expression's Attributes

This Section describes how the assembler assigns attributes to the results of expression evaluation.

If the expression consists of a single symbol, the final attribute is that of the symbol.

If the expression contains an operator, the final attribute depends on the operator type and the attributes of the operands. Some operators impose restrictions on their operands. Expressions violating these restrictions produce error or warning messages from the assembler. An error message results when the assembler does not know how to produce a result. A warning message indicates that the assembler has produced a result, but considers it suspect or even meaningless.

This Section describes the final attributes for each possible combination of operator type and operand attributes. It also notes which combinations trigger an error or warning message.

The descriptions use the following dummies to represent expressions of each type.

Symbol	Description
address_expression	Expression of type address—in other words, an expression with usage type NONE, CODE, DATA, NVDATA, TABLE, BIT, NVBIT, or TBIT. This group includes *segment_symbol*.
number_expression	Expression of type NUMBER—in other words, an expression with usage type NUMBER
code_expression	Expression with usage type CODE
data_expression	Expression with usage type DATA
nvdata_expression	Expression with usage type NVDATA
table_expression	Expression with usage type TABLE
bit_expression	Expression with usage type BIT
nvbit_expression	Expression with usage type NVBIT
tbit_expression	Expression with usage type TBIT
none_expression	Expression with usage type NONE
segment_symbol	A segment symbol alone

(1) Attribute evaluation for ( ) operator

The following are the rules for determining the attribute after evaluating expressions with this operator.

Expression Syntax	Resulting Usage Type	Error Level
(*address_expression*)	Same as *address_expression*	
(*number_expression*)	Same as *number_expression*	

■ Description ■

There is absolutely no change from the attribute of the original expression inside the parentheses.

(2) Attribute evaluation for arithmetic operators + and –

The following are the rules for determining the attribute after evaluating expressions with these operators.

Expression Syntax	Resulting Usage Type	Error Level
+ *number_expression*	Same as *number_expression*	
+ *address_expression*	Same as *address_expression*	
number_expression + *number_expression*	NUMBER	
number_expression + *address_expression*	Same as *address_expression*	
address_expression + *number_expression*	Same as *address_expression*	
address_expression + *address_expression*	NUMBER	Warning
- *number_expression*	Same as *number_expression*	
- *address_expression*	Same as *address_expression*	
number_expression – *number_expression*	NUMBER	
number_expression – *address_expression*	Same as *address_expression*	
address_expression – *number_expression*	Same as *address_expression*	
address_expression – *address_expression*	NUMBER	Error, depending on other conditions

■ Description ■

Expressions with unary + or – inherit the attribute of the original expression.

Expressions with binary + or – inherit the address attribute if either the left or right expression is of type address.

If both operands are addresses, addition produces a warning message; subtraction, an error message if the two addresses are not in the same logical segment.

(3) Attribute evaluation for arithmetic operators *, /, and %

The following are the rules for determining the attribute after evaluating expressions with these operators.

Expression Syntax	Resulting Usage Type	Error Level
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_expression*	NUMBER	
number_expression % *address_expression*	NUMBER	Warning
address_expression % *number_expression*	NUMBER	Warning
address_expression % *address_expression*	NUMBER	Warning

■ Description ■

Two operands of type address produce a warning message.

The resulting user type is always NUMBER.

(4) Attribute evaluation for logic arithmetic operators

The following are the rules for determining the attribute after evaluating expressions with these operators.

Expression Syntax	Resulting Usage Type	Error Level
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 ■

Two operands of type address produce a warning message.

The resulting user type is always NUMBER.

(5) Attribute evaluation for bitwise operators

The following are the rules for determining the attribute after evaluating expressions with these operators.

Expression Syntax	Resulting Usage Type	Error Level
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_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_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 ■

Two operands of type address produce a warning message.

The resulting user type is always NUMBER.

(6) Attribute evaluation for relational operators

The following are the rules for determining the attribute after evaluating expressions with these operators.

Expression Syntax	Resulting Usage Type	Error Level
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

Expression Syntax	Resulting Usage Type	Error Level
address_expression != *address_expression*	NUMBER	

■ **Description** ■

Mixing a number and an address produces a warning message.

Two operands of type address result in the comparison of physical segment addresses.

The resulting user type is always NUMBER.

(7) Attribute evaluation for address operator

The following are the rules for determining the attribute after evaluating expressions with this operator.

Expression Syntax	Resulting Usage Type	Error Level
number_expression :: *number_expression*	NONE	
number_expression :: *address_expression*	NONE	
address_expression :: *number_expression*	NONE	
address_expression :: *address_expression*	NONE	

■ **Description** ■

The above combinations produce neither error nor warning messages.

The resulting user type is always NONE.

(8) Attribute evaluation for the dot operator

The following are the rules for determining the attribute after evaluating expressions with this operator.

Expression Syntax	Resulting Usage Type	Error Level
number_expression.number_expression	NUMBER	
code_expression.number_expression	NUMBER	Warning
data_expression.number_expression	BIT	
nvdata_expression.number_expression	NVBIT	
table_expression.number_expression	TBIT	
bit_expression.number_expression	NUMBER	Warning
nvbit_expression.number_expression	NUMBER	Warning
tbit_expression.number_expression	NUMBER	Warning
none_expression.number_expression	NONE	
number_expression.address_expression	NUMBER	Warning
code_expression.address_expression	NUMBER	Warning
data_expression.address_expression	BIT	Warning
nvdata_expression.address_expression	NVBIT	Warning
table_expression.address_expression	TBIT	Warning
bit_expression.address_expression	NUMBER	Warning
nvbit_expression.address_expression	NUMBER	Warning
tbit_expression.address_expression	NUMBER	Warning
none_expression.address_expression	NONE	Warning

■ Description ■

Using an address expression as the right operand produces a warning message.

Using an address expression with usage type CODE, BIT, NVBIT, or TBIT as the left operand produces a warning message.

The result depends on the usage type of the left operand expression.

(9) Attribute evaluation for special operators

The following are the rules for determining the attribute after evaluating expressions with these operators.

Expression Syntax	Resulting Usage Type	Error Level
BYTE1 *number_expression*	NUMBER	
BYTE1 *address_expression*	NUMBER	Warning
BYTE2 *number_expression*	NUMBER	
BYTE2 *address_expression*	NUMBER	Warning
BYTE3 *number_expression*	NUMBER	
BYTE3 *address_expression*	NUMBER	Warning
BYTE4 *number_expression*	NUMBER	
BYTE4 *address_expression*	NUMBER	Warning
WORD1 *number_expression*	NUMBER	
WORD1 *address_expression*	NUMBER	Warning
WORD2 *number_expression*	NUMBER	
WORD2 *address_expression*	NUMBER	Warning
SEG *address_expression*	NUMBER	
OFFSET *number_expression*	NUMBER	Warning
OFFSET *address_expression*	NUMBER	
BPOS *number_expression*	NUMBER	Warning
BPOS *code_expression*	NUMBER	Warning
BPOS *data_expression*	NUMBER	Warning
BPOS *nvdata_expression*	NUMBER	Warning
BPOS *table_expression*	NUMBER	Warning
BPOS *bit_expression*	NUMBER	
BPOS *nvbit_expression*	NUMBER	
BPOS *tbit_expression*	NUMBER	
BPOS *none_expression*	NUMBER	Warning
SIZE *segment_symbol*	NUMBER	
OVL_ADDRESS *segment_symbol*	NONE	
OVL_SEG *segment_symbol*	NUMBER	
OVL_OFFSET *segment_symbol*	NUMBER	

■ Description ■

Using an address expression with the operators BYTE1 to BYTE4, WORD1, and WORD2 produces a

warning message.

The SEG operator cannot be used with an operand of usage type NUMBER because it is for extracting the physical segment address from an address.

Using the OFFSET or BPOS operator with an operand of usage type NUMBER produces a warning message because they are for operating on address expressions.

Using the BPOS operator with an operand that does not have a compatible usage type (NONE, CODE, DATA, or TABLE) produces a warning message.

The SIZE, OVL_ADDRESS, OVL_SEG , and OVL_OFFSET operators accept only segment symbols as their operands.

The resulting usage type is NONE for the OVL_ADDRESS operator and NUMBER for the rest.

4 Addressing and Instruction Types

4.1 Addressing Syntax

This Section provides Tables listing the basic instruction operands that can appear for each addressing type. For further details on these addressing types, refer to the nX-U8 Core Instruction Manual.

To make assembly language source code easier to read, the assembler goes beyond the nX-U8 Core Instruction Manual, defining its own addressing types and automatically translating them into the standard ones in that manual. These are discussed as they arise.

■ Note ■

Brackets ([ ]) have a different meaning in addressing notation—indirection, not optional items.

4.1.1 Notation

Before discussing addressing types, however, let us first describe the notation used in the descriptions.

Symbol	Description
R*n*	Byte-sized general register: R0, R1,... R14, or R15.
ER*n*	word-sized general register: ER0, ER2,... ER12, or ER14.
XR*n*	double word-sized general register: XR0, XR4, XR8, or XR12.
QR*n*	Quad word-sized general register: QR0 or QR8.
Disp16	16-bit displacement: either a general expression of usage type NUMBER between -65535 and +65535 or a general expression with a physical segment address.
Disp6	6-bit displacement: either a general expression of usage type NUMBER between -32 and +31 or a general expression with a physical segment address.
width	3-bit shift size: a constant expression of usage type NUMBER between 0 and 7.
imm8	unsigned byte-sized immediate value: a general expression of usage type NUMBER between 0 and 0FFH.
imm7	signed 7-bit value: immediate a general expression of usage type NUMBER between -64 and +63.
bit_offset	3-bit bit offset: a constant expression of usage type NUMBER between 0 and 7.
Radr	relative address: a general expression between -128 and +127 representing the distance to the branch target for a relative or conditional branch instruction.

Symbol	Description
snum7	SWI instruction vector number: a general expression of usage type NUMBER between 0 and 63.
signed8	signed byte-sized immediate value: a general expression of usage type NUMBER between -128 and +127 (RASU8 extension of *imm8*).
Sadr	SWI instruction vector address: a general expression.
unsigned8	unsigned byte-sized immediate value: a general expression of usage type NUMBER between 0 and 0FFH. (RASU8 extension of *imm8*).
Dadr	byte address in the data memory space: a general expression.
Cadr	byte address in the code memory space: a general expression.
Dbitadr	bit address in the data memory space: a general expression.
pseg_addr	physical segment address: a general expression.

The following apply to the expressions appearing as operands in instructions.

(1) All addressing types accept forward references. Note, however, that there is no optimization for addressing types whose expressions include forward references.

(2) All addressing types except *bit_offset* and *width* accept general expressions.

(3) An addressing type expecting an expression of type NUMBER also accepts one of type address. The assembler ignores the physical segment address and uses only the offset.

(4) An addressing type expecting an expression of type address also accepts one of type NUMBER. The assembler assumes #0 as the physical segment address.

(5) Addressing types expecting an expression of type address impose restrictions on the expression's usage type. An address expression of the wrong usage type produces a warning message.

4.1.2 Register Addressing

The following register addressing types access the contents of the specified register.

Addressing Notation	Function
R*n*	This addressing type accesses the contents of the specified byte-sized general register (R*n*).

Addressing Notation	Function
ER*n*	This addressing type accesses the contents of the specified word-sized general register (ER*n*). When the instruction table lists ER*n* in an operand, BP may be substituted for ER12 and FP for ER14.
XR*n*	This addressing type accesses the contents of the specified double word-sized general register (XR*n*).
QR*n*	This addressing type accesses the contents of the specified quad word-sized general register (QR*n*).
CR*n*	This addressing type accesses the contents of the specified byte-sized coprocessor register (CR*n*).
CER*n*	This addressing type accesses the contents of the specified word-sized coprocessor register (CER*n*).
CXR*n*	This addressing type accesses the contents of the specified double word-sized coprocessor register (CXR*n*).
CQR*n*	This addressing type accesses the contents of the specified quad word-sized coprocessor register (CQR*n*).
PC	This addressing type accesses the contents of the program counter.
LR	This addressing type accesses the contents of the link register.
EA	This addressing type accesses the contents of the EA register.
SP	This addressing type accesses the contents of the stack pointer.
PSW	This addressing type accesses the contents of the program status word.
ELR	This addressing type accesses the contents of the exception processing link register.
ECSR	This addressing type accesses the contents of the CSR save register.
EPSW	This addressing type accesses the contents of the PSW save register.
R*n.bit_offset*	This addressing type accesses the contents of the bit specified by *bit_offset* in general register R*n*.

4.1.3 Data Memory Addressing

This addressing type accesses the contents of an address in the data memory space.

4.1.3.1 Register Indirect Addressing

The following register indirect addressing types access the contents of the data memory address in the specified register.

Addressing Notation	Function
[EA]	This addressing type accesses the contents of the data memory space at the offset in the EA register. If there is no DSR prefix, this addressing type accesses physical segment #0 in the data memory space.
pseg_addr:[EA]	This variant uses the physical segment address specified by *#pseg_addr*. *pseg_addr* must be of usage type NUMBER. Other usage types produce an error message.
DSR:[EA]	This variant uses the physical segment address in DSR.
R*n*:[EA]	This variant uses the physical segment address in general register R*n*.
[EA+]	This addressing type accesses the contents of the data memory space at the offset in the EA register. If there is no DSR prefix, this addressing type accesses physical segment #0 in the data memory space. After the access, the contents of the EA register are incremented by the size width in bytes: one for byte-sized access, two for word-sized access, four for double word-sized access, and eight for quad word-sized instruction access.
pseg_addr:[EA+]	This variant uses the physical segment address specified by *#pseg_addr*. *pseg_addr* must be of usage type NUMBER. Other usage types produce an error message.
DSR:[EA+]	This variant uses the physical segment address in DSR.
R*n*:[EA+]	This variant uses the physical segment address in general register R*n*.
[ER*n*]	This addressing type accesses the contents of the data memory space at the offset in the word-sized general register ER*n*. If there is no DSR prefix, this addressing type accesses physical segment #0 in the data memory space. BP may be substituted for ER12 and FP for ER14.
pseg_addr:[ER*n*]	This variant uses the physical segment address specified by *#pseg_addr*. *pseg_addr* must be of usage type NUMBER. Other usage types produce an error message.. The LEA instruction does not accept this addressing type.
DSR:[ER*n*]	This variant uses the physical segment address in DSR. The LEA instruction does not accept this addressing type.

Addressing Notation	Function
R*n*:[ER*m*]	This variant uses the physical segment address in general register R*n*. The LEA instruction does not accept this addressing type.
Disp16[ER*n*]	This addressing type accesses the contents of the data memory space at the byte address formed by adding the displacement *Disp16* to the contents of the word-sized general register ER*n*. The default addressing specifier is usually the same as the data model, but is NEAR for the FAR model if both the expression *Disp16* does not contain a forward reference and the physical segment attribute is #0. *Disp16* must be of usage type DATA, NVDATA, TABLE, NONE, or NUMBER. Other usage types produce a warning message.
NEAR *Disp16*[ER*n*]	This variant limits access to physical segment #0. The LEA instruction does not accept this addressing type.
FAR *Disp16*[ER*n*]	This variant accesses physical segment #(SEG *Disp16*), the physical segment in the data memory space to which *Disp16* belongs. The LEA instruction does not accept this addressing type.
DSR:*Disp16*[ER*n*]	This variant uses the physical segment address in DSR and ignores the physical segment to which *Disp16* belongs. *Disp16* must be of usage type NUMBER. Other usage types produce a warning message.. The LEA instruction does not accept this addressing type.
R*n*:*Disp16*[ER*m*]	This variant uses the physical segment address in general register R*n* and ignores the physical segment to which *Disp16* belongs. *Disp16* must be of usage type NUMBER. Other usage types produce a warning message.. The LEA instruction does not accept this addressing type.
Disp16[BP]	This addressing type, which does not appear in the nX-U8 Core Instruction Manual, accesses the contents of the data memory space at the byte address formed by adding the displacement *Disp16* to the contents of the word-sized base pointer BP. The default addressing specifier is usually the same as the data model, but is NEAR for the FAR model if both the expression *Disp16* does not contain a forward reference and the physical segment attribute is #0. *Disp16* must be of usage type DATA, NVDATA, TABLE, NONE, or NUMBER. Other usage types produce a warning message.

Addressing Notation	Function
NEAR *Disp16*[BP]	This variant limits access to physical segment #0. The LEA instruction does not accept this addressing type.
FAR *Disp16*[BP]	This variant accesses physical segment #(SEG *Disp16*), the physical segment in the data memory space to which *Disp16* belongs. The LEA instruction does not accept this addressing type.
DSR:*Disp16*[BP]	This variant uses the physical segment address in DSR and ignores the physical segment to which *Disp16* belongs. *Disp16* must be of usage type NUMBER. Other usage types produce a warning message.. The LEA instruction does not accept this addressing type.
R*n*:*Disp16*[BP]	This variant uses the physical segment address in general register R*n* and ignores the physical segment to which *Disp16* belongs. *Disp16* must be of usage type NUMBER. Other usage types produce a warning message.. The LEA instruction does not accept this addressing type.
Disp16[FP]	This addressing type, which does not appear in the U8 Instruction Manual, accesses the contents of the data memory space at the byte address formed by adding the displacement *Disp16* to the contents of the word-sized frame pointer FP. The default addressing specifier is usually the same as the data model, but is NEAR for the FAR model if both the expression *Disp16* does not contain a forward reference and the physical segment attribute is #0. *Disp16* must be of usage type DATA, NVDATA, TABLE, NONE, or NUMBER. Other usage types produce a warning message.
NEAR *Disp16*[FP]	This variant limits access to physical segment #0. The LEA instruction does not accept this addressing type.
FAR *Disp16*[FP]	This variant accesses physical segment #(SEG *Disp16*), the physical segment in the data memory space to which *Disp16* belongs. The LEA instruction does not accept this addressing type.
DSR:*Disp16*[FP]	This variant uses the physical segment address in DSR and ignores the physical segment to which *Disp16* belongs. *Disp16* must be of usage type NUMBER. Other usage types produce a warning message.. The LEA instruction does not accept this addressing type.

Addressing Notation	Function
R*n*:*Disp16*[FP]	This variant uses the physical segment address in general register R*n* and ignores the physical segment to which *Disp16* belongs. *Disp16* must be of usage type NUMBER. Other usage types produce a warning message.. The LEA instruction does not accept this addressing type.

4.1.3.2 Direct Addressing

The following direct addressing types access the contents of the data memory address from the specified general expression.

Addressing Notation	Function
Dadr	This addressing type accesses the contents of the data memory space at the byte address in the instruction. The default addressing specifier is usually the same as the data model, but is NEAR for the FAR model if both the expression *Dadr* does not contain a forward reference and the physical segment attribute is #0. *Dadr* must be of usage type DATA, NVDATA, TABLE, NONE, or NUMBER. Other usage types produce a warning message.
NEAR *Dadr*	This variant limits access to physical segment #0. The LEA instruction does not accept this addressing type.
FAR *Dadr*	This variant accesses physical segment #(SEG Dadr), the physical segment in the data memory space to which *Dadr* belongs. The LEA instruction does not accept this addressing type.
DSR:*Dadr*	This variant uses the physical segment address in DSR and ignores the physical segment to which Disp16 belongs. *Dadr* must be of usage type NUMBER. Other usage types produce a warning message.. The LEA instruction does not accept this addressing type.
R*n*:*Dadr*	This variant uses the physical segment address in general register R*n* and ignores the physical segment to which *Dadr* belongs. *Dadr* must be of usage type NUMBER. Other usage types produce a warning message.. The LEA instruction does not accept this addressing type.

Addressing Notation	Function
Dbitadr	This addressing type accesses the contents of the data memory space at the bit address in the instruction. The default addressing specifier is usually the same as the data model, but is NEAR for the FAR model if both the expression *Dbitadr* does not contain a forward reference and the physical segment attribute is #0. *Dbitadr* must be of usage type BIT, NVBIT, TBIT, NONE, or NUMBER. Other usage types produce a warning message.
NEAR *Dbitadr*	This variant limits access to physical segment #0. The LEA instruction does not accept this addressing type.
FAR *Dbitadr*	This variant accesses physical segment #(SEG *Dbitadr*), the physical segment in the data memory space to which *Dbitadr* belongs. The LEA instruction does not accept this addressing type.
DSR:*Dbitadr*	This variant uses the physical segment address in DSR and ignores the physical segment to which Disp16 belongs. *Dbitadr* must be of usage type NUMBER. Other usage types produce a warning message.. The LEA instruction does not accept this addressing type.
R*n*:*Dbitadr*	This variant uses the physical segment address in general register R*n* and ignores the physical segment to which *Dbitadr* belongs. *Dbitadr* must be of usage type NUMBER. Other usage types produce a warning message.. The LEA instruction does not accept this addressing type.

4.1.4 Immediate Value Addressing

The following immediate value addressing types use an immediate value contained in the instruction. All except Sadr, an extension, must be of usage type NUMBER.

Addressing Notation	Function
#*imm8*	The specified value is treated as an 8-bit immediate value. *imm8* must be of usage type NUMBER. Other usage types produce a warning message.

Addressing Notation	Function
#signed8	The specified value is treated as a signed 8-bit immediate value. The instruction ADD SP, *#imm8* treats *imm8* as *signed8*. The valid range for *signed8* is between -128 and +127. *signed8* must be of usage type NUMBER. Other usage types produce a warning message.
#unsigned8	The specified value is treated as an unsigned 8-bit immediate value. The instruction MOV PSW, *#imm8* treats *imm8* as *unsigned8*. The valid range for *unsigned8* is between 0 and 0FFH. *unsigned8* must be of usage type NUMBER. Other usage types produce a warning message.
#width	The specified value is treated as a shift size. The valid range for *width* is between 0 and 7. *width* must be of usage type NUMBER. Other usage types produce a warning message.
#snum7	The specified value is treated as a SWI instruction vector number. The valid range for *snum7* is between 0 and 63. *snum7* must be of usage type NUMBER. Other usage types produce a warning message.
#imm7	The specified value is treated as a signed 7-bit immediate value. The valid range for *imm7* is between -64 and +63. *imm7* must be of usage type NUMBER. Other usage types produce a warning message.
Sadr	The specified value is treated as a SWI instruction vector address. The assembler automatically replaces this with *#snum7* using the following expression, where *swi_vector_start* is the starting address for the SWI vector region. *snum7* = (*Sadr – swi_vector_start*)/2 *Sadr* must be of usage type CODE, NONE, or NUMBER. Other usage types produce a warning message.

4.1.5 Code Memory Addressing

The following code memory addressing types access the contents of the code memory address. access the contents of code memory addresses.

Addressing Notation	Function
Cadr	This addressing type specifies the branch target address for the B and BL instructions. Note that it contains a physical segment address, so the instruction can produce a branch to a different physical segment.
Radr	This addressing type specifies a relative branch target address for the conditional branch instructions and optimized branch directives. The target must be in within the same physical segment.
ER*n*	This addressing type specifies the contents of a word-sized general register ER*n* as the branch target offset for the B and BL instructions. The target must be in within the same physical segment.

4.2 Instruction Types

This Section lists the operand and addressing combinations possible with nX-U8 instructions. For further details on individual instructions, refer to the nX-U8 Core Instruction Manual. For further details on addressing syntax, see Section 4.1 “Addressing Syntax.”

4.2.1 Arithmetic Instructions

Mnemonic	Operand 1	Operand 2	Notes
MOV ADD AND OR XOR CMPC ADDC CMP	R*n*	#*imm8* R*m*	*imm8* must be of usage type NUMBER.
SUB SUBC	R*n*	R*m*	
MOV ADD	ER*n*	ER*m* #*imm7*	*imm7* must be of usage type NUMBER.
CMP	ER*n*	ER*m*	

4.2.2 Shift Instructions

Mnemonic	Operand 1	Operand 2	Notes
SLL SRL SRA SLLC SRLC	R*n*	R*m* #*width*	*width* must be of usage type NUMBER.

4.2.3 Load and Store Instructions

Mnemonic	Operand 1	Operand 2	Notes
L ST	R*n* ER*n* XR*n* QR*n*	[EA] *pseg_addr*:[EA] DSR:[EA] R*n*:[EA]	*pseg_addr* must be of usage type NUMBER.
		[EA+] *pseg_addr*:[EA+] DSR:[EA+] R*n*:[EA+]	
	R*n* ER*n*	[ER*m*] *pseg_addr*:[ER*m*] DSR:[ER*m*] R*n*:[ER*m*]	*pseg_addr* must be of usage type NUMBER.
		Disp16[ER*m*] NEAR *Disp16*[ER*m*] FAR *Disp16*[ER*m*]	*Disp16* must be of usage type DATA, NVDATA, TABLE, NONE, or NUMBER.
		Disp16[BP] NEAR *Disp16*[BP] FAR *Disp16*[BP]	
		Disp16[FP] NEAR *Disp16*[FP] FAR *Disp16*[FP]	
		DSR:*Disp16*[ER*m*] R*n*:*Disp16*[ER*m*]	*Disp16* must be of usage type NUMBER.
		DSR:*Disp16*[BP] R*n*:*Disp16*[BP]	
		DSR:*Disp16*[FP] R*n*:*Disp16*[FP]	
		Dadr NEAR *Dadr* FAR *Dadr*	*Dadr* must be of usage type DATA, NVDATA, TABLE, NONE, or NUMBER.

Mnemonic	Operand 1	Operand 2	Notes
		DSR:*Dadr* R*m*:*Dadr*	*Dadr* must be of usage type NUMBER.

4.2.4 Control Register Access Instructions

Mnemonic	Operand 1	Operand 2	Notes
MOV	R*n*	PSW EPSW ECSR	
	PSW EPSW ECSR	R*m*	
	ER*n*	ELR SP	BP may be substituted for ER12 and FP for ER14.
	ELR SP	ER*m*	BP may be substituted for ER12 and FP for ER14.
	PSW	#*unsigned8*	*unsigned8* must be of usage type NUMBER.
ADD	SP	#*signed8*	*signed8* must be of usage type NUMBER.

4.2.5 PUSH/POP Instructions

Mnemonic	Operand 1	Operand 2	Notes
PUSH	R*n* ER*n* XR*n* QR*n* *register_list*		*register_list* contains one or more of the following registers: EPSW, ELR, LR, and EA.
POP	R*n* ER*n* XR*n* QR*n* *register_list*		*register_list* contains one or more of the following registers: PSW, LR, PC, and EA.

4.2.6 Coprocessor Transfer Instructions

Mnemonic	Operand 1	Operand 2	Notes
MOV	CR*n* CER*n* CXR*n* CQR*n*	R*m* [EA] *pseg_addr*:[EA] DSR:[EA] R*m*:[EA] [EA+] *pseg_addr*:[EA+] DSR:[EA+] R*m*:[EA+]	*pseg_addr* must be of usage type NUMBER.
	R*n* [EA] *pseg_addr*:[EA] DSR:[EA] R*n*:[EA] [EA+] *pseg_addr*:[EA+] DSR:[EA+] R*n*:[EA+]	CR*m* CER*m* CXR*m* CQR*m*	*pseg_addr* must be of usage type NUMBER.

4.2.7 EA Register Transfer Instructions

Mnemonic	Operand 1	Operand 2	Notes
LEA	[ER*n*] *Disp16*[ER*n*] *Dadr*		*Disp16* and *Dadr* must be of usage type NUMBER.

4.2.8 ALU Instructions

Mnemonic	Operand 1	Operand 2	Notes
DAA DAS NEG	R*n*		

4.2.9 Bit Access Instructions

Mnemonic	Operand 1	Operand 2	Notes
SB TB RB	R*n.bit_offset* *Dbitadr*		*Dbitadr* must be of usage type BIT, NVBIT, TBIT, NONE, or NUMBER.

4.2.10 PSW Access Instructions

Mnemonic	Operand 1	Operand 2	Notes
EI DI SC RC CPLC			

4.2.11 Conditional Branch Instructions

Mnemonic	Operand 1	Operand 2	Notes
BEQ BNE BLT BLE BGT BGE BLTS BLES BGTS BGES BZ BNZ BCY BNC BOV BNV BPS BNS BAL	*Radr*		*Radr* must be of usage type CODE, NONE, or NUMBER.

4.2.12 Sign Extension Instructions

Mnemonic	Operand 1	Operand 2	Notes
EXTBW	ER*n*		

4.2.13 Software Interrupt Instruction

Mnemonic	Operand 1	Operand 2	Notes
SWI	#*snum7*		*snum7* must be of usage type NUMBER.
	Sadr		*Sadr* must be of usage type CODE, NONE, or NUMBER.
BRK			

4.2.14 Branch Instructions

Mnemonic	Operand 1	Operand 2	Notes
B BL	*Cadr* ER*n*		*Cadr* must be of usage type CODE, NONE, or NUMBER.

4.2.15 Multiplication and Division Instructions

Mnemonic	Operand 1	Operand 2	Notes
MUL DIV	ER*n*	R*m*	

4.2.16 Miscellaneous Instructions

Mnemonic	Operand 1	Operand 2	Notes
INC DEC	[EA]		
RTI RT NOP			

5 Detailed Directive Descriptions

5.1 Assembler Initialization Directives

These directives configure the assembler for the current project. They must, therefore, appear at the beginning of the program.

5.1.1 TYPE Directive

■ Syntax ■

TYPE (*dcl_name*)

■ Description ■

This directive specifies the base name of the DCL file for the target microcontroller. The assembler then reads in device-specific information from the DCL file with the base name dcl_name and the file extension .DCL.

This base name is the microcontroller model number less the L in M610001 for the target microcontroller ML610001, for example.

The assembler searches directories for the DCL file in the following order.

1. current directory
2. directory containing its executable (RASU8.EXE)
3. directory specified by the environment variable DCL

The assembler starts by reading in the entire DCL file. If there are errors, it aborts with an error message at the end of the file. If the read completes successfully, the assembler continues processing the specified source code file.

■ Aside ■

The same TYPE directive must appear in each source code file for the program. If the assembler cannot find a TYPE directive or the specified DCL file, it aborts without assembling the source code file.

Place the TYPE directive (or the include file containing it) at the beginning of the program.

This directive can appear only once per file.

■ **Example** ■

```
        TYPE(M610001)

        EXTRN DATA: _$$SP

        CSEG AT 0H
        DW  _$$SP
        DW  START
        CSEG AT 1000H
    START:
        .
        .
        .
```

This example is for the target microcontroller ML610001. The assembler reads in the DCL file M610001.DCL.

5.1.2 MODEL Directive

■ **Syntax** ■

MODEL *memory_model* [, *data_model*]

or

MODEL *data_model* [, *memory_model*]

■ **Description** ■

This directive specifies the memory (SMALL or LARGE) and data (NEAR or FAR) models to the assembler. The defaults are SMALL and NEAR, respectively.

memory_model	**Description**
SMALL model	All program code goes in physical segment #0.
LARGE model	There may be program code for physical segment #1 or higher.

data_model	**Description**
NEAR	All data without an overt addressing type specifier goes in physical segment #0.
FAR	Data without an overt addressing type specifier may go in physical segment #1 or higher.

■ **Example** ■

```
TYPE (M610001)
MODEL LARGE, NEAR
REL_CODE_SEG SEGMENT CODE
REL_DATA_SEG SEGMENT DATA
```

This example specifies the LARGE memory model and the NEAR data model.

■ **Aside** ■

Note that the directive only specifies the memory and data models to the assembler. Hardware setup and control with the memory model control registers is also required.

5.1.3 ROMWINDOW Directive

■ **Syntax** ■

ROMWINDOW *base_address*, *end_address*

■ **Description** ■

This directive specifies the first and last addresses in the ROM window region using constant expressions.

The combinations available depend on the microcontroller, so refer to the ROMWINDOW statement inside the corresponding DCL file.

■ **Aside** ■

Note that the ROMWINDOW directive only specifies the address range to the assembler. Hardware setup and control is also required in the program.

This directive can appear only once per file. It cannot appear after the NOROMWIN directive.

■ **Example** ■

```
TYPE (610001)
ROMWINDOW 0, 3FFFH
  .
  .
  .
```

This example specifies a ROM window region with addresses 0 to 3FFFH inclusive.

5.1.4 NOROMWIN Directive

■ Syntax ■

NOROMWIN

■ Description ■

This directive specifies that the program does not use a ROM window and thus tells the assembler that there is no TABLE address space in physical segment #0. Allocating a region of type TABLE in physical segment #0 then produces an error message.

Linking modules not using a ROM window (NOROMWIN directive) with ones that do (ROMWIN or neither directive) is not allowed, so the linker aborts with an error message.

Linking modules not using a ROM window with ones that do causes the linker to abort with an error message.

■ Aside ■

The NOROMWIN directive can appear only once per file. It cannot appear after the ROMWINDOW directive.

5.2 File End Directive

This directive specifies the end of assembler input for current file.

5.2.1 END Directive

■ Syntax ■

END

■ Description ■

This directive specifies to the assembler that the current file ends at that point. No statements beyond that line are assembled. If there is no END directive, the assembler processes everything through to the end of the file. The assembler interprets an END directive in an include file as the end of that file, so returns to processing the specifying file.

5.3 Symbol Definition Directives

These directives define symbols, assigning numerical or address values to them.

5.3.1 EQU Directive

■ Syntax ■

symbol EQU *simple_expression*

■ Description ■

This directive defines the specified local symbol from a simple expression that does not contain a forward reference.

The *symbol* is absolute for a constant expression and simple relocatable for a simple relocatable one. The value and usage type are those of the expression.

■ Aside ■

This directive cannot redefine a symbol that has already been defined.

The assembler issues a warning message if the address usage type (one other than NUMBER and NONE) is not compatible with the memory type for the corresponding address.

■ Example ■

```
SEGSYM   SEGMENT DATA 2
         RSEG    SEGSYM
BUF1:    DS      4

BASE     EQU     10H
BUFSIZE  EQU     4H
VALUE    EQU     BASE+BUFSIZE
BUFX     EQU     BUF1+BUFSIZE
```

This example defines four local symbols: BASE, BUFSIZE, and VALUE are absolute symbols with usage type NUMBER; BUFX, a simple relocatable symbol with usage type DATA.

5.3.2 SET Directive

■ Syntax ■

symbol SET *simple_expression*

■ Description ■

This directive defines the specified local symbol from a simple expression that does not contain a forward reference.

This directive is functionally the same as the EQU directive except that symbols defined with SET directives can be redefined with SET directives any number of times.

The *symbol* is absolute for a constant expression and simple relocatable for a simple relocatable one. The value and usage type are those of the expression.

■ Aside ■

This directive cannot redefine a symbol that has already been defined—unless that symbol is defined with a SET directive.

The assembler issues a warning message if the address usage type (one other than NUMBER and NONE) is not compatible with the memory type for the corresponding address.

■ Example ■

```
SETSYM  SET 10H
        MOV R0,     #SETSYM
        .
        .
        .
SETSYM  SET 20H
        MOV R0,     #SETSYM
        .
        .
        .
```

This example defines SETSYM, an absolute symbol with usage type NUMBER, twice. In the two MOV instructions immediately after the SET directives, therefore, SETSYM has the value 10H the first time and 20H the second time.

5.3.3 CODE Directive

■ Syntax ■

symbol CODE *simple_expression*

■ Description ■

This directive defines a local symbol representing a byte address in the CODE address space using a simple expression that does not contain a forward reference.

The *symbol* is absolute for a constant expression and simple relocatable for a simple relocatable one. The value and usage type (CODE) are those of the expression.

The assembler issues a warning message if the address value from the simple expression is not within the CODE address space range.

■ Aside ■

This directive cannot redefine a symbol that has already been defined.

The simple expression must be of usage type CODE, NONE, or NUMBER. Other usage types produce an error message. If the usage type is NUMBER, has physical segment #0.

■ Example ■

```
CODE_SYM1 CODE 1000H
CODE_SYM2 CODE 2:2000H
          CSEG #3 AT  3000H
LABEL:    DW  1000H
CODE_SYM3 CODE LABEL+100H
```

This example defines three absolute symbols with usage type CODE: CODE_SYM1 representing offset 1000H in physical segment #0, CODE_SYM2 representing offset 2000H in physical segment #2, and CODE_SYM3 representing offset 3100H in physical segment #3.

5.3.4 TABLE Directive

■ Syntax ■

symbol TABLE *simple_expression*

■ Description ■

This directive defines a local symbol representing a byte address in the TABLE address space using a simple expression that does not contain a forward reference.

The *symbol* is absolute for a constant expression and simple relocatable for a simple relocatable one. The value and usage type (TABLE) are those of the expression.

The assembler issues a warning message if the address value from the simple expression is not within the TABLE address space range.

The assembler also issues a warning message if the file contains the NOROMWIN directive and *simple_expression* represents an address in physical segment #0.

■ Aside ■

This directive cannot redefine a symbol that has already been defined.

The simple expression must be of usage type TABLE, NONE, or NUMBER. Other usage types produce an error message. If the usage type is NUMBER, has physical segment #0.

The assembler issues a warning message if it is unable to check a *simple_expression* value in physical segment #0 because the address range for the ROM window region is unspecified.

■ Example ■

```
TABLE_SYM1 TABLE 1000H
TABLE_SYM2 TABLE 2:2000H
          TSEG #3 AT  3000H
LABEL:    DW  1234H
TABLE_SYM3 TABLE LABEL+100H
```

This example defines three absolute symbols with usage type TABLE: TABLE_SYM1 representing offset 1000H in physical segment #0, TABLE_SYM2 representing offset 2000H in physical segment #2, and TABLE_SYM3 representing offset 3100H in physical segment #3.

5.3.5 TBIT Directive

■ Syntax ■

symbol TBIT *simple_expression*

■ Description ■

This directive defines a local symbol representing a bit address in the TBIT address space using a simple expression that does not contain a forward reference.

The *symbol* is absolute for a constant expression and simple relocatable for a simple relocatable one. The value and usage type (TBIT) are those of the expression.

The assembler issues a warning message if the address value from the simple expression is not within the TBIT address space range.

The assembler also issues a warning message if the file contains the NOROMWIN directive and *simple_expression* represents an address in physical segment #0.

■ Aside ■

This directive cannot redefine a symbol that has already been defined.

The simple expression must be of usage type TBIT, NONE, or NUMBER. Other usage types produce an error message. If the usage type is NUMBER, has physical segment #0.

The assembler issues a warning message if it is unable to check a *simple_expression* value in physical segment #0 because the address range for the ROM window region is unspecified.

■ Example ■

```
TBIT_SYM1 TBIT 1000H.1
TBIT_SYM2 TBIT 2:2000H.4
          TSEG #3 AT  3000H
LABEL:    DB   0CAH
TBIT_SYM3 TBIT LABEL.3
```

This example uses TBIT directive to define three absolute symbols with usage type TBIT: TBIT_SYM1 representing bit 1 at offset 1000H in physical segment #0, TBIT_SYM2 representing bit 4 at offset 2000H in physical segment #2, and TBIT_SYM3 representing bit 3 at offset 3000H in physical segment #3.

5.3.6 DATA Directive

■ Syntax ■

symbol DATA *simple_expression*

■ Description ■

This directive defines a local symbol representing a byte address in the DATA address space using a simple expression that does not contain a forward reference.

The *symbol* is absolute for a constant expression and simple relocatable for a simple relocatable one. The value and usage type (DATA) are those of the expression.

The assembler issues a warning message if the address value from the simple expression is not within the DATA address space range.

■ Aside ■

This directive cannot redefine a symbol that has already been defined.

The simple expression must be of usage type DATA, NONE, or NUMBER. Other usage types produce an error message. If the usage type is NUMBER, has physical segment #0.

The assembler issues a warning message if it is unable to check a *simple_expression* value in physical segment #0 outside the internal RAM and SFR regions because the address range for the ROM window region is unspecified.

■ Example ■

```
DATA_SYM1 DATA 0A000H
DATA_SYM2 DATA 4:2000H
          DSEG #5 AT  3000H
LABEL:    DS   2
DATA_SYM3 DATA LABEL+100H
```

This example defines three absolute symbols with usage type DATA: with the directive, DATA_SYM1 representing offset A000H in physical segment #0, DATA_SYM2 representing offset 2000H in physical segment #4, and DATA_SYM3 representing offset 3100H in physical segment #5.

5.3.7 BIT Directive

■ Syntax ■

symbol BIT *simple_expression*

■ Description ■

This directive defines a local symbol representing a bit address in the BIT address space using a simple expression that does not contain a forward reference.

The *symbol* is absolute for a constant expression and simple relocatable for a simple relocatable one. The value and usage type (BIT) are those of the expression.

The assembler issues a warning message if the address value from the simple expression is not within the BIT address space range.

■ Aside ■

This directive cannot redefine a symbol that has already been defined.

The simple expression must be of usage type BIT, NONE, or NUMBER. Other usage types produce an error message. If the usage type is NUMBER, has physical segment #0.

The assembler issues a warning message if it is unable to check a *simple_expression* value in physical segment #0 outside the internal RAM and SFR regions because the address range for the ROM window region is unspecified.

■ Example ■

```
BIT_SYM1 BIT  0A000H.1
BIT_SYM2 BIT  4:2000H.2
         DSEG #5 AT  3000H
LABEL:   DS   2
BIT_SYM3 BIT  LABEL.3
```

This example defines three absolute symbols with usage type BIT: BIT_SYM1 representing bit 1 at offset A000H in physical segment #0, BIT_SYM2 representing bit 2 at offset 2000H in physical segment #4, and BIT_SYM3 representing bit 3 at offset 3000H in physical segment #5.

5.3.8 NVDATA Directive

■ Syntax ■

symbol NVDATA *simple_expression*

■ Description ■

This directive defines a local symbol representing a byte address in the NVDATA address space using a simple expression that does not contain a forward reference.

The *symbol* is absolute for a constant expression and simple relocatable for a simple relocatable one. The value and usage type (NVDATA) are those of the expression.

The assembler issues a warning message if the address value from the simple expression is not within the NVDATA address space range.

■ Aside ■

This directive cannot redefine a symbol that has already been defined.

The simple expression must be of usage type NVDATA, NONE, or NUMBER. Other usage types produce an error message. If the usage type is NUMBER, has physical segment #0.

The assembler issues a warning message if it is unable to check a *simple_expression* value in physical segment #0 because the address range for the ROM window region is unspecified.

■ Example ■

```
DATA_SYM1 NVDATA 8000H
DATA_SYM2 NVDATA 4:6000H
          NVSEG #5 AT 4000H
LABEL:    DS   2
NVDATA_SYM3 NVDATA LABEL+100H
```

This example defines three absolute symbols with usage type NVDATA: NVDATA_SYM1 representing offset 8000H in physical segment #0, NVDATA_SYM2 representing offset 6000H in physical segment #4, and NVDATA_SYM3 representing offset 4100H in physical segment #5.

5.3.9 NVBIT Directive

■ Syntax ■

symbol NVBIT *simple_expression*

■ Description ■

This directive defines a local symbol representing a bit address in the NVBIT address space using a simple expression that does not contain a forward reference.

The *symbol* is absolute for a constant expression and simple relocatable for a simple relocatable one. The value and usage type (NVBIT) are those of the expression.

The assembler issues a warning message if the address value from the simple expression is not within the NVBIT address space range.

The assembler also issues a warning message if the file contains the NOROMWIN directive and *simple_expression* represents an address in physical segment #0.

■ Aside ■

This directive cannot redefine a symbol that has already been defined.

The simple expression must be of usage type NVBIT, NONE, or NUMBER. Other usage types produce an error message. If the usage type is NUMBER, has physical segment #0.

The assembler issues a warning message if it is unable to check a *simple_expression* value in physical segment #0 because the address range for the ROM window region is unspecified.

■ Example ■

```
NVBIT_SYM1 NVBIT  8000H.1
NVBIT_SYM2 NVBIT  4:6000H.2
          NVSEG #5 AT  4000H
LABEL:    DS   2
NVBIT_SYM3 NVBIT  LABEL.3
```

This example defines three absolute symbols with usage type NVBIT: BIT_SYM1 representing bit 1 at offset 8000H in physical segment #0, BIT_SYM2 representing bit 2 at offset 6000H in physical segment #4, and BIT_SYM3 representing bit 3 at offset 4000H in physical segment #5.

5.4 Absolute Segment Definition Directives

An nX-U8 assembly language program normally consists of more than a single logical segment and must specify to the assembler when it switches logical segment types.

This Section describes the directives for defining absolute logical segments. The RSEG directive for starting relocatable segments appears below in Section 5.5 “Relocatable Segment Definition Directives.”

5.4.1 CSEG Directive

■ Syntax ■

CSEG [#*pseg_addr*][AT *start_address*]

CSEG [#*pseg_addr*] AT *overlay_address* OVL *allocation_address*

■ Description ■

This directive declares the start of an absolute CODE segment at the specified starting address. For further details on these parameters, see Section 5.4.7 “Parameters for Absolute Segment Definition Directives” below.

This directive accepts as its parameters #*pseg_addr*, AT *start_address*.

The variant with OVL defines an absolute overlay segment. For further details on overlay segments and overlays, see Chapter 10 “Using Overlays.”

5.4.2 DSEG Directive

■ Syntax ■

DSEG [#*pseg_addr*][AT *start_address*]

■ Description ■

This directive declares the start of an absolute DATA segment at the specified starting address. For further details on these parameters, see Section 5.4.7 “Parameters for Absolute Segment Definition Directives” below.

This directive accepts as its parameters #*pseg_addr*, AT *start_address*.

5.4.3 BSEG Directive

■ Syntax ■

BSEG [#*pseg_addr*][AT *start_address*]

■ Description ■

This directive declares the start of an absolute BIT segment at the specified starting address. For further details on these parameters, see Section 5.4.7 “Parameters for Absolute Segment Definition Directives” below.

This directive accepts as its parameters #*pseg_addr*, AT *start_address*.

5.4.4 NVSEG Directive

■ Syntax ■

NVSEG [#*pseg_addr*][AT *start_address*]

■ Description ■

This directive declares the start of an absolute NVDATA segment at the specified starting address. For further details on these parameters, see Section 5.4.7 “Parameters for Absolute Segment Definition Directives” below.

This directive accepts as its parameters #*pseg_addr*, AT *start_address*.

5.4.5 NVBSEG Directive

■ Syntax ■

NVBSEG [#*pseg_addr*][AT *start_address*]

■ Description ■

This directive declares the start of an absolute NVBIT segment at the specified starting address. For further details on these parameters, see Section 5.4.7 “Parameters for Absolute Segment Definition Directives” below.

This directive accepts as its parameters #*pseg_addr*, AT *start_address*.

5.4.6 TSEG Directive

■ Syntax ■

TSEG [*#pseg_addr*][AT *start_address*]

■ Description ■

This directive declares the start of an absolute TABLE segment at the specified starting address. For further details on these parameters, see Section 5.4.7 “Parameters for Absolute Segment Definition Directives” below.

This directive accepts as its parameters *#pseg_addr*, AT *start_address*.

5.4.7 Parameters for Absolute Segment Definition Directives

The absolute segment definition directives (CSEG, DSEG, BSEG, NVSEG, NVBSEG, and TSEG) all share a common parameter syntax: [*#pseg_addr*] [AT *start_address*].

The field *pseg_addr* specifies the physical segment address for the logical segment and must be a constant expression that does not contain a forward reference and has a value within the range available for the segment type.

The field *start_address* after the AT keyword specifies the starting address for the logical segment and must be a constant expression that does not contain a forward reference. This optional portion of the parameter syntax updates the location counter contents to the specified address value in a fashion that depends on the usage type for *start_address*: an expression of type NUMBER updates just the offset; one of type address, on the other hand, updates the physical segment number as well. The latter type cannot appear simultaneously with a *#pseg_addr* specification. In other words, if a *#pseg_addr* specification is present, any *start_address* present is limited to an expression of type NUMBER.

Suppose, for example, we wish to define an absolute CODE segment starting at offset 1000H in physical segment #1.

```
CSEG #1 AT 1000H
CSEG    AT 1:1000H
```

These first two formulations both produce the desired effects.

```
CSEG #1 AT 1:1000H  ;ERROR
```

The assembler issues an error message, however, for this third formulation.

Omitting either the *#pseg_addr* portion or the AT *start_address* portion (or both) causes the logical segment to inherit the immediately preceding settings for logical segments of the same segment type.

(1) Omitting #*pseg_addr*

Omitting this portion causes the logical segment to inherit the immediately preceding physical segment address setting for logical segments of the same segment type—but only if start_address is an expression of type NUMBER.

```
CSEG    #2  AT  1000H
.
.
.
CSEG    AT  3000H ; Inherits physical segment #2
```

The first CSEG directive specifies physical segment #2. The next CSEG directive specifies no physical segment—only the starting offset of 3000H—so the assembler continues using physical segment #2.

(2) Omitting AT *start_address*

Omitting this portion causes the logical segment to inherit the immediately preceding offset setting for logical segments of the same segment type.

```
DSEG    #4  AT  1000H
DS      10H     ;1010H
CSEG    #1  AT  8000H
.
.
.
DSEG    #4 ; Starting offset = 1010H
```

The first DSEG directive in this example specifies a starting offset of 1000H in physical segment #4 and allocates 10H bytes with a DS directive to advance the location counter to 1010H. The next DSEG directive specifies the same physical segment address, but no starting offset. The assembler therefore uses as the starting offset 1010H, the location counter contents at the end of the immediately preceding logical segment in physical segment #4.

(3) Omitting both

Omitting both the #*pseg_addr* portion and the AT *start_address* portion causes the logical segment to inherit the immediately preceding settings for logical segments of the same segment type.

```
DSEG    #4  AT  1000H
DS      10H     ;1010H
DSEG    #1  AT  8000H
DS      20H     ;8020H
.
.
.
DSEG    ; Starting address 1:8020H
```

The last DSEG directive specifies neither a physical segment address nor a starting offset. The assembler therefore uses as the starting address offset 8020H in physical segment #1.

5.5 Relocatable Segment Definition Directives

Relocatable segments are logical segments for which the assembler cannot determine an absolute address. The linker determines the final address.

Defining a relocatable segment is a two-step procedure.

(1) Define a segment symbol with the SEGMENT directive.

(2) Define a relocatable segment with an RSEG directive using the segment symbol as its operand.

5.5.1 SEGMENT Directive

■ Syntax ■

segment_symbol SEGMENT *segment_type* [*boundary_attr*][*seg_attr*][*relocation_attr*]

■ Description ■

This directive defines a segment symbol. A single source file can define up to a maximum of 65535 segment symbols.

The name (*segment_symbol*) serves to distinguish relocatable segments in RSEG directives. It can also appear as an instruction operand, where it represents the base address of the relocatable segment. In this case, the assembler assigns the value 0 to the operand.

The SEGMENT directive has four parameters. The first, *segment_type*, must always appear. The remaining three, *boundary_attr*, *seg_attr*, and *relocation_attr* can be omitted if not necessary.

The following describes each parameter in turn.

segment_type

This field specifies the address space for the relocatable segment: CODE, DATA, BIT, NVDATA, NVBIT, or TABLE.

segment_type	**Description**
CODE	Assign to the CODE address space
DATA	Assign to the DATA address space outside the SFR region
BIT	Assign to the BIT address space outside the SFR region
NVDATA	Assign to the NVDATA address space
NVBIT	Assign to the NVBIT address space

segment_type	**Description**
TABLE	Assign to the TABLE address space

boundary_attr

This field specifies the boundary attribute for the relocatable segment—that is, the boundary alignment for its starting offset. This specification can be either a symbol representing the boundary alignment type or an integer constant. The following lists the choices available and the segment types which support them.

boundary_attr	**Description**	**Segment Types**
UNIT	2-byte boundary for a CODE segment, 1-byte boundary for a TABLE, DATA, or NVDATA segment, 1-bit boundary for a BIT or NVBIT segment.	There are no restrictions.
WORD	2-byte boundary.	CODE, TABLE, DATA, or NVDATA
Integer constant (1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, or 2048)	Specified boundary in bytes for a CODE, TABLE, DATA, or NVDATA segment and in bits for a BIT or NVBIT segment.	The only restriction is for CODE segments. Specifying 1 produces an error message.

boundary_attr defaults to UNIT.

seg_attr

This field specifies the physical segment attribute for the logical segment.

seg_attr	**Description**
#pseg_addr	Assign the logical segment to the physical segment specified by *#pseg_addr*, a constant expression.
ANY	Place no restrictions on the physical segment for assigning the logical segment.

The physical segment specified by *#pseg_addr* must have memory available for the logical segment. If physical segment #1 contains only ROM, for example, the assembler issues warning messages for any DATA or NVDATA segments specifying physical segment #1.

The default depends on the segment type and memory or data model.

Segment Type	Default physical segment attribute for logical segment
CODE	#0 for the SMALL model; ANY for the LARGE model
TABLE, DATA, BIT, NVDATA, or NVBIT	#0 for the NEAR model; ANY for the FAR model

The assembler issues a warning message if the file contains the NOROMWIN directive and specifies a physical segment attribute of #0 for a segment symbol of type TABLE.

relocation_attr

This field specifies the special region attribute, the region for assigning the relocatable segment.

The following lists the choices available and the segment types which support them.

relocation_attr	Description	Segment Type
DYNAMIC	Defines the relocatable segment as a dynamic segment. The linker, after assigning all logical segments to the specified memory spaces, assigns the largest data memory regions remaining to dynamic segments.	DATA
NVRAM	Assigns the relocatable segment to a nonvolatile memory region. Relocatable CODE segments without this attribute are assigned to ROM.	CODE

■ Examples ■

```
CODE_SEG    SEGMENT CODE
DATA_SEG    SEGMENT DATA
NVDATA_SEG  SEGMENT NVDATA
```

The above are examples of the simplest definitions. The assembler assigns CODE_SEG to a ROM region in the CODE address space, DATA_SEG to a RAM region in the DATA address space, and NVDATA_SEG to a nonvolatile memory region in the NVDATA address space.

```
DATA_SEG1   SEGMENT DATA 1
DATA_SEG2   SEGMENT DATA 8
```

These next examples specify boundary attributes: a 1-byte boundary for DATA_SEG1 and an 8-byte boundary for DATA_SEG2. The assembler assigns both logical segments to RAM regions.

```
CODE_SEG     SEGMENT CODE #2 NVRAM
TABLE_SEG    SEGMENT TABLE 8 #1
DYNAMIC_SEG SEGMENT DATA 2 #3 DYNAMIC
```

These final examples combine such parameters as boundary attribute and special region attribute. The assembler assigns CODE_SEG to nonvolatile memory in physical segment #2, TABLE_SEG to an 8-byte boundary in a ROM region physical segment #1, and DYNAMIC_SEG to a 2-byte boundary in a RAM region in physical segment #3.

5.5.2 RSEG Directive

■ Syntax ■

RSEG *segment_symbol*

■ Description ■

This directive starts a relocatable segment using a segment symbol as its identifier. This symbol must defined with a SEGMENT directive prior to the RSEG directive.

A relocatable segment can be defined in multiple blocks, each starting with an RSEG directive specifying the same segment symbol. The assembler always sets the starting offset for a new relocatable segment to 0, so the location counter starts at 0 for the first block. The second and subsequent blocks inherit the final offset from the immediately preceding block.

■ Example ■

```
VAR_DATA  SEGMENT DATA WORD   ; Define segment symbols
CODE_0    SEGMENT CODE #0   ; Define segment symbols
CODE_1    SEGMENT CODE #1   ; Define segment symbols

          RSEG    VAR_DATA    ; Start relocatable DATA segment
BUF1:     DS      4H

          RSEG    CODE_0      ; Start relocatable CODE segment
SUB1:     MOV     ER0,    #00H
          MOV     ER2,    ER0
          LEA     BUF1
          ST      XR0,    [EA]
          RT
          .
          .
          .
```

```
        RSEG    VAR_DATA    ; Continue relocatable segment VAR_DATA
                            ; defined above
BUF2:   DS      10H

        RSEG    CODE_1      ; Start new relocatable CODE segment
SUB2:   MOV     ER0,    #00H
        ST      ER0,    BUF2
        .
        .
        .
```

This example defines one relocatable DATA segment and two relocatable CODE segments.

The relocatable DATA segment VAR_DATA starts at a 2-byte boundary in the DATA address space. The second time that VAR_DATA appears in a RSEG directive, the relocatable segment inherits as its starting offset the final offset from the first block, so the label BUF2 therefore has the offset 4H.

CODE_0 and CODE_1 represent separate relocatable segments in the CODE address space—in physical segments #0 and #1, respectively.

5.5.3 STACKSEG Directive

■ Syntax ■

STACKSEG *stack_size*

■ Description ■

This directive defines the stack segment with the specified size, a constant expression that does not contain a forward reference.

The size must be even. Specifying an odd value produces a warning message and causes the assembler to boost the size one to the next even number.

This directive causes the assembler to automatically generate the stack segment $STACK. If 0 is specified for the stack_size operand of a STACKSEG directive, only the segment symbol $STACK is defined, but no stack area is allocated. Note that the name of this relocatable segment is not allowed as the operand in a RSEG directive.

■ Aside ■

The initial value for the stack pointer—that is, one past the last address in the stack segment—is available as the symbol _$$SP. Before this symbol can be referenced, however, the program must declare it with an EXTRN directive.

```
STACKSEG    200H          ; Define stack segment ($STACK)
EXTRN DATA NEAR:_$$SP     ; Declare _$$SP external
CSEG    AT  00H           ; Reset vectors start at offset 0
DW      _$$SP             ; Define initial value for stack pointer
```

This example reserves 200H bytes for the stack and obtains the end address for storage at offset 0 as the initial value for the stack pointer by declaring _$$SP external with an EXTRN directive and then referencing that address.

5.6 Address Control Directives

5.6.1 ORG Directive

■ Syntax ■

ORG *address*

■ Description ■

This directive resets the location counter for the current logical segment to the specified address.

The exact details depends on whether the ORG directive is in an absolute segment or a relocatable one.

The following discusses each case individually.

(1) Absolute segment

The operand (address) must be a constant expression that does not contain a forward reference. The offset must be greater than or equal to that for the current logical segment. If the constant expression is of type address, it must also be within the corresponding address space—that is, its physical segment address must match that for the segment.

■ Example ■

```
CSEG #1 AT 1000H
.
.
.
ORG 1030H
.
.
.
ORG 1100H
.
.
.
ORG 200H   ; Error
```

This example uses ORG directives in an absolute CODE segment in physical segment #1. Their operands must be at least 1000H, the starting offset. The assembler therefore issues an error message for the last, with an operand of only 200H.

(2) Relocatable segment

The operand (address) must be a simple expression that does not contain a forward reference. Any

simple relocatable symbols in this operand must be in the current relocatable segment.

A constant expression represents an offset relative to the start of the relocatable segment.

■ Example ■

```
DATASEG SEGMENT DATA
        RSEG    DATASEG
LABEL1: DS      10H
        ORG     LABEL1+30H
LABEL2: DS      10H
        ORG     100H
LABEL3: DS      10H
```

This example uses two ORG directives within the same relocatable segment.

The first has as its operand a simple relocatable symbol based on a label in the current relocatable segment.

The constant expression (100H) in the second represents an offset relative to the start of the relocatable segment.

5.6.2 ALIGN Directive

■ Syntax ■

ALIGN

■ Description ■

This directive adjusts the current location so that the next data word or label falls on word boundary. If the current location is odd, it skips a byte. Otherwise, the current location is already a word boundary, so this directive does nothing.

This directive is available in CODE, TABLE, DATA, and NVDATA segments.

This directive is only available in absolute segments and relocatable segments with a boundary attribute 2 or higher. Using it in a relocatable segment with a boundary attribute of 1 produces an error message because the assembler is unable to determine whether the current location is odd or even.

```
Address    Source
-------                  CSEG   AT  200H
00:0200    START:  L     ER0,    STR1START
00:0204            L     R1,     [ER0]
```

```
-------             TSEG    AT  300H
00:0300     STR1:   DB      "OddSize"
00:0307             ALIGN
00:0308     STR1START:
00:0308             DW      STR1
```

This example initializes an odd number of bytes in a TABLE segment with a DB directive. Following this with an ALIGN directive forces the label STR1START to fall on an even address.

Aside on automatic alignment for CODE segments

CODE segments feature automatic alignment, which forces the location counter to always be an even value even without ALIGN directives.

Specifying an odd address for the location counter with a CSEG or ORG directive or allocating an odd number of bytes with DB or DS directives produces a warning message and causes the assembler to boost the current location one byte to the next word boundary.

5.6.3 DS Directive

■ Syntax ■

[*label* :] DS *size*

■ Description ■

This directive reserves the specified number of bytes in the current logical segment in the current address space by adding that number to the location counter for the logical segment.

The operand must be a constant expression that does not contain a forward reference.

This directive is available in CODE, TABLE, DATA, and NVDATA segments.

In a CODE segment, the value must be even. Specifying an odd value produces a warning message from the assembler.

```
DATASEG SEGMENT DATA
        RSEG    DATASEG
BUF:    DS      10H

CODESEG SEGMENT CODE
        RSEG    CODESEG
        MOV     ER0,    #00H
        ST      ER0,    BUF
```

This example reserves a 10H-byte region in relocatable DATA segment DATASEG.

5.6.4 DBIT Directive

■ Syntax ■

[*label* :] DBIT *size*

■ Description ■

This directive reserves the specified number of bits in the current logical segment in the current address space by adding that number to the location counter for the segment.

The operand must be a constant expression that does not contain a forward reference.

This directive is available in BIT and NVBIT segments.

■ Example ■

```
BITSEG  SEGMENT BIT
        RSEG    BITSEG
FLAG:   DBIT    8

CODESEG SEGMENT CODE
        RSEG    CODESEG
        SB      FLAG
```

This example reserves an 8-bit region in the relocatable segment BITSEG.

5.7 Code Initialization Directives

5.7.1 DB Directive

■ Syntax ■

[*label* :] DB { *expression* | *string_constant* | *duplicate_expression*}

[, { *expression* | *string_constant* | *duplicate_expression* }] …

■ Description ■

This directive initializes memory one byte at a time.

Each operand can be a general expression (*expression*), a string constant (*string_constant*), or a duplicate expression (*duplicate_expression*). There are no limits on the number of operands. A general expression can contain forward references, but its value must fit within a single byte—that is, must be between -255 and +255.

A string constant produces a series of bytes with the individual characters appearing in the specified order.

A duplicate expression fills the specified number of bytes in a contiguous address range with the same value. Duplicate expressions have the following syntax.

■ Duplicate Expression Syntax ■

repeat DUP *expression*

The field *repeat* specifies the repetition count; *expression*, the data byte for the initialization.

■ Aside ■

This directive is available in CODE, TABLE, and NVDATA segments.

■ Example ■

```
TABLESEG SEGMENT TABLE
         RSEG    TABLESEG
CHAR_TABLE:
         DB      'A', 'B', 'C', 'D', 'E', 'F'
STRING:
         DB      "String"
INIT_TABLE:
         DB      4 DUP 10H  ; Same as DB 10H, 10H, 10H, 10H
```

This example uses initializations of all three types: general expressions, a string constant, and a duplicate

expression.

5.7.2 DW Directive

■ Syntax ■

[*label* :] DW { *expression* | *duplicate_expression*} [, { *expression* | *duplicate_expression* }] …

■ Description ■

This directive initializes memory one word at a time.

Each operand can be a general expression (*expression*) or a duplicate expression (*duplicate_expression*). There are no limits on the number of operands.

A general expression can contain forward references, but its value must fit within a single word—that is, must be between -65535 and +65535.

A duplicate expression fills the specified number of words in a contiguous address range with the same value. Duplicate expressions have the following syntax.

■ Duplicate Expression Syntax ■

repeat DUP *expression*

The field *repeat* specifies the repetition count; *expression*, the data word for the initialization.

■ Aside ■

This directive is available in CODE, TABLE, and NVDATA segments.

■ Example ■

```
TABLESEG SEGMENT TABLE
         RSEG    TABLESEG
TABLE1:
         DW      -3, -2, -1, 0, 1, 2, 3
TABLE2:
         DW      3 DUP 0FFFFH  ; Same as DB 0FFFFH, 0FFFFH, 0FFFFH
```

This example uses initializations of both types: general expressions and a duplicate expression.

5.7.3 CHKDBDW / NOCHKDBDW Directives

■ Syntax ■

CHKDBDW

NOCHKDBDW

■ Corresponding RASU8 Option ■

/ZC

■ Description ■

The CHKDBDW and NOCHKDBDW directives specify whether to output a warning when a program code is placed in an inaccessible range with DB and DW directives or there is such a possibility.

When a CHKDBDW directive is specified, a check is performed on whether there is a possibility that a program code is placed in an inaccessible range within a range from the next line to the line on which the next NOCHKDBDW directive is specified. This check is performed regardless of whether or not the /ZC option is specified.

When a NOCHKDBDW directive is specified, a check is not performed on whether there is a possibility that a program code is placed in an inaccessible range within a range from the next line to the line on which the next CHKDBDW directive is specified. This check is not performed regardless of whether or not the /ZC option is specified.

■ Example ■

```
CSEG AT 0:0000H
NOCHKDBDW           ; Does not perform the subsequent DB/DW directive
                    ; check.
DW    0F000H        ; Initial value of SP (stack pointer)
DW    PROG_ENTRY    ; Initial value of PC (program pointer)
DW    BRK_ENTRY     ; Branch destination when a brk instruction is
                    ; executed while ELEVEL is 0 or 1

CSEG AT 0:0008H
DW    NMI_ENTRY     ; Non-maskable interrupt
```

This example shows the method of describing a vector table. It is not necessary to access a vector table with an L instruction. Therefore, suppress a warning check by describing a NOCHKDBDW directive.

■ **Example** ■

```
        CHKDBDW             ; Performs the subsequent DB/DW directive
                            ; check.
        CSEG   AT 0:9000H
SUB_ROUTINE1:
        MOV  R0, #3         ; Input values (0 to 7 are assumed in this
                            ; example)

        /* Store the jump table address in ER2.    */
        /* Fix the physical segment address to #0. */
        MOV  R2, #BYTE1 OFFSET JMP_TABLE
        MOV  R3, #BYTE2 OFFSET JMP_TABLE

        ADD  R0, R0         ; Doubles the input value since data
                            ; consists of 2 bytes.
        ADD  R2, R0         ; Adds input value × 2 to the address
        ADDC R3, #0
        L    ER0, [ER2]     ; Completes a read of the jump address.
        B    ER0

JMP_TABLE:                  ; Jump table
        DW   JMP_ADDR0      ; Warning 42 occurs
        DW   JMP_ADDR1      ; Warning 42 occurs
        DW   JMP_ADDR2      ; Warning 42 occurs
        DW   JMP_ADDR3      ; Warning 42 occurs
        DW   JMP_ADDR4      ; Warning 42 occurs
        DW   JMP_ADDR5      ; Warning 42 occurs
        DW   JMP_ADDR6      ; Warning 42 occurs
        DW   JMP_ADDR7      ; Warning 42 occurs
             :
```

When describing table data such as a jump table in a CODE segment, perform a warning check for data that must be accessed from a program by describing a CHKDBDW directive. In the example above, because the DW directive after the label JMP_TABLE is placed outside the range of ROM WINDOW, a warning is output. This warning output can be prevented by moving the jump table to the ROM WINDOW area or physical segment address #1.

5.8 Optimized Branch Directives

The nX-U8 architecture provides long and short versions of most branch instructions. The GJMP and GBcond directives allow the assembler to choose the version that is optimal for the branch target address or the distance to the branch target.

5.8.1 GJMP Directive

■ Syntax ■

[*label* :] GJMP *symbol*

■ Description ■

This directive converts to the optimal instruction (B or BAL) for an unconditional branch to the specified branch target.

If symbol is a forward symbol reference, however, the assembler uses the B instruction unless the /G option appears on the command line. Then it optimizes forward references as well.

■ Aside ■

This directive can only appear in CODE segments.

The operand must be of usage type CODE, NONE, or NUMBER.

■ Example ■

```
        CSEG  AT  1000H
LABEL1:
        DS      40H
        GJMP    LABEL1  ; Converts to BAL instruction
        .
        .
        .
        CSEG  AT  2000H
        GJMP    LABEL1  ; Converts to B instruction
```

This example produces a BAL instruction for the first GJMP directive because the distance to the target (LABEL1) is within the supported range (-128 to +127 words). The second GJMP directive is outside this range, so converts to a B instruction.

5.8.2 GB*cond* Directives

■ Syntax ■

[*label* :] GBGT *symbol*
[*label* :] GBGE *symbol*
[*label* :] GBNC *symbol*
[*label* :] GBEQ *symbol*
[*label* :] GBZ *symbol*
[*label* :] GBNE *symbol*
[*label* :] GBNZ *symbol*
[*label* :] GBLE *symbol*
[*label* :] GBLT *symbol*
[*label* :] GBCY *symbol*
[*label* :] GBPS *symbol*
[*label* :] GBNS *symbol*
[*label* :] GBLTS *symbol*
[*label* :] GBLES *symbol*
[*label* :] GBGTS *symbol*
[*label* :] GBGES *symbol*
[*label* :] GBOV *symbol*
[*label* :] GBNV *symbol*

■ Description ■

These directives produce the optimal conditional branch to the specified branch target.

Using them without the /G option on the command line, however, produces an error message from the assembler.

These directives convert to either a single instruction or an instruction sequence.

B*cond symbol*

or

B*revcond branch_address*

B *symbol*

branch_address:

Here Brevcond and Bcond refer to the members of a pair of exact opposite conditions from the following Table.

Branch instruction	Exact opposite branch instruction
BGT	BLE
BGE or BNC	BLT or BCY
BEQ or BZ	BNE or BNZ
BPS	BNS
BLTS	BGES
BLES	BGTS
BOV	BNV

■ Aside ■

This directive can only appear in CODE segments.

The field symbol must be of usage type CODE, NONE, or NUMBER.

■ Example ■

```
        CSEG  AT  1000H
LABEL1:
        DS      40H
        GBLT    LABEL1  ; Converts to BLT instruction
        .
        .
        .
        CSEG  AT  2000H
        GBLT    LABEL1  ; Converts to BGE 2006H, B LABEL1
```

This example produces a BLT instruction for the first GBLT directive because the distance to the target (LABEL1) is within the supported range (-128 to +127 words). The second GBLT directive is outside this range, so converts to a BGE $+6, B sequence.

5.9 Linkage Control Directives

These directives are primarily used in programs consisting of multiple source code files.

5.9.1 Splitting a Program into Multiple Files

When the program under development consists of multiple source code files, referencing a symbol defined in one file from another requires the following preparations.

(1) The file defining the symbol must declare it public so that other files can reference it.

(2) Any file referencing the symbol must declare it external so that the linker looks for it in other files.

Symbols declared public are called public symbols; ones declared external, external symbols. The directives used are PUBLIC and EXTRN, respectively.

If a source code file declares an external symbol, there must be a public symbol with the same name in another source code file.

There are also communal symbols, defined with COMM directives, that combine features of both public and external symbols. Defining communal symbols with the same name in multiple source code files produces a single symbol as the name for a single memory region common to all those files.

Both public and communal symbols can be referenced from multiple files, so are sometimes called global symbols.

Segment symbols, defined with SEGMENT directives, cannot be declared external with EXTRN directives. Using a segment symbol in multiple source code files therefore requires defining that symbol in each file with a SEGMENT directive.

The rest of this Section describes public, external, communal, and segment symbols individually.

5.9.2 PUBLIC Directive

■ Syntax ■

PUBLIC *symbol* [*symbol* …]

■ Description ■

This directive makes a local symbol public and thus available for reference from other source code files.

The field symbol is the name of the local symbol. Note that the local symbol definition can appear either before or after this directive.

This directive accepts multiple symbols as operands.

■ Aside ■

Referencing a public symbol from another source code file requires an EXTRN directive in the other file declaring the same name as an external symbol.

Declaring the same symbol name public in multiple source code files is not allowed.

A public symbol defined and redefined with SET directives has the value appearing in the last definition in the file.

■ Example ■

```
GLOBAL_NUMBER   EQU  1
DATASEG SEGMENT DATA 2
        RSEG    DATASEG
GLOBAL_DATA:
        DS      2
PUBLIC  GLOBAL_NUMBER GLOBAL_DATA
```

This example makes the absolute symbol GLOBAL_NUMBER and the simple relocatable symbol GLOBAL_DATA public.

5.9.3 EXTRN Directive

■ Syntax ■

EXTRN *usage_type* [*attribute*] : *symbol* [*symbol* …] [*usage_type* [*attribute*] : *symbol* [*symbol* …]] …

■ Description ■

This directive declares an external symbol.

The field *usage_type* specifies the usage type for the external symbol. This specification remains in effect until a new usage type specification appears in the directive.

It must be one of the following.

usage_type	**Description**
CODE	Symbol representing a byte address in the CODE address space
DATA	Symbol representing a byte address in the DATA address space
BIT	Symbol representing a bit address in the BIT address space
NVDATA	Symbol representing a byte address in the NVDATA address space
NVBIT	Symbol representing a bit address in the NVBIT address space

usage_type	**Description**
TABLE	Symbol representing a byte address in the TABLE address space
TBIT	Symbol representing a bit address in the TBIT address space
NONE	Address symbol with unspecified address space
NUMBER	Symbol representing a numerical value

The field *symbol* specifies the name of the external symbol. This symbol must be a symbol declared public or communal in another source code file.

The field *attribute* specifies the physical segment attribute for the external symbol. The settings have the following meanings.

attribute	**Description**
NEAR	#0 is the physical segment attribute for the corresponding symbol.
FAR	ANY is the physical segment attribute for the corresponding symbol.

The field *attribute* is available for all usage types except NUMBER. Symbols of usage type NUMBER do not have a physical segment attribute because they represent numerical values, not addresses.

The default for usage types TABLE, TBIT, DATA, BIT, NVDATA, NVBIT, and NONE depends on the data model: #0 for the NEAR model and ANY for the FAR model. That for usage type CODE depends on the memory model: #0 for the SMALL model and ANY for the LARGE model.

■ Aside ■

This directive cannot specify a symbol already defined in the current source code file.

This directive cannot reference segment symbols.

An external symbol and the corresponding public symbol must match both in usage type and physical segment attribute. The linker issues an error message if there is any mismatch.

■ **Example** ■

```
;File 1
    DSEG    AT  0:8000H
NEAR_VAR:
    DS      2H
    DSEG    AT  7:1000H
FAR_VAR:
    DS      2H
PUBLIC NERA_VAR FAR_VAR
```

```
;File 2
EXTRN DATA NEAR : NEAR_VAR
EXTRN DATA FAR  : FAR_VAR

    CSEG    AT  1000H
    MOV     ER0,    #00H
    ST      ER0,    NEAR_VAR
    ST      ER0,    FAR_VAR
```

This example has external declarations in file 2 for symbols declared public in file 1.

5.9.4 COMM Directive

This directive defines a communal symbol, the name for a data region shared by multiple source code files.

This symbol represents the starting address for the region. The linker decides this address.

Communal symbols resemble relocatable segments, but do not support label definitions within the allocated region or initialization. They also represent a single region common to all source code files, not the independent regions that relocatable segments represent in each file.

The following shows the COMM directive syntax.

■ **Syntax** ■

communal_symbol COMM *segment_type size* [*seg_attr*]

■ **Description** ■

This syntax closely resembles that for the SEGMENT directive except that

- The field segment type is immediately followed a field specifying a size for the region.
- There is no boundary attribute.

segment_type

This field specifies the address space for the communal symbol: DATA, BIT, NVDATA, NVBIT, or TABLE.

segment_type	**Description**
DATA	Assign to the DATA address space outside the SFR region
BIT	Assign to the BIT address space outside the SFR region
NVDATA	Assign to the NVDATA address space
NVBIT	Assign to the NVBIT address space
TABLE	Assign to the TABLE address space

seg_attr

This field specifies the physical segment attribute for the communal symbol.

seg_attr	**Description**
#pseg_addr	Assign the communal symbol to the physical segment specified by *#pseg_addr*, a constant expression.
ANY	Place no restrictions on the physical segment for assigning the communal symbol.

The default depends on the data model: #0 for the SMALL model; ANY for the LARGE model.

The assembler issues a warning message if the file contains the NOROMWIN directive and specifies a physical segment attribute of #0 for a communal symbol of type TABLE.

size

This field is an integer constant specifying a size for the communal symbol region. The unit for this size depends on the segment type: bytes for TABLE, DATA, and NVDATA and bits for BIT and NVBIT.

This directive differs from the SEGMENT directive in that there is no boundary attribute specification. The only alignment adjustment is to a word boundary for segment types TABLE, DATA, and NVDATA with a size greater than 1.

Defining communal symbols with the same name in multiple source code files allocates a single region common to those files.

```
COM_AREA COMM DATA 2
```

Including the above line in multiple source code files, for example, allows them to share a 2-byte region in the DATA address space at the address COM_AREA. What is important to note here, however, is that

the communal symbols COM_AREA from each source code file all refer to the same two bytes.

■ **Aside** ■

Defining the same communal symbol twice in a single source code file produces an error message.

If the individual source code files specify different sizes, the linker uses the maximum as the final size for the region.

It is perfectly acceptable for one source code file to reference a communal symbol defined in another file with an EXTRN directive.

If a COMM directive names a symbol declared public in another source code file, the linker ignores the size specification and treats the statement simply as a reference to that symbol—in other words, exactly the same as an EXTRN directive. Although the C compiler CCU8 generates this sort of reference in its assembly language output, we do not recommend the practice to programmers working in assembly language.

■ **Example** ■

```
          TYPE (M610001)
GL_BUF1 COMM DATA 100H
GL_BITF COMM BIT  4
          .
          .
          .
          L        ER0,    GL_BUF1
          SB       GL_BITF+2
```

This example defines two communal symbols GL_BUF1 and GL_BITF, which it then uses as operands in microcontroller instructions. GL_BUF1 represents a 100H-byte region in the DATA address space; GL_BITF, a 4-bit region in the BIT address space.

5.9.5 Examples Using Public, External, and Communal Symbols

5.9.5.1 Referencing Public Symbols as External Symbols

The following gives an example of using PUBLIC and EXTRN directives to allow one source code file to reference a symbol defined in another.

```
;Source code file 1
        TYPE (M610001)
PUBLIC BUF_SIZE
PUBLIC BUF
BUF_SIZE EQU 20H

DATA_SEG SEGMENT DATA 2
        RSEG    DATA_SEG
BUF:    DS      BUF_SIZE
```

```
;Source code file 2
        TYPE (M610001)
EXTRN NUMBER: BUF_SIZE
EXTRN DATA NEAR: BUF

CODE_SEG SEGMENT CODE
        RSEG    CODE_SEG
PROG1:
        LEA     OFFSET BUF
LOOP:
        MOV     R0,     #BUF_SIZE
        MOV     R1,     #00H
        ST      R1,     [EA+]
        ADD     R0,     #-1
        BNZ     LOOP
        RT
```

This example has source code file 2 using BUF_SIZE and BUF, two symbols defined in source code file 1. Source code file 1 makes the two symbols public with PUBLIC directives. Source code file 2 declares them external with EXTRN directives.

5.9.5.2 Sharing Communal Symbols among Source Code Files

The following gives an example of using COMM directives to define a common data region shared by multiple source code files.

```
;Source code file 1
        TYPE (M610001)

GL_BUF1 COMM DATA 2
GL_BUF2 COMM DATA 2
GL_BUF3 COMM DATA 2

CODE_SEG SEGMENT CODE
        RSEG    CODE_SEG
        L       ER0,    GL_BUF1
        ST      ER0,    GL_BUF2
        ST      ER0,    GL_BUF3
```

```
;Source code file 2
        TYPE (M610001)

GL_BUF1 COMM DATA 2
GL_BUF2 COMM DATA 2
GL_BUF3 COMM DATA 4  ; Different size from source code file 1

CODE_SEG SEGMENT CODE
        RSEG    CODE_SEG
        L       ER0,    GL_BUF1
        L       ER2,    GL_BUF2
        ST      ER0,    GL_BUF3
        ST      ER2,    GL_BUF3+2
```

This example defines three communal symbols (GL_BUF1, GL_BUF2, and GL_BUF3) in two source code files. These symbols represent three data regions common to both source code files. GL_BUF1 and GL_BUF2 both contain two bytes each. The source code files specify different sizes for GL_BUF3, so the linker uses the larger, four bytes.

5.9.5.3 Referencing Communal Symbols as External Symbols

The following gives an example of using EXTRN directives to reference communal symbols defined in another source code file.

```
;Source code file 1
        TYPE (M610001)

GL_BUF1 COMM DATA 2 #0
GL_BUF2 COMM DATA 2 #0
GL_BUF3 COMM DATA 2 ANY
```

```
;Source code file 2
        TYPE (M610001)

EXTRN   DATA NEAR : GL_BUF1 GL_BUF2
EXTRN   DATA FAR  : GL_BUF3

CODE_SEG SEGMENT CODE
        RSEG    CODE_SEG
        L       ER0,    GL_BUF1
        L       ER2,    GL_BUF2
        ST      ER0,    GL_BUF3
```

This example has source code file 2 using EXTRN directives to reference three communal symbols (GL_BUF1, GL_BUF2, and GL_BUF3) defined in source code file 1.

5.9.6 Using Partial Segments

Multiple source code files can define relocatable segments with the same name, producing what are called partial segments. The linker merges them into a single logical segment for assignment to a contiguous memory range.

The following is an example.

```
;Source code file 1
        TYPE (M610001)
        ROMWINDOW 0, 3FFFH
DATA_VAR    SEGMENT DATA  2 #0
INIT_TABLE  SEGMENT TABLE 2 #0

        RSEG    DATA_VAR   ; Linked with DATA_VAR from source code file
                           ; 2
VAR1:   DS      4
VAR2:   DS      10H

        RSEG    INIT_TABLE ; Linked with INIT_TABLE from source code
                           ; file 2
        DW      5678H, 1234H
        DB      0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
PUBLIC  VAR1 VAR2
```

```
;Source code file 2
        TYPE (M610001)
        ROMWINDOW 0, 3FFFH
DATA_VAR    SEGMENT DATA  2 #0
INIT_TABLE  SEGMENT TABLE 2 #0
CODE_SEG    SEGMENT CODE

        RSEG    DATA_VAR   ; Linked with DATA_VAR from source code file
                           ; 1
VAR3:   DS      2
VAR4:   DS      4

        RSEG    INIT_TABLE ; Linked with INIT_TABLE from source code
                           ; file 1
        DW      0
        DW      0FFFFH, 0FFFFH

        RSEG    CODE_SEG
        L       ER0,    SIZE DATA_VAR ; Segment size after linking
        LEA     OFFSET  DATA_VAR      ; Starting address after linking
        MOV     R2,     #BYTE1 OFFSET INIT_TABLE ; Starting address
                                                 ; after linking
        MOV     R3,     #BYTE2 OFFSET INIT_TABLE ; Starting address
                                                 ; after linking
LOOP:
        L       ER4,    [ER2]
        ST      ER4,    [EA+]
        ADD     ER2,    #2
        ADD     ER0,    #-2
        BNE     LOOP
```

This example defines segment symbols DATA_VAR and INIT_TABLE in both source code files. The linker merges the partial segments from each file into logical segments DATA_VAR and INIT_TABLE for assignment to contiguous memory ranges. It links these partial segments in the order in which they appear as it processes the object files on the command line.

The relocatable segment CODE_SEG references the segment symbols DATA_VAR and INIT_TABLE. These symbols represent the starting addresses, after linking, for the two segments.

5.10 File Read Directive

5.10.1 INCLUDE Directive

■ Syntax ■

INCLUDE (*include_file*)

■ Description ■

This directive reads in the specified include file, replacing the directive with the file contents.

An include file can itself contain INCLUDE directives for inserting still more files. The limit for such nesting is eight levels.

The assembler interprets an END directive in an include file as the end of that file, so returns to processing the specifying file.

■ Aside ■

The assembler's /I option is for adding directories to the search path for include files.

■ Example ■

```
;Source code file
    INCLUDE(DEFINE.INC)
    CSEG AT 0:0H
    DW      _$$SP
    DW      START
    .
    .
    .
```

```
;Include file (DEFINE.INC)
    TYPE (M610001)
    MODEL SMALL, NEAR
    ROMWINDOW 0, 7FFFH
```

The include file, DEFINE.INC, in this example lumps together the assembler initialization directives (TYPE, MODEL, and ROMWINDOW) for the project.

5.11 Macro Definition Directive

5.11.1 DEFINE Directive

■ Syntax ■

DEFINE *symbol* ”*macro_body*”

■ Description ■

This directive assigns a text string to a macro symbol. The assembler then replaces any subsequent appearance of that symbol in source code statements with that text string.

The text string can be up to 255 bytes long and can itself contain macro symbols for immediate replacement at that point. The limit for such nesting is eight levels.

■ Aside ■

Macro symbols can only be referenced after the DEFINE directives defining them.

■ Example ■

```
DEFINE  CODESEG     ”SEGMENT CODE”
DEFINE  CLR4BYTE    ”MOV ER0, #0\nST ER0, [EA+]\nST ER0, [EA]”

PROG1   CODESEG     ;PROG1   SEGMENT CODE

        RSEG    PROG1
        LEA     8000H
        CLR4BYTE    ;MOV ER0, #0
                    ;ST  ER0, [EA+]
                    ;ST  ER0, [EA]
```

This example defines two macro symbols CODESEG and CLR4BYTE. The CLR4BYTE macro definition illustrates how a single macro symbol can include multiple instructions. The comments give the results after replacement.

5.12 Conditional Assembly Directives

Conditional assembly means assembling a block of source code statements only when certain conditions are met. A single source program can therefore serve multiple objectives.

This facility uses conditional assembly directives, which share the following syntax.

IF*xxx conditional_operand*

true_conditional_body

[ELSE

false_conditional_body]

ENDIF

Here IF*xxx* denotes the conditional assembly directives IF, IFDEF, or IFNDEF.

The field *conditional_operand* is an expression or symbol whose truth value specifies the condition. The contents vary with the conditional assembly directive.

The fields *true_conditional_body* and *false_conditional_body* represent two source code blocks. The assembler uses the first only if the truth value from *conditional_operand* is TRUE. If the result is FALSE, however, the assembler skips that block and uses the second, if present with an ELSE directive.

The blocks *true_conditional_body* and *false_conditional_body* can themselves contain conditional assembly directives. The limit for such nesting is 15 levels.

5.12.1 IF Directive

■ Syntax ■

IF *expression*

■ Description ■

The conditional operand must be a constant expression that does not contain a forward reference. The conditional operand yields a truth value of TRUE if expression successfully evaluates to a nonzero value and FALSE otherwise. Note that a forward reference, syntax error, or other error in expression yields a truth value of FALSE.

■ 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  8000H
BUF1:   DS      BUF_SIZE1
BUF2:   DS      BUF_SIZE2
```

This example uses the condition expression PROG_SW == 2. This expression evaluates to TRUE because the program starts by setting PROG_SW to 2. As a result, the two buffer sizes are 100H and 200H, respectively.

5.12.2 IFDEF Directive

■ Syntax ■

IFDEF *symbol*

■ Description ■

The conditional operand is any symbol except a reserved word. The truth value is TRUE only if the symbol has been defined previous to this directive in the current source code file. If the program does not define the symbol or defines it after this directive, the truth value is FALSE.

■ Example ■

```
PROG_SW2 EQU 0
         .
         .
         .
IFDEF PROG_SW1
        INCLUDE (INIT1.INC)
ELSE
        INCLUDE (INIT2.INC)
ENDIF
```

This example does not define PROG_SW1, so the condition is FALSE, and the assembler reads in the include file INIT2.INC.

■ Note ■

Using a macro symbol as a conditional assembly directive operand requires particular care. Consider, for example, the following, intended to check whether the symbol SW is defined.

```
DEFINE  SW      "SYM1"
IFDEF   SW  ; Interpreted, after substitution, as IFDEF SYM1
        INCLUDE (FILE1.INC)
ELSE
        INCLUDE (FILE2.INC)
ENDIF
```

Yes, it is defined, but the assembler performs the macro substitution first, so the actual test is for the macro body, SYM1. That symbol is not defined, so the condition is FALSE, and the assembler reads in the include file FILE2.INC.

5.12.3 IFNDEF Directive

■ Syntax ■

IFNDEF *symbol*

■ Description ■

The conditional operand is any symbol except a reserved word.

This directive reverses the testing condition of the IFDEF directive. If the symbol has been defined previous to this directive in the current source code file, the truth value is FALSE. Otherwise, it is TRUE.

■ Example ■

```
PROG_SW2 EQU 0
         .
         .
         .
IFNDEF PROG_SW1
         INCLUDE (INIT1.INC)
ELSE
         INCLUDE (INIT2.INC)
ENDIF
```

The operand in this example is PROG_SW1, which is not defined, so the condition is TRUE, and the assembler reads in the include file INIT1.INC.

5.13 C Source Level Debugging Information Directives

These directives specify C language source level debugging information for source programs originally written in C. The C compiler CCU8 /SD option automatically generates them. They are not for use by programmers.

This Section describes these directives for information purposes only. Be aware that manually adding them to regular assembly language source code files or modifying the ones in CCU8 assembly language output can interfere with proper assembly and later debugging operations.

5.13.1 CFILE Directive

■ Syntax ■

CFILE *file_id total_line* ”*filename*”

■ Description ■

This directive gives the name and other information on a C source code file.

5.13.2 CFUNCTION and CFUNCTIONEND Directives

■ Syntax ■

CFUNCTION *fn_id*

CFUNCTIONEND *fn_id*

■ Description ■

These directives indicate the start and end of a C function.

5.13.3 CARGUMENT Directive

■ Syntax ■

CARGUMENT *attrib size offset* ”*variable_name*” *hierarchy*

■ Description ■

This directive gives information on a C function argument.

5.13.4 CBLOCK and CBLOCKEND Directives

■ Syntax ■

CBLOCK *fn_id block_id c_source_line*

CBLOCKEND *fn_id block_id c_source_line*

■ Description ■

These directives indicate the start and end of a function or a block of statements in the C source program.

5.13.5 CLABEL Directive

■ Syntax ■

CLABEL *label_no* ”*label_name*”

■ Description ■

This directive links a label from the C source code file with the one that the compiler used in its assembly language output.

5.13.6 CLINE and CLINEA Directives

■ Syntax ■

CLINE *line_attr line_no start_column end_column*

CLINEA *file_id line_attr line_no start_column end_column*

■ Description ■

This directive gives C source code file line number information.

5.13.7 CGLOBAL Directive

■ Syntax ■

CGLOBAL *usg_typ attrib size* ”*variable_name*” *hierarchy*

■ Description ■

This directive gives the name and other information on a global variable defined in the C source program.

The field *variable_name* gives the name of the variable. This name, plus an underscore (_) prefix,

appears in the CCU8 assembly language output as a public or communal symbol.

5.13.8 CSGLOBAL Directive

■ Syntax ■

CSGLOBAL *usg_typ attrib size* "*variable_name*" *hierarchy*

■ Description ■

This directive gives the name and other information on a static global variable defined in the C source program.

The field *variable_name* gives the name of the variable. This name, plus an underscore (_) prefix, appears in the CCU8 assembly language output as a local symbol.

5.13.9 CLOCAL Directive

■ Syntax ■

CLOCAL *attrib size offset block_id* "*variable_name*" *hierarchy*

■ Description ■

This directive gives the name and other information on a local variable defined in the C source program.

The field *variable_name* gives the name of the variable.

5.13.10 CSLOCAL Directive

■ Syntax ■

CSLOCAL *attrib size alias_no block_id* "*variable_name*" *hierarchy*

■ Description ■

This directive gives the name and other information on a static local variable defined in the C source program.

The field *variable_name* gives the name of the variable. The corresponding local symbol in the CCU8 assembly language output is the prefix _$ST followed by *alias_no* in decimal notation instead of the hexadecimal one that appears in this directive.

```
CSLOCAL 43H 0002H 000AH 0001H "static_local" 02H 00H 01H
```

The corresponding assembly language local symbol for the C static local symbol *static_local*, for

example, is _$ST10.

5.13.11 CSTRUCTTAG and CSTRUCTMEM Directives

■ Syntax ■

CSTRUCTTAG *fn_id block_id st_id total_mem total_size* ”*tag_name*”

CSTRUCTMEM *attrib size offset* ”*member_name*” *hierarchy*

■ Description ■

These directives give the name and other information on structures defined in the C source program. A CSTRUCTTAG directive specifies the tag; CSTRUCTMEM directives, the members for the immediately preceding CSTRUCTTAG.

5.13.12 CUNIONTAG and CUNIONMEM Directives

■ Syntax ■

CUNIONTAG *fn_id block_id un_id total_mem total_size* ”*tag_name*”

CUNIONMEM *attrib size* ”*member_name*” *hierarchy*

■ Description ■

These directives give information on unions defined in the C source program. A CUNIONTAG directive specifies the tag; CUNIONMEM directives, the members for the immediately preceding CUNIONTAG.

5.13.13 CENUMTAG and CENUMMEM Directives

■ Syntax ■

CENUMTAG *fn_id block_id emu_id total_mem* ”*tag_name*”

CENUMMEM *value* ”*member_name*”

■ Description ■

These directives give the name and other information on enumerations defined in the C source program. A CENUMTAG directive specifies the tag; CENUMMEM directives, the enumerators for the immediately preceding CENUMTAG.

5.13.14 CTYPEDEF Directive

■ Syntax ■

CTYPEDEF *fn_id block_id attrib* ”*type_name*” *hierarchy*

■ Description ■

This directive gives the name and other information on a user defined type defined with a typedef in the C source program.

5.13.15 CVERSION Directive

■ Syntax ■

CVERSION *version_number*

■ Description ■

This directive gives the CCU8 C compiler version information.

5.13.16 CRET Directive

■ Syntax ■

CRET *offset*

■ Description ■

This directive gives information on the offset from the stack pointer when the return address is saved to the stack.

5.14 Emulation Library Directive

This directive, automatically generated by the C compiler CCU8 in its assembly language output, is not for use by assembly language programmers.

This Section describes this directive for information purposes only. Be aware that manually adding it to regular assembly language source code files or modifying one in CCU8 assembly language output can interfere with proper linking.

5.14.1 FASTFLOAT Directive

■ Syntax ■

FASTFLOAT

■ Description ■

This directive specifies to the linker the faster emulation library for floating point arithmetic. Note, however, that this library uses only single-precision floating point arithmetic, trading accuracy for speed.

5.15 Listing Control Directives

These directives control file output, specify output file names, control list output for print files, and control formatting.

5.15.1 OBJ and NOOBJ Directives

■ Syntax ■

OBJ [(*object_file*)]

NOOBJ

■ Corresponding Assembler Options ■

/O[(*object_file*)]

/NO

■ Description ■

These directives control object file output.

The OBJ directive generates an object file. The optional operand specifies a name for it.

The NOOBJ directive skips the object file.

The default is to generate an object file. The default name for the object file is the source code file name with the file extension changed to .OBJ.

■ Aside ■

These directives can appear only once per file. Only the first to appear takes effect. The corresponding command line options take precedence over these directives.

5.15.2 PRN and NOPRN Directives

■ Syntax ■

PRN [(*print_file*)]

NOPRN

■ Corresponding Assembler Options ■

/PR[(*print_file*)]

/NPR

■ Description ■

These directives control print file output.

The PRN directive generates a print file. The optional operand specifies a name for it.

The NOPRN directive skips the print file.

The default is to generate a print file. The default name for the print file is the source code file name with the file extension changed to .PRN.

■ Aside ■

These directives can appear only once per file. Only the first to appear takes effect. The corresponding command line options take precedence over these directives.

5.15.3 ERR and NOERR Directives

■ Syntax ■

ERR [(*error_file*)]

NOERR

■ Corresponding Assembler Options ■

/E[(*error_file*)]

/NE

■ Description ■

These directives control error file output.

The ERR directive generates an error file. The optional operand specifies a name for it.

The NOERR directive skips the error file.

The default is to not generate this file, but send all error messages to the standard error output. The default name for the error file is the source code file name with the file extension changed to .ERR.

■ Aside ■

These directives can appear only once per file. Only the first to appear takes effect. The corresponding command line options take precedence over these directives.

5.15.4 DEBUG and NODEBUG Directives

■ Syntax ■

DEBUG

NODEBUG

■ Corresponding Assembler Options ■

/D

/ND

■ Description ■

These directives control assembly level debugging information output to the object file.

The DEBUG directive generates the debugging information; NODEBUG skips it.

The default is to not include the debugging information.

■ Aside ■

These directives can appear only once per file. Only the first to appear takes effect. The corresponding command line options take precedence over these directives.

5.15.5 LIST and NOLIST Directives

■ Syntax ■

LIST

NOLIST

■ Corresponding Assembler Options ■

/L

/NL

■ Description ■

These directives control assembly listing output to the print file.

The LIST directive enables output of both an assembly listing and the corresponding object code listing from the next line; NOLIST disables it from the next line. Together, they specify the ranges listed.

Note, however, that source code lines triggering error or warning messages are always sent to the assembly listing.

The default is to assume a LIST directive at the start of the program. If the program contains no LIST or NOLIST directives, therefore, the assembly listing covers all source statements.

■ Aside ■

Specifying the assembler's /L option lists all source statements up until the first NOLIST directive in the program, producing the same effect as a LIST directive at the start of the program.

The /NL option, in contrast, skips all source statements up until the first LIST directive in the program, producing the same effect as a NOLIST directive at the start of the program.

5.15.6 SYM and NOSYM Directives

■ Syntax ■

SYM

NOSYM

■ Corresponding Assembler Options ■

/S

/NS

■ Description ■

These directives control symbol table output to the print file. The symbol table provides detailed information on the user symbols appearing in the program.

The SYM directive generates the symbol table; NOSYM skips it.

The default is to not include the symbol table.

■ Aside ■

These directives can appear only once per file. Only the first to appear takes effect. The corresponding

command line options take precedence over these directives.

5.15.7 REF and NOREF Directives

■ Syntax ■

REF

NOREF

■ Corresponding Assembler Options ■

/R

/NR

■ Description ■

These directives control cross-reference listing output to the print file. This cross-reference listing gives user symbols defined in the program and the line numbers where they are used.

The REF directive enables output from the next line; NOREF disables it from the next line. Together, they specify the ranges listed.

The default is to assume a NOREF directive at the start of the program—unless the assembler's /R command line option is in effect.

■ Aside ■

Specifying the assembler's /R option lists all source statements up until the first NOREF directive in the program, producing the same effect as a REF directive at the start of the program.

The /NR option, in contrast, skips all source statements up until the first REF directive in the program, producing the same effect as a NOREF directive at the start of the program.

5.15.8 PAGE Directive

This directive functions differently with and without operands.

This Section describes these two functions individually.

5.15.8.1 PAGE Directive without Operands

■ Syntax ■

PAGE

■ Description ■

A PAGE directive with no operands forces a page break in the print file. The new page starts on the next line containing the PAGE directive.

■ Aside ■

This PAGE directive is ignored when a NOLIST directive is in effect.

5.15.8.2 PAGE Directive with Operands

■ Syntax ■

PAGE [*page_length*][, *page_width*]

■ Corresponding Assembler Options ■

/PL*page_length*

/PW*page_width*

■ Description ■

A PAGE directive with operands specifies the print file page length in lines and the page width in columns.

Both operands must be constant expressions that do not contain a forward reference. At least one must appear because omitting both forces a page break.

The valid range for page length values is 10 to 65535; for page width, 79 to 132. Specifying a value outside the valid range produces the nearest endpoint: 10 for a page length less than 10, 65535 for a page length greater than 65535, 79 for a page width less than 79, 132 for a page width greater than 132.

The default settings are 60 lines and 79 columns, respectively.

■ Aside ■

This variant can appear only once per file.

5.15.9 DATE Directive

■ Syntax ■

DATE ”*charcter_string*”

■ Description ■

This directive specifies a text string for the print file’s date field.

The limit is 25 bytes. Anything beyond the point is ignored.

The default is a string giving the time and date when the assembler loaded.

A program can use multiple DATE directives, but the assembler ignores all but the last.

5.15.10 TITLE Directive

■ Syntax ■

TITLE ”*character_string*”

■ Description ■

This directive specifies a text string for the print file’s title field, part of the header that appears on each page.

The limit is 70 bytes. Anything beyond that point is ignored.

The default text string is empty.

A program can use multiple TITLE directives, but the assembler ignores all but the last.

5.15.11 TAB Directive

■ Syntax ■

TAB [*tab_width*]

■ Corresponding Assembler Options ■

/T[*tab_width*]

■ Description ■

This directive specifies the tab width for aligning tabbed columns in the assembly listing output by adding space characters. The print file can therefore be printed on printers that do not support tab codes.

The operand is a constant expression with a value between 1 and 15. The default width is 8. Specifying a value outside the valid range produces this default.

A program can use multiple TAB directives, but the assembler ignores all but the first.

The corresponding command line option takes precedence over this directive.

5.16 Data Access Control Directive

The data access control directive controls usable data memory spaces. It is limited to only physical segment #0.

5.16.1 NOFAR Directive

■ Syntax ■

NOFAR

■ Description ■

When a NOFAR directive is specified, RASU8 limits data memory spaces to only one for the assembly file. In other words, the data memory space to be output to the object file is limited to only physical segment #0.

RASU8 does not allow the assembly file in which a NOFAR directive is described to access FAR data. If FAR data access is detected, RASU8 outputs an error.

6 RASU8 Relocatable Assembler

6.1 Overview

RASU8 is a relocatable assembler for developing assembly language programs for microcontrollers based on the nX-U8 8-bit RISC processor core. This Chapter describes assembler operating procedures and function.

The assembler relies on DCL files to provide specific details on the target microcontroller. Changing DCL files thus allows it to support a variety of microcontrollers.

Source code files are program files written in nX-U8 assembly language.

Assembler output consists of the following four files.

1. Object file
2. Print file
3. Error file
4. EXTRN declaration file

The object file contains relocatable object code plus information for linking and debugging.

The print file contains lists the contents of the source code file and the corresponding object code. There is also an option for listing the symbols used in the source code file.

The error file contains error messages together with the source code lines triggering them. In the absence of a file specification, the assembler sends this output to the screen.

The EXTRN declaration file provides EXTRN declarations for all public symbols defined in the program.

6.2 File Specification Defaults

The assembler uses, in such contexts as the command line and directive operands, file specifications for the following input and output files.

1. Source code files
2. Include files
3. Option definition files
4. ABL files
5. Object files
6. Print files
7. Error files
8. External symbol declaration files
9. DCL files

The drive and directory portions of these file specifications are optional. The base name portion is also optional for all files except source code files, include files, option definition files, and DCL files. The following lists the defaults for the drive, directory, base name, and file extension portions.

<table>
<tr><th>File specification</th><th>Drive</th><th>Directory</th><th>Base name</th><th>File extension</th></tr>
<tr><td>Source code file</td><td rowspan="2">Current drive</td><td rowspan="2">Current directory on drive</td><td rowspan="4">Must be present</td><td>.ASM</td></tr>
<tr><td>Option definition file</td><td rowspan="2">Must be present</td></tr>
<tr><td>Include file</td><td rowspan="2">Current drive (Note 1)</td><td rowspan="2">Current directory on drive (Note 1)</td></tr>
<tr><td>DCL file</td><td>.DCL (fixed)</td></tr>
<tr><td>ABL file</td><td rowspan="5">Current drive (Note 2)</td><td rowspan="5">Current directory on drive (Note 2)</td><td rowspan="5">Base name from source code file specification</td><td>.ABL</td></tr>
<tr><td>Object file</td><td>.OBJ</td></tr>
<tr><td>Print file</td><td>.PRN</td></tr>
<tr><td>Error file</td><td>.ERR</td></tr>
<tr><td>EXTRN declaration file</td><td>.EXT</td></tr>
</table>

(Note 1) File searching applies when there is no drive portion and the directory portion is either missing or does not start with a backslash (\). For further details on the search paths for include files and DCL files, see Sections 6.5.2.6 “/I” and 5.1.1 “TYPE Directive,” respectively.

(Note 2) If all four portions are left unspecified, the file specification defaults to the drive, directory, and base name of the source code file specification. Only the file extension changes.

6.3 Running the Assembler

This Section describes the procedure for running the assembler.

At the MS-DOS prompt, type RASU8 followed by the name of the source code file and any options and then hit the Return/Enter key.

The assembler has the following command line syntax.

RASU8 [*options*] *source_file* [*options*]

The field *source_file* specifies the source code file to assemble; options, a list of options or option definition files. Options must start with a switch character—slash (/) or hyphen (-). Separate options from the source code file name and each other with at least one whitespace character.

Omitting the field *source_file* displays a brief help screen listing RASU8 command line syntax and options. The assembler immediately returns to the MS-DOS prompt.

■ Example ■

To assemble the source code file MAIN.ASM with the /S option, enter the following command line.

```
RASU8  MAIN /S
```

or

```
RASU8 /S MAIN
```

In the absence of an overt file extension for the source code file name, the assembler adds .ASM. Similarly, the drive and directory default to the current drive and directory, respectively.

Entering a valid command line displays the assembler's sign-on message on the screen followed by the following series of progress messages.

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

Adding the /G option expands this list by one message.

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

The assembler starts by reading in a DCL file and displaying a "loading..." message giving its name.

The assembler makes two passes through the source code file. The first pass tries to determine symbol values and program addresses. The second pass then uses these results in generating the object file. The assembler displays progress messages at the start of each path. If the /G option is specified, these two

progress messages are separated by a third indicating branch instruction optimization.

Errors in the program trigger error messages. For further details on error messages, see Section 6.9 “Error Messages.”

When assembly is complete, the assembler displays the following summary and returns to the MS-DOS prompt.

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

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

The first three lines list the output files created: the print file, the object file, and the error file. Note that the last defaults to “Console,” which is the assembler’s name for the screen.

Following these file names are assembly statistics: total number of error messages, total number of warning messages, and total number of lines read from the source code and include files. The Warnings line also lists, with /W, the warning types checked by the assembler.

■ Reference ■

The assembler sends all its screen display messages to the standard output device. The programmer can therefore save these to a file with standard MS-DOS redirection.

Alternatively, there are also the /E option and the ERR directive for redirecting only the error and warning messages to a disk file.

6.4 Option Definition Files

In addition to reading the source code file and options from the command line, the assembler can also read them from text files called option definition files.

6.4.1 Specifying Option Definition File

To specify an option definition file on the command line, place an at mark (@) immediately before its name. Do not insert any whitespace characters between the two.

■ Example 1 ■

To assemble the source code file MAIN.ASM with the options specified in the option definition file

FOO.OPT, enter the following command line.

```
RASU8  MAIN.ASM  @FOO.OPT
```

■ Example 2 ■

To assemble MAIN.ASM with both the source code file and the options specified in the option definition file BAR.OPT and add the /S option, enter the following command line.

```
RASU8  @BAR.OPT  /S
```

6.4.2 Option Definition File Syntax

An option definition file contains the following elements.

1. Source code file specification
2. Options
3. Comments

These elements are separated with spaces (20H), tabs (09H), and line feeds (0AH). The assembler skips carriage returns (0DH).

There are no limits on the number of options or characters per line.

Comments start with a semicolon (;), sharp (#), or // combination. The assembler skips that and all characters from there to the line feed (0AH). Block comments are not supported.

■ Example ■

The following is an example of an option definition file for assembling MAIN.ASM with the options /E, /R, /NL and /Xextrn.ext.

```
;----------------------------------------
; Sample option definition file (BAR.OPT)
;----------------------------------------
  MAIN.ASM       ; Source code file specification
  /E             ; Enable error file output (with default name)
  /R /NL         ; Modify print file output items
  /Xextrn.ext    ; Generate external symbol declaration file with name
                 ; extrn.ext
```

6.5 Options

Options control assembler operation, output file format, and the like. All options start with a switch character, followed by the option name. Some options then take parameters.

The switch character can be either a slash (/) or a hyphen (-). This manual uses the slash (/) for convenience of explanation.

Case does not matter in option names. They can use upper or lower case.

Do not insert any whitespace characters between the option name and the preceding switch character or any following parameters.

Many options have corresponding directives with exactly the same functionality.

6.5.1 List of Available Options

The following lists the options provided by the assembler.

An asterisk (*) in the Default column indicates that the option is the default when neither it nor its corresponding directive appears; a number, the default for a numerical parameter.

Option	Default	Corresponding directive	
/MS	*	MODEL SMALL	Use SMALL memory model
/ML		MODEL LARGE	Use LARGE memory model
/DN	*	MODEL NEAR	Use NEAR data model
/DF		MODEL FAR	Use FAR data model
/CD	*		Distinguish between upper and lower case in user and SFR symbols
/NCD			Do not distinguish between upper and lower case in user and SFR symbols
/SL[*symbol_length*]	32		Extend number of significant characters recognized by assembler in user symbols
/W[*warning_type*]	*		Enable checking for specified warning types
/NW[*warning_type*]			Disable checking for specified warning types
/I*include_path*			Add directory to include file search path
/DEF*symbol*[=*body*]		DEFINE	Define macro symbol
/KE or /KEUC			Parse source code file using EUC double-byte character encoding
/G			Enable forward reference optimization for GJMP directives and use of GBcond directives

Option	Default	Corresponding directive	
/PR[*print_file*]	*	PRN	Generate print file with specified name
/NPR		NOPRN	Suppress print file
/A[*abl_file*]			Generate absolute print file
/L	*	LIST	Generate assembly listing
/NL		NOLIST	Suppress assembly listing
/S		SYM	Generate symbol table
/NS	*	NOSYM	Suppress symbol table
/R		REF	Generate cross-reference listing
/NR	*	NOREF	Suppress cross-reference listing
/PW*page_width*	79	PAGE , *page_width*	Specify number of bytes per line in print file
/NPW			Remove line length limit for print file
/PL*page_length*	60	PAGE *page_length*	Specify number of lines per page in print file
/NPL			Remove page length limit for print file
/T[*tab_width*]	8	TAB [*tab_width*]	Specify tab width
/O[*object_file*]	*	OBJ[(*object_file*)]	Generate object file with specified name
/NO			Suppress object file output
/SD			Include C source level debugging information in object file
/D			Include assembly level debugging information in object file

Option	Default	Corresponding directive	
/ND	*		Suppress assembly level debugging information object
/E[*error_file*]		ERR[(*error_file*)]	Send error messages to specified file
/NE	*	NOERR	Send error messages to screen
/X[*extrn_file*]			Generate external symbol declaration file
/BRAM(*start_address*,*end_address*)			Add external RAM region
/BROM(*start_address*,*end_address*)			Add external ROM region
/BNVRAM(*start_address*,*end_address*)			Add external nonvolatile memory region
/BNVRAMP(*start_address*,*end_address*)			Add external nonvolatile memory region to physical segment address #0 in code memory space
/ZC			Check whether a program code is placed in an inaccessible range with DB and DW directives

6.5.2 Option Descriptions

6.5.2.1 /MS and /ML

■ Syntax ■

/MS

/ML

■ Description ■

These command line options specify the memory model for the application program: SMALL (/MS) or LARGE (/ML).

The memory model defaults to SMALL.

For further details on memory models, see Section 2.6 “Memory Models.”

■ Corresponding Directive ■

The MODEL directive provides an alternate means to specify the memory model. These command line options take precedence over the directive, however.

For further details on the MODEL directive, see Section 5.1.2 “MODEL Directive.”

■ Example ■

To assemble the source code file FOO.ASM with the LARGE memory model, enter the following command line.

```
RASU8   FOO.ASM   /ML
```

■ Aside ■

The programmer cannot specify both /ML and /MS.

6.5.2.2 /DN and /DF

■ Syntax ■

/DN

/DF

■ Description ■

These command line options specify the data model for the application program: NEAR (/DN) or FAR (/DF).

The data model defaults to NEAR.

For further details on data models, see Section 2.7 “Data Models.”

■ Corresponding Directive ■

The MODEL directive provides an alternate means to specify the data model. These command line options take precedence over the directive, however.

For further details on the MODEL directive, see Section 5.1.2 “MODEL Directive.”

■ Example ■

To assemble the source code file FOO.ASM with the FAR data model, enter the following command line.

```
RASU8   FOO.ASM   /DF
```

■ Aside ■

The programmer cannot specify both /DN and /DF.

6.5.2.3 /CD and /NCD

■ Syntax ■

/CD

/NCD

■ Description ■

These command line options control whether the assembler distinguishes between upper and lower case in the letters making up user and SFR symbols. Specifying the /CD option, the default, distinguishes between upper and lower case—that is, two symbols are the same only if they agree in both spelling and case. Specifying the /NCD option symbol, on the other hand, forces all letters to upper case, so that only

the spelling need match, and saves these symbols in upper case in the print and object files.

Note that this case control applies only to user symbols defined as labels, segment names, etc. in the program and to SFR symbols defined in DCL files. Instructions, directives, and other reserved words are always case insensitive.

■ Example ■

To assemble the source code file FOO.ASM without distinguishing between upper and lower case, enter the following command line.

```
RASU8   FOO.ASM /NCD
```

Only with the /NCD option does the following source code from the source code file FOO.ASM assemble without producing a “symbol undefined” error message.

```
        CSEG
        L       R0, UCSYM
        SB
        DSEG    AT   0:200H
UcSym:
        DS      10H
```

This is because, although the symbols UcSym and UCSYM represent different combinations of upper and lower case letters, the assembler checks only the spelling and treats them as the same symbol. Assembling without the /NCD option produces an “undefined symbol” error message.

■ Aside ■

The programmer cannot specify both /CD and /NCD.

6.5.2.4 /SL

■ Syntax ■

/SL[*symbol_length*]

■ Description ■

From RASU8 V1.60, the default number of characters recognized as a symbol in a source file has been extended to 255 characters from 32 characters.

The number of characters specified with /SL option is ignored by this modification, and RASU8 always recognizes 255 characters.

6.5.2.5 /W and /NW

■ **Syntax** ■

/W[*warning_type*]

/NW[*warning_type*]

■ **Description** ■

These command line options respectively enable (/W) and disable (/NW) checking for the specified warning types.

The operand lists the warning types using letters from the following list.

Letter	Warning Type
R	Check relocatable segment definitions
P	Check directives
E	Check expression syntax
A	Check addressing syntax
S	Check SFR access attributes
T	Check ROMWINDOW region specifications

Leaving the operand blank specifies all warning types, so that specifying /W or /NW alone enables or disables all checks, respectively.

The default is to enable all checks.

For further details on specific warning messages and their warning types, see Section 6.9.2.3 “Warning Messages.”

■ **Example** ■

To assemble the source code file FOO.ASM without checking directive or expression syntax, enter the following command line.

```
RASU8   FOO.ASM   /NWPE
```

6.5.2.6 /I

■ **Syntax** ■

/I*include_path*

■ **Description** ■

This command line option adds a directory to the search path for files specified with INCLUDE directives. Use multiple /I options to specify multiple directories.

The assembler searches directories for the include file in the following order.

(1) current directory

(2) directories specified with /I options, in the order that the /I options are specified

For further details on the INCLUDE directive, see Section 5.10.1 “INCLUDE Directive.”

■ **Example** ■

To assemble the source code file FOO.ASM searching for include files in the order current directory, C:\USR\SHARE\INC, and C:\USR\PRV\INC, enter the following command line.

```
RASU8   FOO.ASM   /IC:\USR\SHARE\INC   /IC:\USR\PRV\INC
```

6.5.2.7 /DEF

■ **Syntax** ■

/DEF*symbol*[=*body*]

■ **Description** ■

This command line option defines a macro symbol.

Do not insert any whitespace characters on either side of the equal sign (=) separating the *symbol* and *body* fields.

Omitting the optional =*body* portion assigns “1” to the macro body.

■ **Corresponding Directive** ■

The DEFINE directive provides an alternate means to define a macro symbol.

■ **Example** ■

To assemble the source code file FOO.ASM with “TYPE(M610001)” and “1” in the macro bodies for macro symbols READDCL and ONE, respectively, enter the following command line.

```
RASU8   FOO.ASM   /DEFREADDCL=TYPE(M610001)   /DEFONE
```

6.5.2.8 /KE and /KEUC

■ Syntax ■

/KE

/KEUC

■ Description ■

Adding either option to the command line switches the assembler from the default Shift JIS (SJIS) encoding for double-byte characters to the EUC encoding. Both encodings assign 2-byte codes to Japanese “full width” characters, but use different sets of code points. The following is the assembler’s “good enough for government work” definition of valid byte combinations.

	Shift JIS (SJIS) encoding	**EUC code encoding (/KE or /KEUC specified)**
First byte of double-byte character	81H to 9FH 0E0H to 0FCH	0B0H to 0F4H 0A1H to 0A8H
Second byte of double-byte character	40H to 0FCH	0A0H to 0FEH

6.5.2.9 /G

■ Syntax ■

/G

■ Description ■

This command line option enables forward reference optimization for GJMP directives and the use of GB*cond* directives.

If the /G option is not specified, the assembler only checks whether the branch target address specified as a GJMP directive operand is within range of a relative branch from the current location for symbols with backward references. Otherwise, it always substitutes a B instruction.

Specifying the /G option switches to an optimization algorithm that does not depend on the reference direction. The assembler checks the distance to any branch target address that is in the same contiguous logical segment—that is, both addresses are either in absolute segments with the same segment type or in the same relocatable segment with no intervening CSEG or ORG directives changing the current location counter for that segment with AT operands.

■ **Example 1** ■

The GJMP directives in the following example are all in the same contiguous logical segments as their branch target addresses. The ORG directive between GJMP CLAB3 and the label CLAB3 does not interfere because it changes the location counter for the DATA segment, not the CODE one.

```
        TYPE(M610001)
        CSEG AT 0:1000H
        MODEL LARGE
CLAB1:
        .
        .
        .
        GJMP CLAB1

COD_SEG       SEGMENT CODE
        RSEG COD_SEG
        GJMP CLAB2
        GJMP CLAB3
CLAB2:

        DEG AT 8000H
        ORG 100H
        .
        .
        .

        RSEG COD_SEG
CLAB3:
        .
        .
        .
```

■ **Example 2** ■

The GJMP directives in the next example are not within the same contiguous logical segments as their branch target addresses, so the assembler replaces them all with B instructions.

```
CLAB1    CODE 0:1060H

       CSEG AT 0:1000H
CLAB2:
       GJMP CLAB1

       ORG 1020H
       GJMP CLAB2
       GJMP CLAB3

       CSEG AT 0:1040H
CLAB3:

COD_SEG1 SEGMENT CODE
       RSEG COD_SEG1
       GJMP CLAB4
       GJMP CLAB1

COD_SEG2 SEGMENT CODE
       RSEG    COD_SEG2
CLAB4:
       .
       .
       .
```

6.5.2.10 /PR and /NPR

■ **Syntax** ■

/PR[*print_file*]

/NPR

■ **Description** ■

These command line options respectively enable (/PR) and disable (/NPR) print file output.

The operand specifies a name for the print file. For further details on defaults when this field is fully or partially omitted, see Section 6.2 “File Specification Defaults.”

Note, however, that the /A option overrides the /NPR option to always generate a print file.

The default (/PR) is to generate a print file. The default name for the output file is the source code file name with the file extension changed to .PRN.

■ Corresponding Directives ■

The PRN and NOPRN directives provide alternate means to control print file output. The corresponding command line options take precedence over these directives. For further details on these directives, see Section 5.15.2 “PRN and NOPRN Directives.”

■ Example ■

```
RASU8  FOO.ASM /PROUTPUT.LST
```

This example generates the print file OUTPUT.LST.

```
RASU8  FOO.ASM /NPR
```

This example suppresses print file output.

■ Aside ■

The programmer cannot specify both /PR and /NPR.

6.5.2.11 /A

■ Syntax ■

/A[*abl_file*]

■ Description ■

This command line option specifies the ABL file to use in generating an absolute print (.APR) file.

An absolute print file is a print file with all machine code, addresses, and other indeterminate portions fully resolved.

An ABL file is a binary file, created by the linker, containing information necessary for generating absolute print files.

Adding the /A option does not change the procedure for specifying the file name for the /PR option—except for changing the default file extension to .APR from the .PRN used for normal print files.

For further details on absolute print files, see Chapter 11 “Absolute Print File Generation.”

■ Example ■

To generate an absolute print file for the source code file FOO.ASM, enter the following command line,

where APRINFO.ABL is the corresponding ABL file.

```
RASU8   FOO.ASM /AAPRINFO
```

6.5.2.12 /L and /NL

■ Syntax ■

/L

/NL

■ Description ■

These command line options respectively enable (/L) and disable (/NL) assembly listing output to the print file at the start of the source code file.

Specifying the /L option lists all source statements up until the first NOLIST directive in the program. The /NL option, in contrast, skips all source statements up until the first LIST directive in the program. From that point onward, the LIST and NOLIST directives toggle output on and off, respectively. Note, however, that source code lines triggering error or warning messages are always sent to the assembly listing. The default is to enable (/L) output.

For further details on the assembly listing output, see Section 6.7 “Print Files” below.

■ Corresponding Directives ■

The LIST and NOLIST directives provide alternate means to control assembly listing output. The corresponding command line options take precedence over these directives. For further details on these directives, see Section 5.15.5 “LIST and NOLIST Directives.”

■ Example ■

To assemble the source code file FOO.ASM and include its contents in the assembly listing, enter the following command line.

```
RASU8   FOO.ASM /L
```

To skip the assembly source code output, enter the following command line.

```
RASU8   FOO.ASM /NL
```

■ Aside ■

The programmer cannot specify both /L and /NL.

6.5.2.13 /S and /NS

■ Syntax ■

/S

/NS

■ Description ■

These command line options respectively enable (/S) and disable (/NS) symbol table output to the print file.

The default is to disable (/NS) output.

For further details on the symbol table output, see Section 6.7 “Print Files” below.

■ Corresponding Directives ■

The SYM and NOSYM directives provide alternate means to control symbol table output. The corresponding command line options take precedence over these directives. For further details on these directives, see Section 5.15.6 “SYM and NOSYM Directives.”

■ Example ■

To assemble the source code file FOO.ASM and include the symbol table in the print file, enter the following command line.

```
RASU8  FOO.ASM /S
```

To skip the symbol table, enter the following command line.

```
RASU8  FOO.ASM /NS
```

■ Aside ■

The programmer cannot specify both /S and /NS.

6.5.2.14 /R and /NR

■ Syntax ■

/R

/NR

■ Description ■

These command line options respectively enable (/R) and disable (/NR) cross-reference listing output to the print file at the start of the source code file. Specifying the /R option cross-references all source statements up until the first NOREF directive in the program. The /NR option, in contrast, skips cross-references for all source statements up until the first REF directive in the program. From that point onward, the REF and NOREF directives toggle output on and off, respectively. Note, however, that this switching function is seldom used within source code files because most programmers prefer cross-references at the file level.

The default is to disable (/NR) output.

For further details on the cross-reference listing output, see Section 6.7 “Print Files” below.

■ Corresponding Directives ■

The REF and NOREF directives provide alternate means to control cross-reference listing output. The corresponding command line options take precedence over these directives. For further details on these directives, see Section 5.15.7 “REF and NOREF Directives.”

■ Example ■

To assemble the source code file FOO.ASM and include the cross-reference listing in the print file, enter the following command line.

```
RASU8   FOO.ASM /R
```

To skip the cross-reference listing, enter the following command line.

```
RASU8   FOO.ASM /NR
```

■ Aside ■

The programmer cannot specify both /R and /NR.

6.5.2.15 /PW and /NPW

■ Syntax ■

/PW*page_width*

/NPW

■ Description ■

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

The operand is an integer constant specifying the line length in bytes. The valid range is 79 to 132. Specifying a value outside the valid range produces the nearest endpoint: 79 for a number less than 79, 132 for one greater than 132.

The /NPW option removes the line length limit for the print file, causing the assembler to never insert line feeds even when a line extends beyond 132 bytes.

The default width is 79 columns.

■ Corresponding Directives ■

The PAGE directive provides an alternate means to specify the line length, as its second operand.

The NOPAGE directive provides an alternate means to remove the line length, but be aware that it simultaneously disables automatic page breaks. The directive provides no way to retain automatic page breaks while disabling line wrapping.

These command line options take precedence over the directive, however.

■ Example ■

```
RASU8  FOO.ASM /PW132
```

This example wraps print file lines at 132 bytes.

```
RASU8  FOO.ASM /NPW
```

This example disables line wrapping.

■ Aside ■

The programmer cannot specify both /PW and /NPW.

6.5.2.16 /PL and /NPL

■ Syntax ■

/PL*page_length*

/NPL

■ Description ■

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

The operand is an integer constant specifying the page length in lines. The valid range is 10 to 65535. Specifying a value outside the valid range produces the nearest endpoint: 10 for a number less than 10, 65535 for one greater than 65535.

The /NPL option removes the page length limit for print file, causing the assembler to never insert page headers and footers even when a page extends beyond 65535 lines. Note, however, that this option does not suppress the forced page breaks manually inserted with PAGE directives or the automatic page break at the end of the output.

The default length is 60 lines.

■ Corresponding Directives ■

The PAGE directive provides an alternate means to specify the page length, as its first operand.

The NOPAGE directive provides an alternate means to remove the page length, but be aware that it simultaneously disables line wrapping. The directive provides no way to retain automatic page breaks while disabling line wrapping.

These command page options take precedence over the directives, however.

■ Example ■

```
RASU8  FOO.ASM /PL100
```

This example inserts page breaks in the print file after 100 lines.

```
RASU8  FOO.ASM /NPL
```

This example does not limit the page length.

■ Aside ■

The programmer cannot specify both /PL and /NPL.

6.5.2.17 /T

■ Syntax ■

/T[*tab_width*]

■ Description ■

This command line option specifies the tab width for aligning tabbed columns in the assembly listing output by adding space characters. The print file can therefore be printed on printers that do not support tab codes.

The operand is a constant expression with a value between 1 and 15. The default width is 8. Specifying a value outside the valid range produces this default.

■ Corresponding Directive ■

The TAB directive provides an alternate means to specify the tab width. This command line option takes precedence over the directive, however.

If neither the /T option nor the TAB directive is specified, the assembler sends tab codes to the print file unaltered.

■ Example ■

```
RASU8  FOO.ASM /T4
```

This example replaces tab codes with up to four spaces to align the print file columns.

6.5.2.18 /O and /NO

■ Syntax ■

/O[*object_file*]

/NO

■ Description ■

These command line options respectively enable (/O) and disable (/NO) object file output.

The operand specifies a name for the object file. For further details on defaults when this field is fully or partially omitted, see Section 6.2 “File Specification Defaults.”

The default is to generate (/O) an object file. The default name for the output file is the source code file name with the file extension changed to .OBJ.

■ Corresponding Directives ■

The OBJ and NOOBJ directives provide alternate means to control object file output. The corresponding command line options take precedence over these directives. For further details on these directives, see Section 5.15.1 “OBJ and NOOBJ Directives.”

■ Example ■

```
RASU8  FOO.ASM /OOUTPUT.OBJ
```

This example assembles the source code file FOO.ASM to produce the object file OUTPUT.OBJ.

```
RASU8  FOO.ASM /NO
```

This example suppresses object file output.

■ Aside ■

The programmer cannot specify both /O and /NO.

6.5.2.19 /SD

■ Syntax ■

/SD

■ Description ■

This command line option, for use only with CCU8 assembly language source code output, processes C source level debugging directives and includes the resulting information in the object file output. Without it, C source level debugging is impossible.

■ Example ■

```
RASU8  CCFOO /SD
```

This example assembles the CCU8 assembly language source code output file CCFOO.ASM into an object file containing C source level debugging information.

6.5.2.20 /D and /ND

■ Syntax ■

/D

/ND

■ Description ■

These command line options respectively enable (/D) and disable (/ND) assembly level debugging information output to the object file for use in symbolic debugging of the program.

The default is not to include this debugging information in the file.

■ Corresponding Directives ■

The DEBUG and NODEBUG directives provide alternate means to control assembly level debugging information output. The corresponding command line options take precedence over these directives. For further details on these directives, see Section 5.15.4 “DEBUG and NODEBUG Directives.”

■ Example ■

```
RASU8  FOO.ASM /D
```

This example includes assembly level debugging information in the object file.

■ **Aside** ■

The programmer cannot specify both /D and /ND.

6.5.2.21 /E and /NE

■ **Syntax** ■

/E[*error_file*]

/NE

■ **Description** ■

These command line options respectively enable (/E) and disable (/NE) error message output to a file.

The operand specifies a name for the error message file. For further details on defaults when this field is fully or partially omitted, see Section 6.2 “File Specification Defaults.”

The default is to not generate (/NE) this file, but send all error messages to the standard error output.

Note that this directive controls routing for error and warning messages only. To reroute fatal error messages and internal processing error messages to a file, use standard MS-DOS redirection.

■ **Corresponding Directives** ■

The ERR and NOERR directives provide alternate means to control error message output. The corresponding command line options take precedence over these directives. For further details on these directives, see Section 5.15.3 “ERR and NOERR Directives.”

■ **Example** ■

```
RASU8   FOO.ASM /EERROR.LST
```

This example generates the error file ERROR.LST.

■ **Aside** ■

The programmer cannot specify both /E and /NE.

6.5.2.22 /X

■ Syntax ■

/X[*extrn_file*]

■ Description ■

This command line option generates an EXTRN declaration file.

The operand specifies a name for the EXTRN declaration file. For further details on defaults when this field is fully or partially omitted, see Section 6.2 “File Specification Defaults.”

The default is to not generate this file.

For further details on EXTRN declaration files, see Section 6.8 “EXTRN Declaration Files.”

■ Example ■

```
RASU8  FOO.ASM /XEXTRN.INC
```

This example generates the EXTRN declaration file EXTRN.INC.

6.5.2.23 /BXXX (/BRAM, /BROM, /BNVRAM, and /BNVRAMP)

■ Syntax ■

/BRAM(*start_address*,*end_address*)

/BROM(*start_address*,*end_address*)

/BNVRAM(*start_address*,*end_address*)

/BNVRAMP(*start_address*,*end_address*)

■ Description ■

These command line options specify external memory regions to the assembler. Note that the programmer cannot redefine memory types built into the microcontroller.

The operands specify the first and last addresses for the region. The /BRAM command line option adds an external RAM region. The address range can be any region between 0:0000H and 255:0FFFFH. Any portion in physical segment address #0, however, is assigned to the data memory space.

The /BROM command line option adds a ROM region. The address range can be any region between 0:0000H and 255:0FFFFH. Any portion in physical segment address #0, however, is assigned to code memory space.

The /BNVRAM command line option adds an external nonvolatile memory region. The address range can be any region between 0:0000H and 255:0FFFFH. Any portion in physical segment address #0,

however, is assigned to the data memory space.

The /BNVRAMP command line option adds a nonvolatile memory region to physical segment address #0 in the code memory space. The address range can be any region between 0:0000H and 0:0FFFFH.

6.5.2.24 /ZC

■ Syntax ■

/ZC

■ Example ■

When the /ZC option is used, a warning is output if a program code is placed in an inaccessible range with DB and DW directives or there is such a possibility.

This option can partially specify to enable or disable a warning check with CHKDBDW and NOCHKDBDW directives.

6.6 Return Codes

At the end of assembly, the assembler exits with one of the following return codes indicating the status to the batch file or other caller.

Return Code	Description
0	Success
1	The assembler produced at least one warning message.
2	The assembler produced at least one error message.
3	The assembler aborted with a fatal error, internal processing error, or DCL error.

6.7 Print Files

This Section describes the syntax for reading assembler print files.

These files consist of the following parts.

1. Assembly listing.
   This portion lists the program source code and the corresponding object code.

2. Cross-reference list.
   This portion lists the line numbers where each user symbol is defined or referenced in the program.

3. Symbol table.
   This portion describes the user symbols used in the program.

4. Final summary.
   This portion gives totals for error and warning messages as well as address space information.

The following options and directives control output to the print file.

	Directive	Option
Print file in general	PRN, NOPRN	/PR, /NPR
Assembly listing	LIST, NOLIST	/L, /NL
Cross-reference listing	REF, NOREF	/R, /NR
Symbol table	SYM, NOSYM	/S, /NS

6.7.1 Assembly Listing

The following is an example of an assembly listing.

The following discussion refers to the numbers in parentheses down the left edge. Note that these have been added for use in this discussion. They do not appear in the actual assembly listing file itself.

```
(1) RASU8(ML610001)Relocatable Assembler, Ver.1.00.11   assemble list. page:   1
(2)  Source File: SAMPLE.asm
(3)  Object File: SAMPLE.obj
(4)  Date  : 2000/05/13 Thu.[16:53]
(5)  Title :
(6) ## Loc. Object                  Line   Source Statements

                                      1   ;*********************************
                                      2   ; sample program
                                      3   ;*********************************
                                      4           TYPE (M610001)
                                      5           MODEL SMALL, NEAR
                                      6           ROMWINDOW  0, 7FFFH
                                      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
(8) ------  END OF INCLUDE  --------
(9)   --------------------------     17           CSEG    AT 2000H
                                     18   ;---------------------------------
  ┌ 00:2000                          19   LABEL:
  │   --------------------------     20           DSEG
(10)00:8000                          21           DS      100H
  │   --------------------------     22           BSEG
  │ 00:40000 (8000.0)                23           DBIT    200H
  └ 00:41000 (8200.0)                24           ORG     8200H.0
                                     25   ;---------------------------------
      --------------------------     26           CSEG
  ┌ 00:2000 12-90 00-30              27           L       ER0, 3000H
(11)00:2004 FF 02                    28           MOV     R2, #0FFH
  └ 00:2006 00'03                    29           MOV     R3, #EXT_NUM
                                     30   ;---------------------------------
  ┌     = 0000FFFFH                  31   NUM_SYM         EQU     0FFFFH
(12)    = 00:FFFFH                   32   DATA_SYM        DATA    0FFFFH
  └     = 00:47FFFH (8FFFH.7)        33   BIT_SYM         BIT     08FFFH.7
                                     34   ;---------------------------------
      --------------------------     35           CSEG
  ┌ 00:2008 01 02 03 04 05 06 07 08  36           DB      1,2,3,4,5,6,7,8,
(13)00:2010 09 0A                   >>>   9,10
  │ 00:2012 01-00 02-00 03-00 04-00  37           DW      1,2,3,4,
  └ 00:201A 05-00 06-00 07-00       >>>   5,6,7
                                     38   ;---------------------------------
                                     39           L       R0, R0
  ┌  ** Error 00: bad operand
  │ 00:2020 11-90 FF-7F              40           ST      R0, BIT_SYM
(14) ** Error 29: out of range
  │  ** Error 29: out of range
  └  ** Warning 12: usage type mismatch
                                     41
```

(1) Each page starts with the microcontroller model number, assembler version number, and page

number.

(2) This line gives the source code file name.

(3) This line gives the object file name.

(4) This line gives the date and time of assembly.

(2), (3), and (4) appear only on the first page.

(5) This line gives the text specified by the TITLE directive. The default text string is empty.

(6) This line gives the column headings for the assembly listing. These headings have the following meanings.

Heading	Description
##	This column gives the physical segment address in hexadecimal. If the assembler cannot finalize the physical segment number, "??" appears instead. A colon (:) joins this number to the next (Loc).
Loc.	This column gives the location counter in hexadecimal. This number represents an absolute address for an absolute segment and an offset from the segment starting address for a relocatable one. For bit type segments, this column gives the bit address and, in parentheses, the dot operator equivalent. 01000 (0200.0)
Object	This column gives the object code in bytes in hexadecimal. If the assembler cannot finalize a field, it appends a single quotation mark (’).
Line	This column gives a line number in decimal. Note that this column counts include file contents as well, so does not necessarily match the line number in the source code file.
Source Statements	This column gives the corresponding line from the source code file or current include file.

The assembler sometimes replaces the first three columns (##, Loc, and Object) with such things as a special message or an error message.

(7) This special message indicates the start of an include file inserted with the INCLUDE directive. From the next line, the Source column switches to lines from that file.

(8) This special message indicates the end of the include file.

(9) This divider indicates a change of segment with CSEG, DSEG, RSEG, or similar directive.

(10) These numbers give the location counter contents for statements with a label, DS directive, DBIT

directive, or ORG directive.

(11) These lines add object code. Hyphens (-) join the bytes in word-sized objects. If the assembler cannot finalize a field, it appends a single quotation mark (’).

(12) These numbers give the values (numerical or address) assigned to a symbol with local symbol definition directives (EQU, SET, CODE, DATA, NVDATA, TABLE, BIT, or NVBIT).

(13) The assembler divides the initialized data from DB and DW directives into 8-byte units and indicates continuation lines with the notation “>>>“ in the Line column.

(14) Error and warning messages appear immediately after the source code triggering them.

6.7.2 Cross-Reference Listing

A cross-reference listing gives the user symbols defined in the program and the line numbers where they are used.

The following is an example of a cross-reference listing.

```
RASU8(ML610001)Relocatable Assembler, Ver.1.00.11   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
```

The listing has two columns: the symbol name and the line numbers where it is used.

A sharp (#) indicates the line defining the symbol—or the last such line if the symbol is defined with SET directives. The notation “(SFR)” preceding the line numbers indicates an SFR symbol defined in the DCL file.

6.7.3 Symbol Table

The symbol table provides, in separate sections, detailed information on the user symbols and relocatable segments in the program.

6.7.3.1 Symbol Information

This portion of the symbol table provides detailed information on all symbols defined in the program and on SFR symbols referenced at least once.

The following is an example of such a listing.

```
----- 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
SEG021 ..... seg  BIT    ANY           0H.0    6
UNDEF ...... ***
```

The following describes each column individually.

The symbol column lists the symbols in ascending order.

The type column gives a type from the following list.

type	**Description**
`***`	Undefined symbol
`sfr`	SFR symbol
`loc`	Local symbol
`pub`	Public symbol
`seg`	Segment symbol
`com`	Communal symbol
`ext`	External symbol
`mcr`	Macro symbol. Note that the listing uses a different format, displaying the macro body instead.

The usgtyp column gives the usage type: NUMBER, NONE, CODE, DATA, NVDATA, TABLE, BIT, NVBIT, or TBIT.

The usgtyp column gives the usage type from the following list.

usgtyp	**Description**
`NUMBER`	Usage type NUMBER
`NONE`	Usage type NONE
`CODE`	Usage type CODE
`DATA`	Usage type DATA
`NVDATA`	Usage type NVDATA
`TABLE`	Usage type TABLE
`BIT`	Usage type BIT
`NVBIT`	Usage type NVBIT
`TBIT`	Usage type TBIT

The physeg column gives the physical segment attribute.

physeg	**Description**
`---`	N/A (usage type NUMBER)
`#`*`XX`*	Physical segment address *XX* (in hexadecimal)
`ANY`	Unknown to the assembler. The linker must decide.

The value column gives the symbol value in hexadecimal.

The ID column has three uses. For segment, communal, and external symbols, it gives the order in which the symbol was defined in its group. For simple relocatable symbols, it gives the ID for the segment to which the symbol belongs. Otherwise, it is blank.

6.7.3.2 Segment Information

This portion of the symbol table provides detailed information on all relocatable segments in the program.

The following is an example.

```
----- segment information -----

S-ID symbol       segtyp physeg  size  bound reltype
---- ------------ ------ ------ ------ ----- -------
   1 SEG000 ..... CODE   ANY        0H PAGE
   2 SEG001 ..... CODE   ANY        0H OCT
   3 SEG010 ..... CODE   #00        0H WORD
   4 SEG011 ..... CODE   #00       10H UNIT
   5 SEG021 ..... BIT    ANY      200H UNIT   NVRAM
```

The following describes each column individually.

The S-ID column numbers the segment symbols in the order in which they are defined; the symbol column lists them.

The segtyp column gives the segment type: CODE, DATA, NVDATA, TABLE, BIT, or NVBIT.

The segtyp column gives the segment type from the following list.

segtyp	**Description**
`CODE`	Segment type CODE
`DATA`	Segment type DATA
`NVDATA`	Segment type NVDATA
`TABLE`	Segment type TABLE
`BIT`	Segment type BIT
`NVBIT`	Segment type NVBIT

The physeg column gives the physical segment attribute.

The size column gives the segment size in hexadecimal. The unit for this size depends on the segment type: bytes for TABLE, DATA, and NVDATA and bits for BIT and NVBIT.

The bound column gives the boundary attribute from the following list.

bound	**Description**
UNIT	2-byte boundary for a CODE segment, 1-bit boundary for a BIT or NVBIT segment, 1-byte boundary for the rest
WORD	2-byte boundary
OCT	8-byte boundary
PAGE	256-byte boundary
Integer constant (1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, or 2048)	Specified boundary in bits for a BIT or NVBIT segment and in bytes for the rest

The reltyp column gives the special region attribute for assigning the relocatable segment, if specified: DYNAMIC or NVRAM. Otherwise, it is blank.

6.7.4 Final Summary

The final summary varies with the CPU core.

The following is an example.

■ Example ■

```
Target       : ML610001 (nX-U8/100)  ← (1)
Memory Model : LARGE                 ← (2)
Data   Model : FAR                   ← (3)
ROM WINDOW   :    0H to 7FFFH        ← (4)
Internal RAM : 8000H to 8FFFH        ← (5)

Errors   : 0                         ← (6)
Warnings : 0  (/Wrpeast)             ← (7)
Lines    : 62                        ← (8)
```

(1) This line gives the microcontroller model number and core name.

(2) This line gives the memory model.

(3) This line gives the data model.

(4) This line gives the memory range specified by the ROMWINDOW directive, the notation “None” if the NOROMWIN directive is specified, or the notation “not specified” if neither directive appears.

(5) This line gives the internal RAM range.

(6) This line gives the total number of error messages.

(7) This line gives the total number of warning messages.

(8) This line gives the number of lines read from the source code and include files.

6.8 EXTRN Declaration Files

This Section describes the procedures for generating and using RASU8 EXTRN declaration files.

6.8.1 What Are EXTRN Declaration Files?

An EXTRN declaration file lists EXTRN declarations for public symbols defined in the program. To generate it, add the /X option to the assembler command line.

Referencing a symbol between source code files requires both a public declaration in the source code file defining the symbol and an external declaration in the file referencing it. The normal procedure is for the programmer to manually insert both declarations, but the task of symbol management becomes increasingly complex as the number of symbols and files grows. EXTRN declaration files provide a means of resolving these problems.

6.8.2 Generating and Using EXTRN Declaration Files

Consider the following simple example in which source code module F001.ASM defines the subroutines SUB00 and SUB01 and the DATA address space symbols BUF00 and BUF01 for reference by the source code modules F002.ASM and F003.ASM.

F001.ASM

```
        TYPE (M610001)
        PUBLIC  SUB00 SUB01 BUF00 BUF01

        CSEG    AT 0:1000H
SUB00:
        ; sub routine SUB00
        RT

R_CODE  SEGMENT CODE
        RSEG    R_CODE
SUB01:
        ; sub routine SUB01
        RT

        DSEG    AT 0:8000H
BUF00:  DS      10H

R_DATA  SEGMENT DATA
        RSEG    R_DATA
BUF01:  DS      100H
```

Assembling this file with the /X command line option generates the following EXTRN declaration file F001.EXT.

```
RASU8 F001 /X
```

F001.EXT

```
;; External symbol declaration file.

       EXTRN  CODE NEAR     : SUB00
       EXTRN  CODE NEAR     : SUB01
       EXTRN  DATA NEAR     : BUF00
       EXTRN  DATA NEAR     : BUF01
;; End of listing.
```

Now all the programmer has to do to be able to reference the public symbols defined in F001.ASM is add the following line at the start of F002.ASM and F003.ASM.

```
       INCLUDE (F001.EXT)
```

Now assemble F001.ASM with the following command line.

```
RASU8 F001 /X /DF /ML
```

Note how the external symbol declarations change in the new F001.EXT.

```
;; External symbol declaration file.

       EXTRN  CODE FAR      : SUB00
       EXTRN  CODE FAR      : SUB01
       EXTRN  DATA NEAR     : BUF00
       EXTRN  DATA FAR      : BUF01
;; End of listing.
```

Using an EXTRN declaration file this way not only relieves the programmer of the external declaration management task, but also makes the program both easier to read and easier to maintain. Note also how the files in this example automatically add the physical segment attributes matching the memory and data models, so that the assembler can fully optimize addressing types for SUB00, SUB01, and BUF01.

6.9 Error Messages

The assembler reports errors during assembly in two ways.

1. Error messages to the screen or an error file
2. Error numbers in the print file

The assembler classifies errors into the following types.

1. Fatal errors
2. Assembly errors
3. Warnings
4. Internal processing errors

A fatal error is an I/O or other critical error so serious that the assembler cannot continue. The assembler therefore aborts at that point.

An assembly error is one arising during source code file processing. The assembler issues an error message, but continues anyway, generating the print and object files.

A warning alerts the programmer to a potential problem with the program. The assembler simply issues a warning message and continues, generating the print and object files.

An internal processing error represents a problem detected within the assembler itself. The assembler therefore aborts at that point.

The default is for the assembler to send these messages to the standard output device (screen). The programmer can therefore save them, together with other screen output, to a file with standard MS-DOS redirection. Alternatively, there are also the /E option and the ERR directive for redirecting the error and warning messages to the specified file.

6.9.1 Error Message Format

Error messages sent to the screen or error file have the following format.

■ Syntax ■

filename(*line1*) : *line2* : *type number* : *message*

The *filename* field gives the name of the file triggering the error; line1, the line number in the source code file; line2, the line number in the print file; *type*, the error type from the following list.

type	**Error Type**
Fatal Error	Fatal error
Error	Assembly error
Warning	Warning

The *number* and *message* fields represent the actual error message , an error code and text combination from Section 6.9.2 “Error Message List” below.

6.9.2 Error Message List

The following lists assembler error messages by error type. To the left of the text is an error code giving the error type and a unique number for the error message. Following the text is a description.

6.9.2.1 Fatal Error Messages

F00 insufficient memory

There is not enough memory to continue. The most likely cause is a lack of Windows virtual memory, so take the usual steps to lighten the load: deleting temporary and other unnecessary files to free up hard disk space, raising the upper bound on virtual memory to the entire disk space, closing any other applications that you may have open, etc.

Alternatively, try removing the /R command line option and any REF directives.

If this error message persists, try such measures as splitting the program into smaller modules and reducing the number of symbols.

F01 file not found : *file_name*

The assembler cannot find the specified source code file, include file, or DCL file.

F02 cannot open file : *file_name*

The assembler cannot create the specified object file, print file, or error file. Check for invalid characters or a nonexistent directory in the file name specification.

F03 cannot close file : *file_name*

The assembler cannot close the specified file. The most likely cause is insufficient disk space.

F04 error(s) found in DCL file

If there are syntax errors in a DCL file, the assembler cannot guarantee results, so aborts with an error message at the end of the file. This error should not arise with original DCL files for U8.

F05 file seek error

There was a file I/O error.

F06 too many INCLUDE nesting levels

Include files are nested more than eight levels deep.

F07 line number overflow

The line count for the program and all its include files exceeds 9,999,999 lines.

F08 I/O error writing file

There was a file I/O error writing to the object file.

F09 TYPE directive missing

Either there is no TYPE directive specifying the model number or is it preceded by another instruction or directive.

F10 unclosed block comment

There is no closing */ for an opening /*.

F11 illegal reading binary file

This message indicates an unusable ABL file. It is followed by a line in the following format giving further details.
ABL file : *message*
First check the following.

(1) Did the assembler originally issue error messages? Exclude warning messages.
(2) Have you edited the source code file since the original assembly?

(3) Have you changed specifications between assembler runs? The following must all match completely: memory model, data model, branch optimization setting, ROMWINDOW region specifications, external memory specifications (/BXXX options), case sensitivity (/CD and /NCD options), and include file search path (/I options).
(4) Did the linker issue any fatal error messages other than ones related to addressing types?
(5) Did you forget to specify the linker's /A option?
(6) Did you generate the ABL file by using the latest linker ?

Please check the above items, and if necessary, assemble your source file, and link the object files and generates the ABL file. Then regenerates the absolute print file by using assembler and the ABL file.
If this error message persists, contact your nearest LAPIS Semiconductor sales office.

The following lists the lines, prefixed with "ABL file:," that expand upon the "F11 error in binary file" fatal error message.

ABL file: module information is not found

The ABL file does not contain information for the current assembly language program.

ABL file: CORE ID mismatch

The CPU core specified in the ABL file and the current assembly language program do not match.

ABL file: Target Machine mismatch

The target microcontroller model numbers specified in the ABL file and the current assembly language program do not match.

ABL file: Memory Model mismatch

The memory models specified in the ABL file and the current assembly language program do not match.

ABL file: DATA Model mismatch

The data models specified in the ABL file and the current assembly language program do not match.

ABL file: branch optimization mismatch

The branch optimization settings specified in the ABL file and the current assembly language program do not match.

ABL file: symbol is not entry

The current assembly language program does not define a symbol included in the ABL file.

ABL file: illegal physical segment attribute

The physical segment attribute of a segment symbol has abnormal value.

ABL file: illegal segment type

The segment type of a segment symbol has abnormal value.

ABL file: illegal usage type

The symbol usage type has abnormal value.

ABL file: symbol type mismatch

The symbol type (segment, communal, etc.) specified in the ABL file and the current assembly language program do not match.

ABL file: symbol usage type mismatch

The symbol usage type (CODE, DATA, BIT, etc.) specified in the ABL file and the current assembly language program do not match.

ABL file: local symbol value mismatch: *symbol*

The specified label or other local symbol has an incorrect value.

ABL file: file format is illegal

There is a structural problem in the ABL file.

ABL file: absolute machine code mismatch (line *xxxx*)

The reassembled machine code does not match the fixed up machine code from the linker.

ABL file: location of absolute machine code mismatch (line *xxxx*)

The address of the reassembled machine code does not match that of the fixed up machine code from the linker.

F12 checksum error reading file

An ABL file record failed the checksum check.

F13 I/O error reading file

There was a file I/O error reading the ABL file.

F14 old DCL file

The DCL file is for a prior RASU8 version.

F16 source code filename not specified

No source code file was specified at start-up.

6.9.2.2 Assembly Error Messages

E00 bad operand

There is an error in an operand. For a microcontroller instruction, this message indicates an incorrect addressing specification or the wrong number of operands; for a directive, a syntax error.

E01 bad syntax

This message indicates a basic syntax mistake preceding instruction decoding.

E03 physical segment address out of range

The physical segment address exceeds the number of segments actually available. If the number is within the range of physical segments supported by the target microcontroller, however, the problem may be with a memory model specification of SMALL.

E04 bad character : *c* (*XX*)

The specified character *c* (with hexadecimal code *XX*) is not allowed in its current context.

E05 illegal integer constant

There is a syntax error in an integer or address constant.

E06 illegal escape sequence

There is a syntax error in an escape sequence for a character or string constant.

E07 unexpected EOL

E08 unexpected EOF

A character (‘*c*’) or string (“..”) constant is missing the closing delimiter.

E09 illegal string constant

There is a syntax error in a string constant.

E10 string constant too long

The length limit for string constants is 255 characters.

E11 illegal option : *option*

The assembler does not recognize the specified option, so ignores it.

E12 constant required

An instruction operand or command line option requires a missing integer constant.

E13 declaration duplicated

The same directive or option appears more than once.

E14 location out of range

The AT address in a CSEG or other segment start directive or the starting address in an ORG directive falls above or below the specified segment range. An instruction or directive (DS, DBIT, GJMP, GCAL, DB, or DW) has updated the location counter past the upper limit for the segment.

E17 AT address must be NUMBER

A segment start directive with both the # and AT portions must have only an offset for the latter. Specifying an expression of type address, as in the following example, produces this error message.

```
CSEG  AT  2:3000H #4
```

E18 segment/usage type mismatch

The specified segment or usage type does not match that required by the instruction or directive. Specific examples include the following.

- The usage type of the starting address in an ORG, CSEG, or other segment start directive does not match the segment type for the current segment.
- The usage type of the operand in a CODE or other symbol definition directive does not match the instruction type.
- A DS directive appears in a bit segment.
- A DBIT directive appears in a byte segment.
- A DB or DW directive appears in a segment other than CODE, NVDATA, or TABLE.

E19 undefined symbol : *symbol*

The specified *symbol* has not been defined.

E20 segment symbol required

The SIZE, OVL_ADDRESS, OVL_SEG, and OVL_OFFSET operators require a segment symbol as the right operand expression. The RSEG directive requires one as its only operand.

E21 forward reference not allowed

The line contains a forward reference in an operand. Many directives do not allow them. The RSEG directive also requires that the specified segment name be previously defined.

E22 stack segment not allowed

The current context does not accept the stack segment symbol $STACK.

E23 symbol redefinition : *symbol*

The specified *symbol* is already defined.

E25 segment ID mismatch

An ORG directive in a relocatable segment accepts a relocatable address expression in the operand only if that expression represents an address in the current segment.

E26 address not allowed

An ORG directive in a relocatable segment of type ANY requires an operand of usage type NUMBER.

E27 physical segment address mismatch

The physical segment addresses do not match. Either a ROMWINDOW directive specifies an address in physical segment #1 or higher as the operand or an ORG directive in a segment with a known physical segment number specifies a different number.

E28 local symbol required : *symbol*

The specified *symbol* in a public declaration must be defined as a local symbol.

E29 out of range : *message*

The operand value exceeds the range specified for the region appearing, with other information, in *message*.

E30 illegal boundary

E31 illegal relocation type

There is an incorrect boundary alignment or special region attribute specification in a SEGMENT or COMM directive.

E33 entry overflow

The number of segment, communal, or external symbols exceeds 65535.
Alternatively, there are too many external memory regions added with /BXXX (/BRAM, /BROM, /BNVRAM, and /BNVRAMP) options.

E34 string constant required

The DATE or TITLE directive operand must be a string constant.
Alternatively, there is a syntax error in a C source level debugging directive operand.

E35 absolute expression required

The operand must be a constant expression in such contexts as many directive operands, SWI instruction interrupt number, and shift instruction shift width.

E36 simple relocatable expression required

The operand in an EQU or other symbol definition directive or an ORG directive must be either a constant expression or a simple relocatable expression.

E37 expression is unresolved

There is a further operation on the results of an unresolved expression.
Alternatively, the operand in an EQU or other symbol definition directive or an ORG directive contains an unresolved expression.

E38 illegal expression format

This message indicates a basic mistake in expression syntax—unbalanced parentheses or brackets for example.

E39 invalid relocatable expression

Relocatable symbols do not support the specified operator.

E40 division by zero

The divisor operand in a division or modulo expression is 0.

E41 illegal bit offset

The right operand (bit offset) in a dot operator expression is not a constant.
Alternatively, the bit offset used in bit addressing is not a constant or has a value greater than 7.

E42 right expression of SEG operator must be address

The SEG operator cannot be used with an operand of usage type NUMBER.

E44 illegal core name

The #CORE statement in the DCL file does not contain a valid CPU core name.

E46 mnemonic required

The #INSTRUCTION statement in the DCL file must be followed by instruction mnemonics.

E48 #ENDCASE without #CASE

An #ENDCASE statement appears in the DCL file without a matching #CASE statement.

E51 CODE segment only

Microcontroller instructions, GJMP directives, GCAL directives, CLINE directives, and CLINEA directives can only appear in CODE segments.

E52 GJMP/GBcond operand must be symbol

The GJMP and GBcond directives accept only symbols as operands.

E54 out of relative jump range

Either the current location and the branch target address are in different physical segments or the distance from the first is outside the range -128 to +127.

E55 LABEL or NAME format error

A statement uses a label instead of a symbol name or vice versa. The following examples trigger this error message.

```
LABEL:   EQU   100H
NAMES    DS    100H
```

E56 invalid CPU instruction

The program uses an instruction mnemonic that is not defined in the DCL file's #INSTRUCTION statement.

E57 invalid initialization directive

The assembler initialization directives (ROMWINDOW, NOROMWIN, MODEL) do not appear in their correct position at the start of the file—that is, they are preceded by some other instruction or directive. The following example triggers this error message.

```
EXTRN   NUMBER : MAXADDRESS  ; ROMWINDOW, NOROMWIN, and MODEL
                             ; cannot appear after this.
NOROMWIN

MODEL   LARGE
```

E58 illegal SFR word/byte attribute

An SFR access attribute definition in the DCL file has a syntax error in the word/byte access attribute field.

E59 illegal SFR bit attribute

An SFR access attribute definition in the DCL file has a syntax error in the bit access attribute field.

E60 out of SFR address range

An SFR access attribute definition in the DCL file uses an SFR address that lies outside the region defined with the SFR keyword.

E61 misplaced ENDIF directive

An ENDIF directive appears without a matching conditional assembly start directive (IF, IFDEF, or IFNDEF).

E62 misplaced ELSE directive

An ELSE directive appears without a matching conditional assembly start directive (IF, IFDEF, or IFNDEF).

E63 unexpected end of file in conditional directive

There is no corresponding ENDIF directive for a conditional assembly start directive (IF, IFDEF, IFNDEF). This error always appears at the last line of the program.

E64 too many conditional directive nesting levels

Conditional directives are nested more than 15 levels deep.

E65 too many macro nesting levels

Macros are nested more than eight levels deep.

E67 symbol for CDB directive not defined

A CSLOCAL, CSGLOBAL, or CLABEL directive cannot find the specified local symbol.

E71 label or '$' is not allowed

When branch optimization is in effect, labels defined in the CODE segment, the current location symbol, and certain others are treated the same as forward symbol references, so are therefore not allowed as operands in CSEG or EQU directives or other contexts not accepting forward symbol references.

E72 invalid NEAR/FAR

There is a syntax error in a NEAR address specifier—a NEAR address specifier with an address in physical segment #1 or higher, for example.

E73 usage type NUMBER expected

The context requires an expression of type NUMBER.
The following example triggers this error message.

```
ADDR  EQU     1:1234H
    MOV     QR0, ADDR:[EA+]
```

E74 cannot write to ROM

The program attempts to write to a ROM region.

E75 invalid fn_id

The operand fn_id in a C source level debugging information directive has an incorrect value.

E76 invalid block_id

The operand block_id in a C source level debugging information directive has an incorrect value.

E77 cfunction cannot nest

CFUNCTION-CFUNCTIONEND directive blocks overlap—because a CFUNCTIONEND directive has been accidentally deleted or has a different fn_id, for example.

E78 invalid position

A C source level debugging information directive appears in the wrong position. The usual cause is a missing C source level debugging directive of the appropriate corresponding type.

E79 overlay location out of range

There is no ROM at the actual address assigned to an overlay CODE segment.

E80 illegal range

A /BXXX (/BRAM, /BROM, /BNVRAM, or /BNVRAMP) option has specified an address range that is inappropriate, that places the end address before the starting address, or overlaps an existing region.

E81 missing physical segment address

The OVL operand in a CSEG directive does not specify the physical segment address for the execution address.

E82 missing member directives for previous CxxxTAG directive

There are fewer CSTRUCTMEM or CENUMMEM directives than specified in the preceding CSTRUCTTAG or CENUMTAG directive.

E84 unclosed CFUNCTION directive exits

A CFUNCTION directive is missing the corresponding CFUNCTIONEND directive—that is, the assembler encountered the end of the source code file first.

E85 segment address mismatch

The segment address changes between a CFUNCTION directive and the corresponding CFUNCTIONEND directive. The assembler cannot generate correct C source level debugging information for output.

E90 duplicated CRET directive between CFUNCTION and CFUNCTIONEND

A CRET directive has been described twice or more between a CFUNCTION directive and a CFUNCTIONEND directive. RASU8 cannot output correct C debug information.

6.9.2.3 Warning Messages

Warning messages are divided into six types.

The /W and /NW command line options respectively enable and disable checking for the specified warning types using letters from the following list.

Letter	Warning Type
R	Check relocatable segment definitions
P	Check directives
E	Check expression syntax
A	Check addressing syntax
S	Check SFR access attributes
T	Check ROMWINDOW region specifications

To enable checking for warning types R, E, and T, for example, use the /WRET command line option.

The following list of warning messages indicates the warning type using these letters in parentheses after the warning number.

W01(R) stack size must be even

Specifying an odd value for the stack size causes the assembler to boost the size one to the next even number.

W02(P) duplicate option or directive

The same directive or option appears more than once. The duplicate is ignored.

W05(E) expression of type address required

Applying the OFFSET operator to an expression that is already a numerical value triggers this warning message.

W06(E) expression of type NUMBER required

Specifying an expression of type address as the right operand for a BYTE1, BYTE2, BYTE3, BYTE4, WORD1, or WORD2 operator or in an addressing type accepting only usage type NUMBER triggers this warning message.
The following is an example.

```
ADRSYM  EQU     0:2345H
IS78H   EQU     BYTE1 01:5678H    ; Warning 06
        LEA     2:3000H           ; Warning 06
        L       R0, R5:ADRSYM           ; Warning 06
        ADD     SP, #0:12H        ; Warning 06
        SWI     #0:12H            ; Warning 06
```

All lines except the first trigger this warning message.

W08(E) segment address mismatch

The two address operands on either side of the operator do not have the same segment address.

W09(E) address attribute not inherited

The expression loses its address attribute and is treated as a numerical value.

W10(E) cannot check physical segment address

The assembler is unable to guarantee that the physical segment addresses match.

W11(E) right expression of operator must be NUMBER

The right operand for a shift operator must be of usage type NUMBER.

W13(E) left expression of bit operator must be byte address

The left operand for the dot operator must be an expression of type byte.

W16(E) BPOS operator should be used only on bit address

The right operand for the BPOS operator must be an expression of type bit.

W25(S) illegal access to SFR

This warning message indicates illegal access to the SFR region—a write to a read-only register or word access to a register supporting only byte access, for example.

W26(S) cannot access to high byte in SFR word

This warning message indicates byte access to a register supporting only word access.

W28(A) cannot access to high byte

This warning message indicates word access to an odd-numbered address in RAM.

W29(A) cannot write to ROM window

The program attempts to write to an address in the ROM window region.

W31(E) reference before first definition

A symbol defined with SET directives is referenced before the first definition. The symbol therefore assumes the last defined value.

W35(A) branch address must be even

The offset to the branch target address for a relative branch instruction must be even.

W36(A) physical segment address not determined

The assembler cannot determine whether to use NEAR or FAR addressing, so postpones address checking to the linker.

W37(T) ROMWINDOW/NOROMWIN is not specified

In the absence of either directive, the assembler cannot check physical segment number #0 addresses in the data memory space.

W38(A) current location aligned

The immediately preceding instruction left an odd value in the current location counter for a CODE segment, so the assembler boosts the contents one to the next even address.

W39(A) address out of range

There is no memory at the address specified with the instruction operand. Either there is no memory at the address specified in a CODE, DATA, or other symbol definition directive or the programmer has forgotten to add the appropriate external memory region with a /BXXX (/BRAM, /BROM, /BNVRAM, or /BNVRAMP) option.

W40(A) physical segment address out of range

This warning message indicates that there is no memory for the corresponding segment type in the physical segment address specified in a SEGMENT directive. One possible reason is that the programmer has forgotten to add the appropriate external memory region with a /BXXX (/BRAM, /BROM, /BNVRAM, or /BNVRAMP) option.

W42(A) cannot load this data by L instruction

A constant code is described in memory that cannot be accessed as data or cannot be

determined to do so. Change the placement address of the constant code, or suppress a check with a NOCHKDBDW directive.

W48(A) DSR prefix generated

When the physical segment of a data memory is #0, DSR prefix code is unnecessary.

W49(R) ABL file format is old. Please rebuild ABL file by the latest linker

The specified ABL file is generated by old version linker. Please rebuild and generate the ABL file by the latest linker.

6.9.2.4 Internal Processing Error Messages

**** RASU8 Internal Error : Process [*function*] ****

These error messages indicate a problem detected within the assembler itself. The field *function* is a text string indicating the position.

These error messages should normally not appear, so, if they do, contact your nearest LAPIS Semiconductor sales office.

7 RLU8 Linker

7.1 Overview

The linker RLU8 merges object files created with the relocatable assembler RASU8 into an absolute object file.

In addition to object files, this tool also accepts input from library files created with the librarian LIBU8, extracting object modules from them using one of the following three approaches.

(1) Extracting all object modules

(2) Extracting only the specified object modules

(3) Extracting only the object modules necessary to search to resolve unresolved external references

This document sometimes abbreviates object module to simply module.

The absolute object file created by the linker contains object code with all relocatable portions assigned final addresses. There are also options for including debugging information in this output.

The linker also generates a map file listing segment assignments and public symbols for use in determining segment starting addresses, etc. for debugging purposes.

This document refers to addresses closer to 0 as lower memory and ones closer to 0FFFFH as upper memory.

7.2 RLU8 Operating Procedures

7.2.1 Command Line Syntax

The following is the RLU8 command line syntax.

RLU8 *object_files* [, [*absolute_file*] [, [*map_file*][, [*libraries*]]]] [;]

The commas (,) divide the command line into four fields.

The *object_files* field specifies the names of the object and library files to link.

The *absolute_file* field changes the output file name from the default.

The *map_file* field changes the map file name from the default.

The *libraries* field specifies the names of library files to search to resolve unresolved external references.

Options for changing linker operation from the defaults can appear in any field before the semicolon.

All fields other than the first are optional. To skip a field, leave it blank by typing only the trailing comma. Hitting the Return key instead displays a prompt asking for input for that field.

The semicolon (;) at the end of the command line signals the end of user input. Without it, the linker displays prompts asking for input for any remaining fields. Adding it skips these and uses the defaults for those fields.

A file name specification is a Windows long file name up to 255 bytes long, including path and file extension. Note, however, that it must not contain spaces.

The file extension starts at the last period (.) in the file name portion. To specify a file name with no file extension, end it with a period (.). Otherwise, the linker uses the default file extension for that field.

If a file specification does not include a path, the linker uses the current directory on the current drive. To read from or create a file somewhere other than this default path, therefore, include a path in the file specification.

All fields other than the first accept file specifications consisting of a path alone. This path must, however, end with a backslash (\)—\USR\APDIR\, for example. Otherwise, the linker treats the last directory portion as the base name, adding the default file extension for that field.

The rest of this Section describes the fields separately.

7.2.1.1 *object_files* Field

This field specifies the names of the object and library files to link. It must contain at least one file name. The default file extension is .OBJ.

Separate multiple file specifications with spaces or plus signs (+). To continue on a new line, type a plus sign (+) as the last character and hit the Return key. Note, however, that it is not possible to split a file specification across two or more lines.

This field also accepts library files, but the file extension must be .LIB. Specifying any other file extension causes the linker to treat that file as an object file. Leaving the file extension blank produces the same result because the default used is .OBJ.

Specifying a library file here instead of in the field *library_filename* links in all object modules from the library file regardless of whether they are necessary to resolve unresolved external references. The net effect is the same as typing the names of the object files used to make that library file. A single name is, of course, much easier to type on the command line or at a linker prompt.

library_filename (*module_name* …)

It is also possible to link only specific modules by following the library file name with parentheses containing a list of module names separated with spaces. The linker then skips all modules not specified.

```
RLU8 MAIN PROJECT.LIB ( GETDATA CALC DISPLAY );
```

This example links MAIN.OBJ with the modules GETDATA, CALC, and DISPLAY from the library file PROJECT.LIB.

7.2.1.1.1 File Search Strategy

For each object or library file in the *object_files* field, the linker searches one of two directories.

(1) Overt path from the file specification

(2) Current directory on the current disk

If the linker does not find the file in the chosen directory, it aborts with an error message.

7.2.1.1.2 Displaying Help Screen

Leaving the *object_files* field blank by simply hitting the Return key displays an input prompt. Hitting the Return key a second time displays a brief screen summarizing the command line syntax.

7.2.1.2 *absolute_file* Field

This field changes the output file name from the default.

The default file extension is .ABS. Leaving this field blank creates a file with the default path and base name from the first file specification in the *object_files* field, but with this default extension. Specifying only a path—with a trailing backslash (\)—creates a file with that path, the default base name, and the default extension. If this field is nonempty but specifies no overt path, the output path is the current directory on the current drive.

7.2.1.3 *map_file* Field

This field changes the map file name from the default.

A map file is a text file listing link results. For further details on the map file format, see Section 7.7 “Map Files.”

The default file extension is .MAP. Leaving this field blank creates a file with the default path and base name from the absolute object file name, but with this default extension. Specifying only a path—with a trailing backslash (\)—creates a file with that path, the default base name, and the default extension. If this field is nonempty but specifies no overt path, the output path is the current directory on the current drive.

To skip the map file output, type NUL in this field.

7.2.1.4 *libraries* Field

This field specifies the names of library files to search to resolve unresolved external references. The default file extension is .LIB.

Separate multiple file specifications with spaces or plus signs (+). To continue on a new line, type a plus sign (+) as the last character and hit the Return key. Note, however, that it is not possible to split a file specification across two or more lines.

The linker searches the library files in the order in which they appear in this field.

If the file specification includes an overt path, the linker looks only in that directory. Otherwise, it searches for the file in the following directories in the following order.

(1) current directory

(2) directories specified by the environment variable LIBU8

If any unresolved external references remain at the end of the specified list of library files, the default is to leave them unresolved. If the /CC option is specified, however, the linker continues the search with

the following emulation libraries for C language programs.

Emulation Library	Contents	Notes
LONGU8.LIB	Integer arithmetic library	
DOUBLEU8.LIB	Double-precision floating point library	CCU8 options determine which precision to use for floating point arithmetic.
FLOATU8.LIB	Single-precision floating point library	

The linker interprets a libraries field file specification consisting of a path alone as a directory to search for these emulation libraries. This path must, however, end with a backslash (\). Otherwise, the linker treats the last directory portion as the base name, adding the default file extension .LIB.

The linker searches for these emulation libraries in the following directories in the following order.

(1) current directory

(2) directories specified in the libraries field

(3) directories specified by the environment variable LIBU8

If the linker does not find the emulation libraries in these directories, it aborts with an error message.

7.2.1.5 Command Line Examples

This Section discusses specific examples of RLU8 command lines.

■ **Example** ■

```
RLU8 MAIN CALC DISP , , MAINLIST , USER.LIB
RLU8 MAIN + CALC + DISP , , MAINLIST , USER.LIB
```

These two command lines specify exactly the same things to the linker: link object files MAIN.OBJ, CALC.OBJ, and DISP.OBJ to create an absolute object file with the default name (MAIN.ABS), name the map file MAINLIST.MAP, and resolve external references with the library file USER.LIB.

■ **Example** ■

```
RLU8 MAIN CALC DISP , , NUL ;
```

This example is the same except for the *map_file* field. The NUL suppresses map file output.

■ **Example** ■

```
RLU8 PROJECT1.LIB;
```

This example links all modules in the library file PROJECT1.LIB and produces two output files with the default base name: the absolute object file PROJECT1.ABS and the map file PROJECT1.MAP.

■ **Example** ■

```
C> RLU8 MAIN GETDATA +
INPUT FILES [.OBJ]: CALC ERRHDL +
INPUT FILES [.OBJ]: DISPLAY USER.LIB ;
```

This example splits the *object_files* field into three input lines, starting with the object files MAIN.OBJ and GETDATA.OBJ on the command line. The plus sign at the end of the first line causes the linker to prompt for further input in the same field. The second line adds CALC.OBJ and ERRHDL.OBJ, and ends with another plus sign, so the linker displays the prompt again. The third line adds DISPLAY.OBJ and USER.LIB. The semicolon at the end, however, indicates the end of both the field and the command line, so the linker skips the prompts for the remaining fields and starts processing.

7.2.2 Execution Procedures

The linker requires at least one object or library file specification to start processing.

There are three basic ways to specify linker input. These can be used either alone or in combination.

(1) Directly on the command line

(2) Interactively in response to RLU8 prompts

(3) Indirectly in a response file specified at the prescribed position on the command line

The procedures for specifying everything directly on the command line have already been discussed in Section 7.2.1 “Command Line Syntax” above, so this Section describes the other two.

7.2.2.1 Interactive Prompts

If the command line ends in the middle of a field with no semicolon to indicate the end of the command line, the linker displays a prompt asking for more input. These prompts appear as necessary in the following order.

```
INPUT FILES [.OBJ]:
OUTPUT FILE [base_name.ABS]:
MAP FILE [base_name.MAP]:
LIBRARIES [.LIB]:
```

The linker pauses after each prompt and does not display the next until it receives line input.

The following relates these prompts to the equivalent fields on the command line.

Prompt	Command Line Field
INPUT FILES	*object_files*
OUTPUT FILE	*absolute_file*
MAP FILE	*map_file*
LIBRARIES	*libraries*

To use these prompts for all fields, enter only RLU8 at the MS-DOS prompt.

Options for changing linker operation from the defaults can appear in any field before the semicolon.

The brackets after the prompts indicate defaults for the corresponding fields. Simply hitting the Return key accepts the default base name *(base_name*), from the first file specification in the *object_files* field. To change this, type a different name.

To accept the defaults for all remaining prompts and skip those prompts, type a semicolon and hit the

Return key.

Omitting the file extension from a file specification causes the linker to assume the default one for the field. To specify a file name with no file extension, end it with a period (.).

Separate multiple file specifications in the same field with spaces or plus signs (+). To split long input across lines, type a plus sign (+) as the last character, hit the Return key, and continue on the new line. Note, however, that it is not possible to split a file specification across two or more lines.

7.2.2.2 Using a Response File

This form of linker input uses a text file containing responses for the above prompts. It allows you to save frequently used input as a disk file and avoid the 127-byte limit on MS-DOS command lines.

■ Note ■

Response files have limits of their own. The linker copies the response file to a buffer for parsing the command, so exceeding the buffer capacity causes the linker to abort with an error message.

■ Using a Response File ■

To switch input to a response file at any position on the command line or at any prompt, place an at mark (@) immediately before its name. There is no default file extension for response files, so always specify the complete name—with path, if necessary.

A response file specification can appear in any one of the four command line fields or as a response to the corresponding prompts.

The file contents can represent data for one field, multiple fields, or all remaining fields. The linker imposes no particular limits. It simply assigns the fields that it reads from the file to the current and subsequent fields. When it has data for all four fields or encounters a semicolon indicating the end of the command line, it ignores any text remaining in the response file or on the command line.

■ Example ■

```
RLU8 MAIN @MYOBJ.RES , MYLIB.LIB
```

The above example specifies the response file MYOBJ.RES after the object file MAIN.OBJ. Suppose that MYOBJ.RES contains the single line SUB, OUT, MAP.

The next example represents the same input using prompts.

```
C>RLU8 MAIN +
INPUT FILES [.OBJ]: @MYOBJ.RES ,
LIBRARIES [.LIB]: MYLIB.LIB
```

■ Response File Format ■

The linker recognizes only commas as field delimiters. It treats line feeds as spaces. Command line options can appear in any field preceding the terminating semicolon.

For the benefit of human readers, response files can contain comments. These start with a // combination or sharp (#). The linker skips that and all characters from there to the line feed (0AH).

The following is an example of a response file. Note that the line numbers down the left edge have been added for use in the following discussion. They do not appear in the actual response file.

```
1: // TM MODEL X1
2: // RELEASE 2.3.1
3: TMX1 GETDATA CALC
4: COMP DISPLAY    #new module
5: TABLE           #original data
6: /S
7: ,,TMX1LIST,
8: // library
9: MATH.LIB
```

Lines 1, 2 and 8 contain only comments, so are completely ignored. Everything in lines 3 through 6—other than the comments starting with sharps in lines 4 and 5—represents input (TMX1... /S) for the *object_files* field. Line 7 leaves the *absolute_file* field blank and specifies TMX1LIST as the base name for the *map_file* field. Line 8 specifies MATH.LIB as the first file for the libraries field.

Suppose that we save this text under the file name TMX1.RES and run the linker with the following command line.

```
RLU8 @TMX1.RES
```

Here is how the linker interprets this response file.

(1) The linker creates the absolute object TMX1.ABS from the six object files specified (TMX1.OBJ, GETDATA.OBJ, CALC.OBJ, COMP.OBJ, DISPLAY.OBJ, and TABLE.OBJ).

(2) The linker links in any necessary modules from the library file MATH.LIB.

(3) The linker generates the map file as TMX1LIST.MAP.

(4) The /S option tells the linker to add public and communal symbol tables to the map file.

7.3 Progress Messages

Once the linker has the necessary user input, it displays a sign-on message and then the following messages in the following order to indicate the current processing stage.

```
Loading segments and symbols...
Allocating segments...
Writing fixed data...
```

The first indicates that the linker is reading in segments and symbols from the specified object files; the second, that it is assigning segments to specific addresses in the corresponding memory spaces; the third, that it is fixing up unresolved operands.

If all processing completes properly, the linker displays the following summary.

```
Absfile: absolute_file
Mapfile: map_file
Ablfile: abl_file

Linkage completed.
```

The first two lines give the absolute object and map files created by the linker; the third, the ABL file name. This last only appears, however, when the /A option is specified.

If there is an error, the linker aborts with the corresponding message for the current stage, one of the following.

```
Discontinue!  Loading error detected.
Discontinue!  Allocation error detected.
Discontinue!  Fix up error detected.
```

The first indicates an error while reading in segments and symbols from the specified object files; the second, one while assigning segments to specific addresses in the corresponding memory spaces; the third, one while fixing up unresolved operands.

7.4 Return Codes

The linker exits with one of the following return codes indicating the status to MAKE, batch file, or other caller.

Return Code	Description
0	Success
1	The linker produced at least one warning message.
2	The linker produced at least one error message.
3	The linker encountered a fatal error.
4	The linker detected a command line error.
5	The user interrupted execution with a break (Ctrl+C).

The linker does not generate an absolute object file for return codes 2, 3, or 4.

7.5 Options

Command line options control linker operation, output file format, and the like. This Section first discusses option syntax and the individual options.

7.5.1 Specifying Options

Let us start with the rules for using command line options.

7.5.1.1 Syntax

The following is the syntax for command line options.

/*option_name* [(*argument_list*)]

All options start with a switch character, followed by the option name. Some options then take argument lists in parentheses. The three can be separated with spaces.

The linker ignores case in option names. The /CODE option, for example, can also be written as /Code or /code.

7.5.1.2 Specification Positions

Command line options can appear in any one of the four command line fields, in response to their corresponding prompts, or in the corresponding response file fields. They can appear in multiple locations or together in a single location.

7.5.1.3 Name Arguments

Some options take names as their arguments. The linker distinguishes case. The following /CODE option, for example, treats the arguments MOUSE, mouse, and Mouse as different names.

```
/CODE( MOUSE mouse Mouse )
```

■ Note ■

The assembler by default distinguishes between upper and lower case in user and SFR address symbols, but also provides the /NCD option for suppressing the distinction. Specifying the /NCD option, however, converts all user defined symbols to upper case for output to the object file. All symbols in modules assembled with the /NCD must, therefore, appear in upper case when used as arguments to RLU8 options.

7.5.1.4 Address Arguments

Some options take memory addresses as their arguments. An address specification consists of a physical segment address, a colon (:), and an offset.

[*physical_seg* :]*offset*

The first, optional field (*physical_seg*) must be a physical segment address between 0 and 0FFH; the second, an offset between 0 and 0FFFFH. The physical segment address, if omitted, defaults to #0.

Note that spaces are allowed before or after the colon.

These numbers can be in decimal or hexadecimal notation. Decimal notation uses the digits 0 to 9; hexadecimal, these digits plus the letters A to F (or a to f). The latter requires the suffix H (or h). Hexadecimal 1234, for example, can be written as 1234H or 1234h.

If a hexadecimal representation starts with a letter (A to F or a to f), prefix it with 0. The hexadecimal constant C800H, for example, starts with the letter C, so add a leading zero to write it as 0C800H.

7.5.2 List of Available Options

The following lists the options provided by the linker.

An asterisk (*) in the middle column indicates that the option is the default when neither it nor its corresponding directive appears.

Option		Function
/D		Include debugging information in the output.
/ND	*	Skip debugging information in the output.
/S		Include the public symbol table in the output.
/NS	*	Skip the public symbol table in the output.
/CODE		Specify the address range for CODE segment assignments.
/DATA		Specify the address range for DATA segment assignments.
/BIT		Specify the address range for BIT segment assignments.
/NVDATA		Specify the address range for NVDATA segment assignments.
/NVBIT		Specify the address range for NVBIT segment assignments.
/TABLE		Specify the address range for TABLE segment assignments.
/ORDER		Specify the allocation order for segments with the same precedence.

Option		Function
/ROM		Specify an address range for external ROM.
/RAM		Specify an address range for external RAM.
/NVRAM		Specify an address range for external NVRAM.
/NVRAMP		Specify an address range for external NVRAM in code memory physical segment #0.
/CC		Enable automatic search of emulation libraries.
/SD		Include C source level debugging information in the output.
/NSD	*	Skip C source level debugging information in the output.
/STACK		Change the stack segment size.
/A		Generate an ABL file.
/NA	*	Skip the ABL file output.
/COMB		Combine CODE or TABLE segments.
/EXC		Suppress error messages for unresolved external symbols.
/ROMWIN		Specify the address range for the ROMWINDOW region.
/PDIF		Relax memory information checks between modules.
/OVERLAY		Specify an overlay region.
/LA		Link all the segments.
/CP		Change the priority of the table segments.

7.5.3 Option Descriptions

7.5.3.1 /D and /ND

■ Syntax ■

/D

/ND

■ Description ■

These command line options respectively enable (/D) and disable (/ND) the output of assembly level debugging information to the absolute object file. This debugging information includes names for local

symbols, public symbols, communal symbols, and segments.

Note that the /D option has no effect if the input object files do not include assembly level debugging information.

These options represent paired toggles. If the command line mixes them, the last to appear takes precedence. The default is /ND.

7.5.3.2 /S and /NS

■ Syntax ■

/S

/NS

■ Description ■

These command line options respectively enable (/S) and disable (/NS) the output of an alphabetical list of all public symbols and communal symbols to the map file. Note that the /S option has no effect if the map_file field specifies NUL, suppressing map file output.

These options represent paired toggles. If the command line mixes them, the last to appear takes precedence. The default is /NS.

7.5.3.3 /CODE, /TABLE, /DATA, /BIT, /NVDATA, and /NVBIT Options

■ Syntax ■

/CODE ([*address*] *segment_name* [-] [*address*] ...)

/TABLE ([*address*] *segment_name* [-] [*address*] ...)

/DATA ([*address*] *segment_name* [-] [*address*] ...)

/BIT ([*address*] *segment_name* [-] [*address*] ...)

/NVDATA ([*address*] *segment_name* [-] [*address*] ...)

/NVBIT ([*address*] *segment_name* [-] [*address*] ...)

■ Description ■

These command line options control the assignment of relocatable segments, so are collectively called segment assignment control options.

The field segment_name specifies the segment name; address, an address using the notation described under Section 7.5.1.4 “Address Arguments” above—1234H (or 1234h) for hexadecimal 1234, for

example. This address must be within the range available for the specified segment type.

These command line options have the same names as the logical segment types for which they control as shown in the following Table.

Segment Assignment Control Option	Segment Type
/CODE	CODE segments
/TABLE	TABLE segments
/DATA	DATA segments
/BIT	BIT segments
/NVDATA	NVDATA segments
/NVBIT	NVBIT segments

The linker normally assigns logical segments to the available ROM, RAM, and NVRAM regions in code or data memory in the order specified in Section 7.6.5.3 “Assignment Precedence” below, but assigns higher precedence to segments appearing in these command line options.

These command line options offer a choice of two assignment strategies for the specified relocatable segments.

(1) assigning to any address above the specified address

(2) assigning to the specified address

The rest of this Section describes these strategies individually and then shows how they may be combined.

(1) assigning to any address above the specified address

This approach involves specifying segment names and base addresses. The linker then treats the specified logical segments as relocatable ones.

The base address always starts with the initial value of 0, but is updated each time that an address appears inside the parentheses.

Note also that this base address restarts at 0 for each new /CODE option.

The following gives an example of these specifications.

■ Example ■

```
/CODE(SEG1 SEG2 100H SEG3 SEG4) /CODE(SEG5)
```

The base address starts with an initial value of 0, so the linker assigns segments SEG1 and SEG2 to addresses greater than or equal to 0. It interprets the next argument, 100H, as a base address update specification, so assigns segments SEG3 and SEG4 to addresses greater than or equal to 100H. Using a

separate CODE option starts over with a base address of 0, so the linker assigns segment SEG5 to an address greater than or equal to 0—and possibly between SEG2 and SEG3.

The net result is to assign the five segments to the ranges in the following Table.

Segment	Assignment Range
SEG1	0000H to FFFFH
SEG2	0000H to FFFFH
SEG3	0100H to FFFFH
SEG4	0100H to FFFFH
SEG5	0000H to FFFFH

(2) assigning to the specified address

Following a segment name with a hyphen (-) and an address specifies assignment to that address. The linker then treats the corresponding segment as an absolute segment.

The following gives an example of these specifications.

```
/CODE(SEG1-3800H SEG2-8000H SEG3-1:2000H)
```

This example assigns segment SEG1 to address 3800H, SEG2 to 8000H, and SEG3 to 1:2000H.

Note that these specifications can be split.

```
/CODE(SEG1-3800H) /CODE(SEG2-8000H) /CODE(SEG3-1:2000H)
```

(3) Mixing the two assignment types

The above two approaches can also be combined as shown in the following example.

```
/CODE(0F0H SEG1 SEG2-100H SEG3 200H SEG4 SEG5-1:300H SEG6)
```

These specifications assign the segments to the following ranges.

Segment	Assignment Range
SEG1	00F0H to FFFFH
SEG2	0100H
SEG3	0100H to FFFFH
SEG4	0200H to FFFFH

Segment	Assignment Range
SEG5	1:0300H
SEG6	1:0300H to 1:FFFFH

The address specified by the segment assignment control option can disagree with the boundary attribute specified for that segment in the source program. The linker ignores that attribute, assigns the segment to the specified address, and issues a warning message.

■ Aside ■

The /BIT and /NVBIT options interpret any addresses given as bit addresses.

They accept addresses in the following formats.

(1) a bit address using the notation described under Section 7.5.1.4 “Address Arguments” above

(2) a byte address, the dot operator (.), and a bit position in the following format

data_address.bit_position

The field *data_address* is a data address; *bit_position*, a numerical value from 0 to 7 representing a bit position. Both use the notation described under Section 7.5.1.4 “Address Arguments” above.

Bit 5 at data address 1234H, for example, can be written as either 91A5H or 1234H.5.

Note, however, that the highest possible address argument is 0FFFFH. The /BIT and /NVBIT options therefore require the second (dot operator) format instead of a direct specification for bit addresses above this limit. The only way to specify the bit address 7FFF3H, for example, is with 0FFFEH.3.

7.5.3.4 /ORDER

■ Syntax ■

/ORDER(*segment_name ...*)

■ Description ■

This command line option specifies the assignment order for segments with the same precedence. The linker always assigns segments in decreasing order of precedence, but the processing order for segments is arbitrary unless specified with this command line option.

Multiple /ORDER options are totally independent. Consider, for example, the following specifications for segments SEG1, SEG2, SEG3, and SEG4 all with the same precedence.

```
/ORDER(SEG1 SEG2) /ORDER(SEG3 SEG4)
```

The two /ORDER options specify that the linker assign SEG1 before SEG2 and SEG3 before SEG4, but

say nothing about the relative order for the two pairs.

Note that the segments appearing in an /ORDER option need not have the same precedence. If the segments SEG1 and SEG2 have one precedence and segments SEG3 and SEG4 a higher one, for example, the relative precedence still applies.

The above specification produces the processing order SEG3, SEG4, SEG1, and SEG2.

7.5.3.5 /ROM, /RAM, /NVRAM and /NVRAMP

■ Syntax ■

/ROM (*start_address*, *end_address*)

/RAM (*start_address*, *end_address*)

/NVRAM (*start_address*, *end_address*)

/NVRAMP (*start_address*, *end_address*)

■ Description ■

These command line options specify ranges of external memory available as ROM (/ROM), RAM (/RAM), or nonvolatile memory (NVRAM). The /NVRAM option is for physical segment #0 in the data memory space or physical segment #1 or higher in memory space; /NVRAMP, for physical segment #0 in the code memory space.

The two arguments specify the first and last addresses in the range. Both use the notation described under Section 7.5.1.4 “Address Arguments” above.

The linker adds the regions specified with these options to its list of external memory regions for use in assigning logical segments and communal symbols.

There can be more than one of these command line options. If two specifications overlap, the linker merges the two regions if they have the same memory type and ignores the overlapping addresses otherwise.

In the absence of these command line options, the linker assumes that there is no external memory installed and assigns segments and communal symbols based solely on the memory information in the object modules.

The following specifications add external ROM in the address range 1:0000H to 3:FFFFH, external RAM in the address range 4:0000H to 7:FFFFH, external nonvolatile memory in the address range 0:A000H to 0:BFFFH in physical segment #0 in the code memory space, and external nonvolatile memory in the address range 8:0000H to 9:FFFFH.

```
/ROM( 1:0000H, 3:0FFFFH )
/RAM( 4:0000H, 7:0FFFFH )
/NVRAMP( 0:0A000H, 0:0BFFFH )
/NVRAM( 8:0000H, 9:0FFFFH )
```

/ROM options specify regions for assigning CODE and TABLE segments; /RAM options, for assigning DATA and BIT segments; /NVRAM options, for assigning NVDATA and NVBIT segments.

7.5.3.6 /CC

■ Syntax ■

/CC

■ Description ■

This command line option enables automatic searching for necessary modules in the emulation libraries for C language programs. For further details, see Section 7.2.1.4 “libraries Field” above.

Specifying this option automatically adds the following options as well.

```
/COMB($$init_info $$init_info_end)
/COMB($$content_of_init $$end_of_init)
```

The segments specified by these two options are generated by the C compiler CCU8. Their primary function is initializing data necessary for executing a C language program.

7.5.3.7 /SD and /NSD

■ Syntax ■

/SD

/NSD

■ Description ■

These command line options (/SD) and disable (/NSD) output to the absolute object file of C source level debugging information for use by a C source level debugger. This output contains such information as line numbers and variable definitions. The /SD option has no effect, however, if the input object files do not include C source level debugging information.

These options represent paired toggles. If the command line mixes them, the last to appear takes precedence. The default is /NSD.

7.5.3.8 /STACK

■ Syntax ■

/STACK(*size*)

■ Description ■

This command line option specifies a new size for the stack segment, created with the assembler directive STACKSEG.

The argument size specifies the new size. Note that this size must be small enough to fit within physical segment #0. It must also be even. Specifying an odd value causes the linker to boost the size one to the next even number.

This command line option produces an error message if the input modules do not define a stack segment.

7.5.3.9 /A and /NA

■ Syntax ■

/A [(*abl_file*)]

/NA

■ Description ■

These command line options respectively enable (/A) and disable (/NA) generation of an ABL file containing information necessary for generating an absolute listing file with RASU8. For further details on absolute listing files, see Section 11.2 “Generating Absolute Listing Files.”

The optional argument *abl_file* specifies a name for the ABL file. The default name for this file is the absolute object file name with the file extension changed to .ABL.

These options represent paired toggles. If the command line mixes them, the last to appear takes precedence. The default is /NA.

7.5.3.10 /COMB

■ Syntax ■

/COMB(*segment_name1 segment_name2 ...*)

■ Description ■

This command line option joins the specified relocatable segments together in the order specified and assigns the result to the appropriate memory space. The list, delimited with spaces, can have three or even more entries. The relocatable segments must be of type CODE or TABLE, however.

The segments must all match in terms of type, special region attributes, physical segment attributes, and physical segment address. If the linker encounters a segment whose boundary alignment does not match that of the first (main) segment, it issues a warning message and chooses the larger of the two. Other mismatches produce a warning message and cause the linker to skip that segment.

The linker does not complain if does not find a particular segment.

When /CODE option or /TABLE option is specified for the second or subsequent segment (henceforth sub segment) specified by this option, the linker displays warning. And the linker ignores /CODE option or /TABLE option for the segments.

```
/COMB(SegA SegB) /CODE(100h SegA SegB)
```

In the above example, /CODE option for SegA specified by /COMB option becomes effective.

7.5.3.11 /EXC

■ Syntax ■

/EXC

■ Description ■

This command line option suppresses error messages for unresolved external symbols that are not used.

7.5.3.12 /ROMWIN

■ Syntax ■

/ROMWIN(*start_address* , *end_address*)

■ Description ■

This command line option specifies the first and last addresses in the ROM window region.

Any mismatches between this range and the ones in input modules produce an error message.

This range must also be compatible with the information in the ROM window portion of the DCL file.

7.5.3.13 /PDIF

■ Syntax ■

/PDIF

■ Description ■

This command line option relaxes the checks on memory information between modules to override differences arising from the assembler's /BXXX (/BRAM, /BROM, /BNVRAM, and /BNVRAMP) options for adding external memory. The linker issues a warning message and ORs the data for the two modules. If the memory regions overlap or are of different types, however, the linker aborts with a fatal error.

7.5.3.14 /OVERLAY

■ Syntax ■

/OVERLAY(*area_name*, *start_address*, *end_address*){[*overlay_unit*[*overlay_unit* ...]]}

■ Description ■

This command line option specifies an overlay region. It has two main parts: the overlay region specifications and the segments making up the overlay units assigned to that overlay region.

For further details on the overlay function, see Chapter 10 “Using Overlays.”

The first three arguments are a name for the overlay region, its first address, and its last. The braces enclose a list of overlay units with the following format.

■ Overlay Unit Syntax ■

UNIT(*segment* [*segment* ...])

An overlay unit specification consists of the keyword UNIT followed by, in parentheses and delimited with spaces, a list of segments making up the overlay unit.

Note that the linker assigns the segments to the overlay region in the order specified.

Overlay region definitions can overlap, but the linker issues a warning message to alert the user just in

case the overlap is not intentional.

The name for the overlay region must be unique. Duplications produce a command line error message from the linker.

An overlay region is restricted to a single physical segment. Specifications spanning multiple physical segments cause the linker to abort with an error message.

A segment making up an overlay unit can only appear once in the set of all overlay options. A duplication causes the linker to abort with a command line error message.

7.5.3.15 /LA

■ Syntax ■

/LA

■ Description ■

This command line option specifies, when it links all functions and tables.

When this option is specified, the linker treats all functions and tables as candidate for allocation to memory without checking the reference relation of the segment.

7.5.3.16 /CP

■ Syntax ■

/CP

■ Description ■

This command line option specifies, when changing the allocation priority of the TABLE segments without physical segment specification into 16. The priority 16 is the same as the priority before RLU8 V1.50.

For further details on the allocation priority of the segments, see Section 7.6.6.3 “Assignment Precedence.”

7.6 Linking

The linker uses the following procedure to create an absolute object file from the object modules from the files specified in the *object_files* field.

(1) The linker checks the module for link compatibility. Certain data must match.

(2) The linker resolves global symbols, searching the specified libraries if necessary, to resolve external symbols.

(3) The linker merges segments with the same name.

(4) The linker merges communal symbols with the same name.

(5) The reference relation for CODE/TABLE segment is checked. And the un-referenced CODE/TABLE segments are excepted from the candidate for allocation.

(6) The linker assigns segments, communal symbols, and pseudo-segments to memory.

(7) The linker fixes up unresolved operands and sends the results to the absolute object file.

The rest of this Section describes the above steps individually.

7.6.1 Checking Module Compatibility

This step investigates whether it is possible to link the specified modules.

The following lists the items checked.

(1) CPU Core

(2) Microcontroller model number

(3) Memory model

(4) Memory information

(5) ROM window attributes

7.6.1.1 CPU Matching Check of CPU Cores

CPU cores must match mutually. If they do not match, the linker aborts with a fatal error.

7.6.1.2 Checking Microcontroller Model Number

The microcontroller model number must match across modules.

Note, however, that general-purpose modules with no particular microcontroller model number of their own pass this check. The following summarizes the linker rules here.

Module 1	Module 2	Resulting attribute
General	General	General (Note, however, that this attribute produces an error message if it remains at the end.)
General	ML610001	ML610001
ML610001	ML610001	ML610001
ML610002	ML610001	This mismatch between model numbers produces an error message.

7.6.1.3 Checking Memory Model

The memory model must match across modules.

A mismatch causes the linker to abort with a fatal error.

7.6.1.4 Checking Memory Information

The memory information must match across modules.

Note, however, that the linker checks only modules with a valid microcontroller model number. The memory information in general-purpose modules is ignored.

How the linker handles mismatches depends on whether the /PDIF command line option is in effect.

(1) If the option is not specified, the linker aborts with a fatal error.

(2) If the option is specified, the linker issues a warning message and ORs the data for the two modules. If the memory regions overlap or are of different types, however, the linker aborts with a fatal error.

7.6.1.5 Checking ROM Window Attributes

The ROM window attributes must match across modules.

The following Table summarizes the results of this checking for all possible module combinations. ROMWINDOW or NOROMWIN means that the assembly language source code contains the corresponding directive; “default,” that neither is present.

Module 1	Module 2	Link compatible	Notes
Default	Default	YES	This combination produces an error message because, without a /ROMWIN option, the linker does not know the ROM window address range.

Module 1	Module 2	Link compatible	Notes
Default	ROMWINDOW	YES	Module 2 determines the final ROM window address range.
ROMWINDOW	ROMWINDOW	YES	An error message results, however, if the two ROM window address ranges do not match.
NOROMWIN	Default	NO	This combination produces an error message.
NOROMWIN	ROMWIN	NO	This combination produces an error message.
NOROMWIN	NOROMWIN	YES	The linker assumes that there is no ROM window.

7.6.2 Resolving Global Symbols

As the linker reads in the object modules from the files in the order specified by the *object_files* field, it resolves external symbols by matching them with public or communal symbols with the same name. If unresolved external symbols remain after reading all modules, the linker then searches the library files specified in the libraries field. If a library contains a module defining a public symbol with the same name as the unresolved external symbol, the linker adds that module to the list to be linked. It repeats this searching until it has resolved all external symbols.

In deciding whether a public or communal symbol from another module can resolve an external symbol, the linker checks the usage type and physical segment attributes. Only an exact match produces resolution.

If a communal symbol has the same name as a public symbol, the linker treats it as a public symbol, not a communal symbol. Whether they have the same segment type then becomes the resolution condition. If they match here too, the linker discards the information associated with the communal symbol and does not assign any memory space.

7.6.3 Merging Partial Segments

It is perfectly acceptable for multiple modules to have segments with the same name. The linker starts by merging these partial segments into a single segment using the following iterative process merging two segments at a time.

Before merging two partial segments, however, the linker checks for link compatibility. The segments must satisfy the following conditions to be linked.

(1) They must have the same segment type.

(2) If one partial segment has the special region attribute DYNAMIC, the other must have it too.

(3) The combined size, the sum of the two segment sizes, must not exceed the memory size available

for assignment.

The merged result has the following attributes.

(1) The segment type is that of the two original segments.

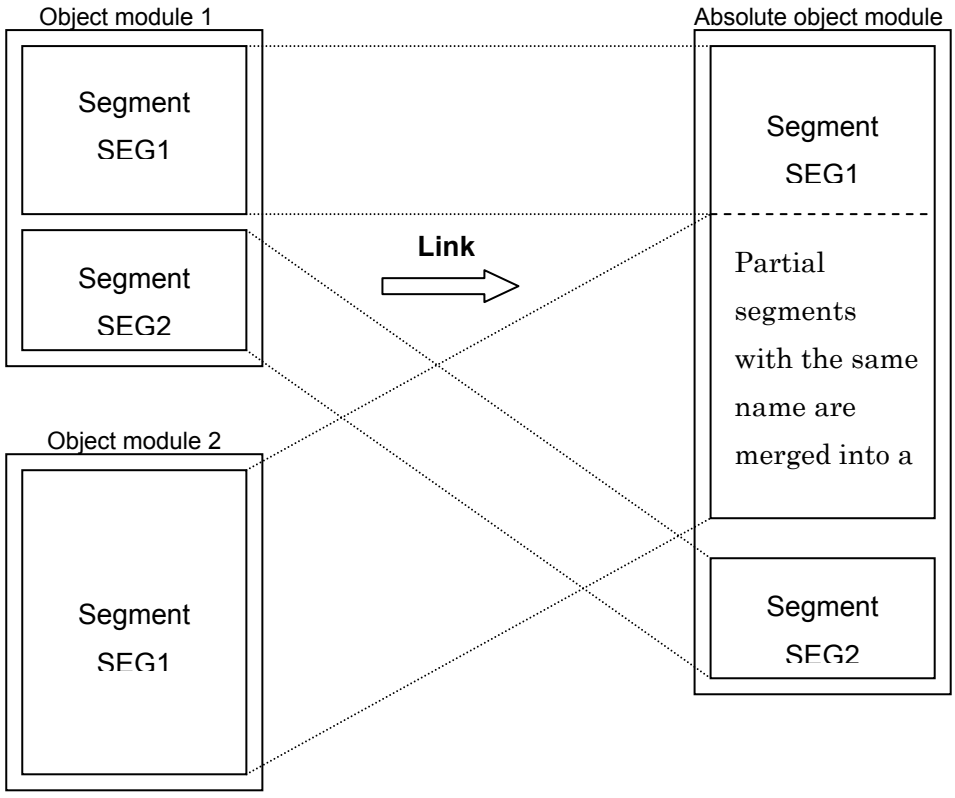
(2) The boundary attribute is the larger of the two.

(3) The size is the sum of the two segment sizes.

The linker merges partial segments in the order that they appear in modules, appending each new one to the end of the last merged result. This process produces a single segment occupying a contiguous region in memory.

After the above procedure, however, individual partial segments sometimes do not fall on the boundaries specified in the source code. Consider, for example, two partial segments with boundary alignment 2. The merged segment inherits the same boundary alignment, so starts on an even address. So does the first partial segment. Where the second one now starts depends on the size of the first in bytes. The starting address is even for an even number of bytes and odd for an odd number of bytes. The latter can have important repercussions if the user application program addresses the second segment with word-sized access.

The following illustrates the reordering involved in merging partial segments.

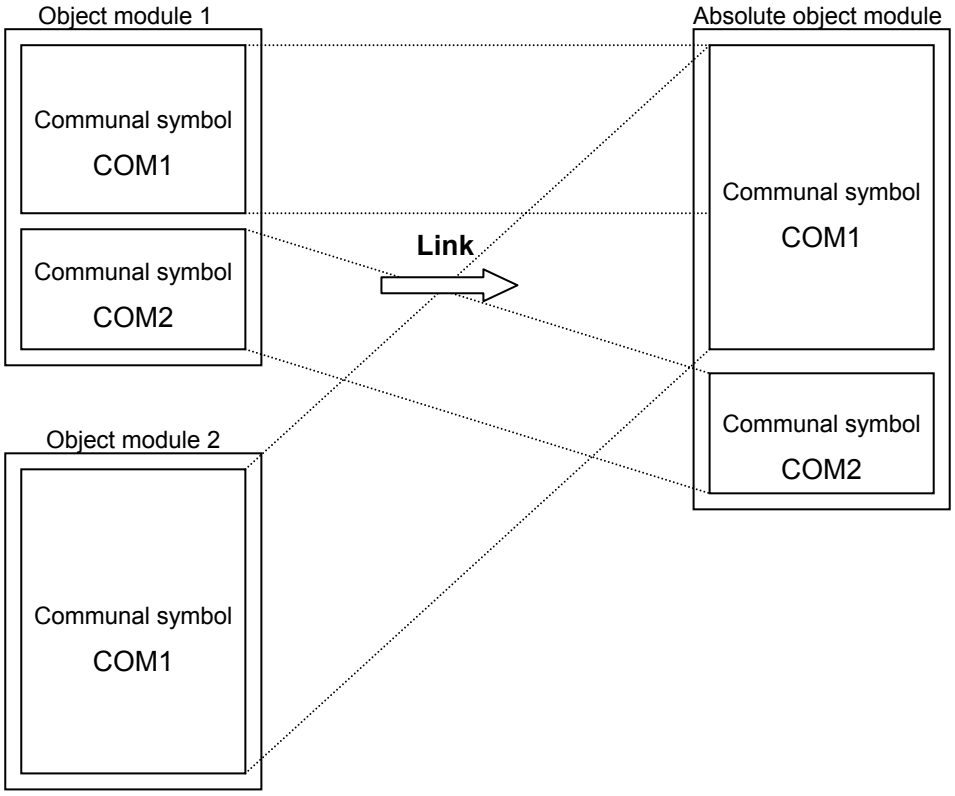


7.6.4 Merging Communal Symbols

The linker merges communal symbols with an iterative approach highly similar to the one above for segments. The only difference is that, instead of linking regions end to end, the linker overlaps them so that the communal symbols share the same starting address. The final size, therefore, is the largest encountered with the original communal symbols.

The compatibility checking is exactly the same as for linking segments.

The following illustrates the reordering involved in merging communal symbols.



7.6.5 Checking Reference Relation for Segments

When /LA option is not specified, the linker checks the reference relation for CODE/TABLE segments. And the un-referenced CODE/TABLE segments are excepted from the candidate for allocation.

7.6.6 Assigning Segments

Once the linker merged partial segments and communal symbols, it assigns segments, communal

symbols, and pseudo-segments to their memory spaces. Section 7.6.6.1 “Assignment Spaces and Regions” below describes the process for segments and communal symbols; Section 7.6.6.2 “Pseudo-Segments,” that for pseudo-segments.

The linker determines the memory regions available from the DCL file and any /ROM, /RAM, /NVRAM, or /ROMWIN command line options. For further details on these options, see Sections 7.5.3.5 “/ROM, /RAM, /NVRAM and /NVRAMP” and 7.5.3.12 “/ROMWIN” above.

The linker assigns segments to memory in decreasing order of precedence. (See Section 7.6.6.3 “Assignment Precedence” below.) The assignment order for segments with the same precedence is arbitrary. If a segment and a communal symbol have the same precedence, the segment goes first. The linker generally searches memory for a free region to assign a segment or communal symbol from address 0 upward. The only exceptions are relocatable segments of type DATA or BIT assigned to physical segment #0. Then, it searches from 0:0FFFFH downward.

The linker uses the first free region found. If the search fails, the linker aborts with an error message.

Boundary alignment for segments is that specified by the source code.

Boundary alignment for communal symbols depends on the segment type and size: always on a bit boundary for types BIT and NVBIT regardless of size; on a byte boundary for types DATA, NVDATA, or TABLE with a size of one byte; on a word boundary for types DATA, NVDATA, or TABLE with a larger size.

Apart from boundary alignment, the linker treats segments and communal symbols the same.

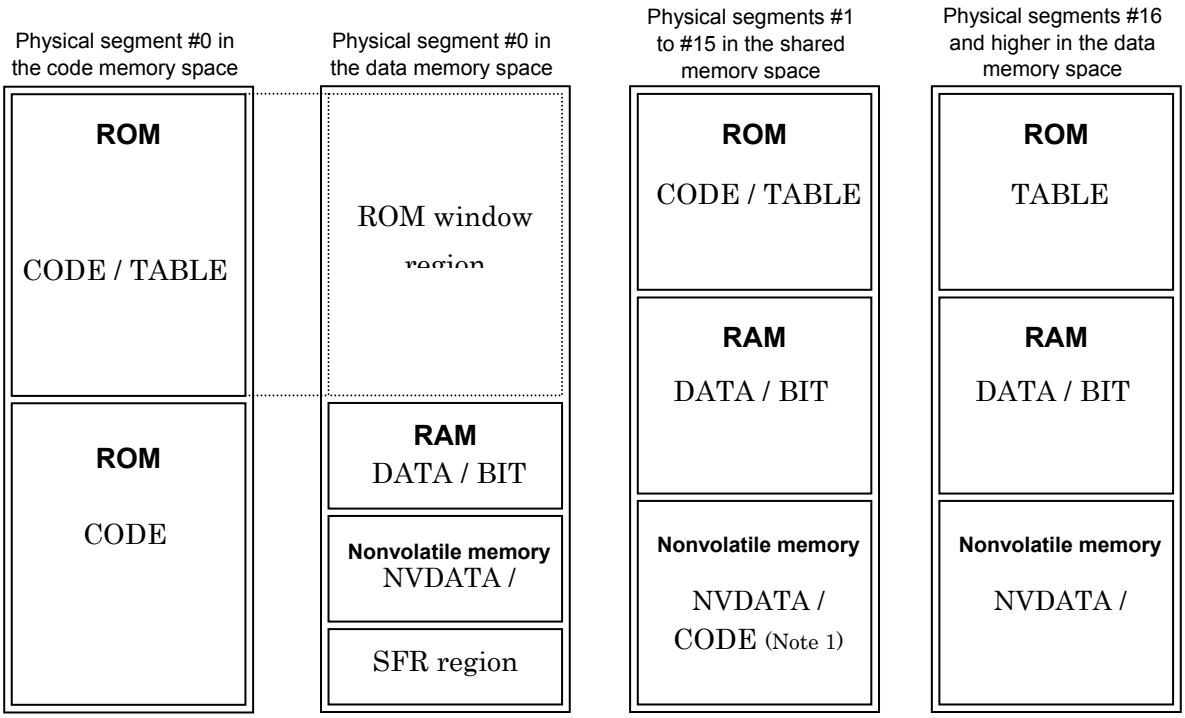
The linker never assigns segments across physical segment boundaries.

7.6.6.1 Assignment Spaces and Regions

The spaces and regions for assigning segments and communal symbols depend on the segment type, the type of memory installed, and the special region attributes. The following describes the basic process.

■ Segment and Memory Types ■

The following summarizes the relationship between the segment and memory types.



Note 1: The only relocatable CODE segments that may be assigned to nonvolatile memory are those whose segment symbol definitions specify the special region attribute NVRAM.

7.6.6.2 Pseudo-Segments

The linker automatically assigns pseudo-segments to the following memory regions to prevent user application programs from defining their own segments or communal symbols there. Note that they are not defined by the programmer.

(1) the SFR region in the data memory space

(2) the ROM window region in the data memory space

(3) overlay regions
For further details on the overlay function, see Chapter 10 “Using Overlays.”

7.6.6.3 Assignment Precedence

The linker gives precedence first to segments with absolute addresses, next to ones with option specifications, and then to ones with the strictest conditions. The following summarizes the order of precedence.

Precedence	Segment Conditions
1	All absolute segments defined with CSEG, DSEG, BSEG, NVSEG, NVBSEG, and TSEG directives plus pseudo-segments automatically generated by RLU8
2	All segments assigned absolute addresses with /CODE, /DATA, /BIT, /NVDATA, /NVBIT, and /TABLE options—the following, for example `/CODE( CodeSegment-1:2000H )`
3	Relocatable TABLE segments assigned to physical segment #0 with a /TABLE option, but not an absolute address—the following, for example `/TABLE( 0:1000H TableSegment0 )`
4	Relocatable TABLE segments assigned to physical segment #0 Or, relocatable TABLE segments without the physical segment specification in the case of the following conditions: - Without /CP option - ROM exists only in physical segment #0
5	Relocatable NVDATA and NVBIT segments without absolute addresses that are arguments to /NVDATA and /NVBIT options
6	Relocatable CODE segments without absolute addresses that are arguments to /CODE options
7	Relocatable TABLE segments without absolute addresses that are arguments to /TABLE options and assigned to a physical segment other than #0—the following, for example `/TABLE( 1:1000H TableSegment1 )`
8	Relocatable DATA and BIT segments without absolute addresses that are arguments to /DATA and /BIT options
9	Relocatable NVDATA and NVBIT segments with physical segment specifications
10	Relocatable CODE segments with physical segment specifications

Precedence	Segment Conditions
11	Relocatable TABLE segments with physical segment specifications other than #0
12	DATA and BIT segments with physical segment specifications—except for the stack segment and any dynamic segments
13	Stack segment
14	NVDATA and BIT segments other than the above, without physical segment specifications
15	CODE segments other than the above, without physical segment specifications
16	Without /CP option: Relocatable TABLE segments without physical segment specifications when ROM exists in physical segment #1 or higher With /CP option: All relocatable TABLE segments without physical segment specifications
17	DATA and BIT segments without physical segment specifications—except for the stack segment and any dynamic segments
18	Dynamic segments

7.6.7 Fix-Up Processing

Once the linker has assigned absolute values to all symbols, it next fixes up operands left unresolved by the assembler.

For most normal programs, the assembler is unable to assign absolute values to all portions—operands using relocatable symbols, for example. The assembler therefore temporarily assigns the value 0 to that portion and includes a fix-up record for the linker in the object file.

The following is the procedure for a single fix-up record.

(1) The linker calculates the absolute value from the fix-up record.

(2) The linker checks that value, as necessary.

(3) The linker replaces the dummy zero from the assembler with the new value.

The linker repeats the above procedure for each fix-up record.

7.7 Map Files

This section describes the layout of map files.

A map file starts with the following portion, which shows such things as the start-up options from the RLU8 command line.

(1)
```
RLU8 Object Linker, Ver.1.00.5 Linkage Information
[Thu Jun 08 16:21:15 2000]



                          ------------------
                           Control Synopsis
                          ------------------
```

(2)
```
I/O controls:    D NSD  S NA
```

(3)
```
Locating controls:

     Type     Address             Name
     ----     -------             ----
     CODE     At    01:1000       CSEG02
     DATA     After 07:8000       DSEG02
```

(4)
```
Other controls: STACK( 0100H(256) )
```

(1) The map file starts with a header giving the RLU8 version number and the map file creation date and time.

(2) This portion gives the start-up settings for the command line options /D, /ND, /SD, /NSD, /S, /NS, /A, and /NA.

(3) This portion summarizes any segment assignment control options (/CODE, /TABLE, /DATA, /BIT, /NVDATA, and /NVBIT) from the command line. The Address column gives, in hexadecimal, the *address* specified preceded with a keyword indicating whether that address is a base address (After) or an absolute one (At).

(4) This portion lists any command line options not already appearing under (2) or (3) above.

The next section gives the following information on the modules processed.

```
                        ------------------------
                         Object Module Synopsis
                        ------------------------

                                                                    Creator
(5)  Module Name           File Name                                CCU8    RASU8
     -----------           ---------                                -----   -----
     sample1               sample1.obj                              3.07    1.30
     sample2               sample2.obj                              -.--    1.30


(6)  Number of Modules: 2

     Number of Symbols:
(7)   +-----------------------------------------------------------------------------+
      |           | CODE | DATA | BIT  |NVDATA|NVBIT |TABLE |NUMBER| TBIT || total |
      |-----------+------+------+------+------+------+------+------+------++-------|
      | SEGMENT   |     3|     3|     0|     2|     0|     2|      |      ||     10|
      |-----------+------+------+------+------+------+------+------+------++-------|
      | COMMUNAL  |     0|     1|     0|     0|     0|     0|      |      ||      1|
      |-----------+------+------+------+------+------+------+------+------++-------|
      | PUBLIC    |     1|     1|     0|     1|     0|     1|     0|     0||      4|
      +-----------------------------------------------------------------------------+

     Target:  ML610001
(8)  Model:   LARGE
(9)

     Memory Map - Program memory space #0:
           Type     Start      Stop
(10)      ----------------------------
           ROM      00:0000    00:7FFF
           NVRAM    00:8000    00:FFFF

     Memory Map - Data memory space #0:
           Type     Start      Stop
          ----------------------------
           RAM      00:0000    00:7FFF
           NVRAM    00:8000    00:9FFF
           RAM      00:A000    00:FFFF

     Memory Map - Memory space above #1:
           Type     Start      Stop
          ----------------------------
           ROM      01:0000    06:FFFF
           RAM      07:0000    09:FFFF
           NVRAM    0A:0000    0C:FFFF
           ROM      0D:0000    0F:7FFF
```

(5) This portion lists the names of the modules processed, the names of the files containing them, and the respective version numbers of CCU8 and RASU8 that created them. If the module processed has no CCU8 version information, "-.--" is displayed in the CCU8 field.

(6) This line gives the number of modules processed.

(7) This portion gives the number of segments, communal symbols, and public symbols found in all modules processed for each segment type. Note that merged segments and communal symbols are only counted once each.

(8) This line gives the name of the target microcontroller for the program.

(9) This line gives the memory model for the program: SMALL or LARGE.

(10) This portion gives memory maps for physical segment #0 in the code memory space, for physical segment #0 in the data memory space, and for physical segments higher than #1. The first column gives the type of memory installed; the other two, the address range in hexadecimal.

The next section gives link maps, segment and communal symbol assignments, for each memory space.

```
                        ------------------
                         Segment Synopsis
                        ------------------

(11) Link Map - Program memory space #0 ( ROMWINDOW: 0000 - 3FFF )

      Type     Start      Stop       Size              Name
     ------------------------------------------------------
      S CODE   00:0000    00:0003    0004(4)           (absolute)
      S TABLE  00:0004    00:0041    003E(62)          TSEG01
      S CODE   00:0042    00:0083    0042(66)          CSEG01


(12) Link Map - Data memory space #0

      Type     Start      Stop       Size              Name
     ------------------------------------------------------
      Q ROMWIN 00:0000    00:3FFF    4000(16384)       (ROMWIN)
>GAP<          00:4000.0  00:7FFF.7  4000.0(16384.0)   (RAM)
      S NVDATA 00:8000    00:803F    0040(64)          NVDSEG01
>GAP<          00:8040.0  00:9FFF.7  1FC0.0(8128.0)    (NVRAM)
>GAP<          00:A000.0  00:FDBF.7  5DC0.0(24000.0)   (RAM)
      S DATA   00:FDC0    00:FEBF    0100(256)         $STACK
      S DATA   00:FEC0    00:FEFF    0040(64)          DSEG01
      Q SFR    00:FF00    00:FFFF    0100(256)         (SFR)


(13) Link Map - Memory space above #1

      Type     Start      Stop       Size              Name
     ------------------------------------------------------
      S TABLE  01:0000    01:004F    0050(80)          TSEG02
>GAP<          01:0050.0  01:0FFF.7  0FB0.0(4016.0)    (ROM)
      S CODE   01:1000    01:103F    0040(64)          CSEG02
 ---
      C DATA   07:0000    07:0003    0004(4)           DCOM01
>GAP<          07:0004.0  07:7FFF.7  7FFC.0(32764.0)   (RAM)
```

(14)

```
      S DATA   07:8000    07:803F    0040(64)          DSEG02
 ---
      S NVDATA 0A:0000    0A:003F    0040(64)          NVDSEG02


Total size (CODE  ) = 00086   (134)
Total size (DATA  ) = 00184   (388)
Total size (BIT   ) = 00000.0 (0.0)
Total size (NVDATA) = 00080   (128)
Total size (NVBIT ) = 00000.0 (0.0)
Total size (TABLE ) = 0008E   (142)
```

(11) This portion gives a link map for physical segment #0 in code memory.
If the user application program defines a ROM window region, the address range appears in the first line in the following format.

```
( ROMWINDOW: 0000 - 3FFF )
```

If there is no ROM window region defined, the following notation appears.

```
( ROMWINDOW: Not exist )
```

The single letters S, C, and Q starting each line in the Type column indicate segments, communal symbols, pseudo-segments, respectively. The remainder of the Type column is either the segment type or, for a pseudo-segment, the corresponding notation from the following Table.

Type and Name Column Notations for Pseudo-Segments

Pseudo-segment	Type column notation	Name column notation
SFR	SFR	(SFR)
ROM WINDOW	ROMWIN	(ROMWIN)
Overlay region	OVRLAY	(*area_name*) where *area_name* is the name for the overlay region specified with the /OVERLAY option.

The Start and Stop columns give the address range in hexadecimal.

The Size column gives the size of the segment in hexadecimal and decimal.

The Name column gives the segment name. The notation (absolute) indicates an absolute segment. An asterisk (*) to the right of a name indicates a segment specified in an /OVERLAY option. Pseudo-segments use the names from the Table above.

The following messages sometimes appear to the left of the Type field.

Message	Description
>GAP<	The address range is free. The Name column then gives the memory type for this free region in parentheses: (ROM), (RAM), or nonvolatile memory (NVRAM).
OVL	Segments overlap in memory.
OUT	The address specified for the segment with a /DATA, /CODE, or similar option or the absolute segment falls outside the region available.
---	This notations indicates a physical segment boundary.

The totals represent the total space occupied by segments and communal symbols in both hexadecimal and decimal. They do not include the sizes of pseudo-segments.

(12) This portion gives a link map for physical segment #0 in data memory. The display format is the same as that for the code memory space above.

(13) This portion gives a link map for memory in physical segments #1 and higher. The display format is the same as that for physical segment #0.

(14) This portion gives totals representing the total space occupied by segments and communal symbols for each segment type in both hexadecimal and decimal. These totals do not include the sizes of pseudo-segments.

If the linker is, as the result of an error, unable to assign a segment or communal symbol anywhere, the linker uses the following format.

```
Ignored 2 segments or communal symbols:
      S CODE                          3393(13203)       TEST_C_SEG
      S DATA                          2415(9237)        TEST_D_SEG
```

The segment of the function/table which is not allocated to a memory in order that it is not referred to is displayed for each module as follows.

```
Not Linked Segments:
      Module Name          Type       Size              Segment Name
     ------------------------------------------------------------
      file1
                           CODE       0100(256)         $$func1$file1
                           CODE       0120(288)         $$func3$file1
     ------------------------------------------------------------
      file2
                           CODE       0200(512)         $$func10$file2
     ------------------------------------------------------------
```

The segment names shown in the above example are segments which the compiler outputs by default. For the user defined segment, the defined name is output. The function name or the table name is included in the name which is inserted between $$ and $ of the segment which a compiler outputs by default. Using this information, it can know which function or table is not linked.

The above portions represent the standard ones that always appear. The linker provides command line options for sending additional information to the map file.

The rest of this Section describes these additions.

Specifying the /D or /SD option adds the symbol table from the debugging information sent to the absolute object file. This listing groups symbols by module.

```
       ----------------------
        Symbol Table Synopsis
       ----------------------

Module          Value     Type          Symbol
------          -----     ----------    ------
sample1
               0000001C   Loc NUMBER    _$$WINVAL
                00:3FFF   Loc TABLE     _$$ROMWINEND
                00:0000   Loc TABLE     _$$ROMWINSTART
                00:8000   Pub NVDATA    NVDATA1
                00:FEC0   Pub DATA      BUF1
                00:0004   Pub TABLE     STRING
                00:0042   Pub CODE      _START


Module          Value     Type          Symbol
------          -----     ----------    ------
sample2
               0000001C   Loc NUMBER    _$$WINVAL
                00:3FFF   Loc TABLE     _$$ROMWINEND
                00:0000   Loc TABLE     _$$ROMWINSTART
```

The notations Pub and Loc in the Type column respectively indicate public and local symbols.

Specifying the /S option adds an alphabetical list of all public and communal symbols used in the program.

```
Public Symbols Reference

Symbol      Value       Type     Module
------      -----       ----     ------
BUF1        00:FEC0     DATA     sample1
NVDATA1     00:8000     NVDATA   sample1
STRING      00:0004     TABLE    sample1
_$$SP       00:FEC0     DATA     sample1
_START      00:0042     CODE     sample1
```

If there is no error more serious than a warning, the linker ends the map file with the following line.

```
End of mapfile.
```

7.8 Error Messages

The linker reports errors detected with error messages to standard output.

The linker classifies errors into the following types.

Command line errors

A command line error is a syntax or other error so serious that the linker cannot continue. The linker therefore aborts immediately.

Fatal errors

A fatal error is an I/O or other critical error so serious that the linker cannot continue. The linker therefore aborts immediately.

Linker errors

A linker error is one that prevents creating the absolute object file. The linker aborts with an error message.

Warnings

A warning alerts the programmer to a potential problem with the program. The linker simply issues a warning message and continues, generating the absolute object.

7.8.1 Error Message Format

Linker error messages have the following format.

■ Syntax ■

error_type error_code : *message*

The *error_type* field gives the error group from the following list.

error_type	**Error type**
Command line error	Command line error
Fatal error	Fatal error
Error	Error
Warning	Warning

The *error_code* and message fields represent the actual error message, an error code and text combination from Section 7.9 “Error Message List” below.

The linker augments the basic error message with the file name, symbol name, and other information necessary for elucidating the source of the error. Errors detected during fix-ups, for example, provide three such pieces of information.

offset / *segment_name* / *module_name*

This line pinpoints the error location right down to the object *module*, *segment*, and *offset*.

This offset represents a number appearing in the location column in the assembly listing portion of the print file created by the assembler. Consider the following RASU8 print file TEST.PRN, for example.

```
## Loc. Object                    Line   Source Statements

                                     1       TYPE (M610001)
                                     2
                                     3       MODEL   LARGE, FAR
                                     4       ROMWINDOW   0, 3FFFH
                                     5
                                     6   EXTRN   NONE:ODD_ADDR EVEN_ADDR
                                     7   PROG    SEGMENT CODE
                                     8
  --------------------------         9       RSEG    PROG
??:0000 00 E0                       10       MOV     ER0,    #0
??:0002 00'E3 13-90 00-00'          11       ST      ER0,    ODD_ADDR
??:0008 00'E3 13-90 00-00'          12       ST      ER0,    EVEN_ADDR
```

Now suppose that linking this object module produces the following warning message.

```
  Warning W006: Cannot access high byte, 0006/PROG/TEST
```

This message warns of a problem detected while fixing up unresolved data at offset 0006 in the segment PROC defined in the module TEST. Examining the print file reveals that the problem is with the symbol ODD_ADDR used as an instruction operand in line 11.

7.9 Error Message List

The following lists linker error messages by error type. To the left of the text is an error code giving the error type and a unique number for the error message. Following the text is a description.

7.9.1 Command Line Error Messages

C001 Command line syntax error

There is a problem with a command line specification.

C002 Duplicate segment name '*segment_name*'

The same segment name appears more than once in the /COMB or /OVERLAY options. Such duplications cause the linker to abort with an error message.

C003 Duplicate overlay area name

The names for all overlay regions must be unique. Eliminate all such duplications.

C004 Invalid overlay area

There is a problem with the overlay range specified in an /OVERLAY option.

C005 Invalid numerical argument

The specified value falls outside the permissible range (1 to 65536).

C006 Bad constant

There is a problem with a numerical value specified as an argument to an option. This message indicates a missing letter “H” at the end of a hexadecimal constant—/CODE(SEG1-0F00), for example.

C007 Unrecognized option name

The linker does not support the specified option.

C008 Missing object file name

The RLU8 command line must specify at least one object file.

C009 Invalid specified area

There is a problem with the specified address range.

C010 Invalid ROM window

There is a problem with the address range specified for the ROM window.

C011 Command line buffer overflow

The input string is too long for the command parsing buffer.

7.9.2 Fatal Error Messages

F001 Insufficient memory

There is not enough memory to continue.

F002 Cannot open file

The linker cannot open the specified file.

F003 Cannot close file

The linker cannot close the specified file.

F005 Versions not compatible

There is a problem with the assembler version number.

F006 Bad module

The object module appears to be damaged, so try reassembling it.

F007 Record length too long

A record in the object module is too long. The object module appears to be damaged, so try reassembling it.

F008 Checksum error

The linker has detected a checksum error. The object module appears to be damaged, so try reassembling it.

F009 Invalid Core ID

The linker is trying to link object modules of different CPU cores. Specify the same DCL file in all assembly source code files using a TYPE directive, reassemble, and relink.

F010 Inconsistent machine name

The object modules use different microcontroller model numbers. Edit the source file TYPE directives, reassemble, and relink.

F011 Inconsistent memory model

The object modules use different memory models. Edit the source file MODEL directives, reassemble, and relink.

F014 Invalid family ID

An object module is not for the nX-U8 architecture.

F015 Inconsistent memory values

The object modules contain different memory information. Reassemble with consistent /BXXX (/BRAM, /BROM, /BNVRAM, and /BNVRAMP) options for adding external memory regions and relink. Specifying the /PDIF option sometimes eliminates these error messages.

F016 Illegal translator ID

The linker RLU8 accepts only object modules created with the assembler RASU8.

F017 Illegal object module format

The specified object file is not in the nX-U8 format.

F018 Illegal library type

The specified library file is not in the nX-U8 format.

F019 File seek error

There was a file I/O error.

F020 Cannot write, disk full!?

There was a file I/O error writing to disk. The most likely cause is insufficient disk space.

F021 Not a library file

The specified file is not a library file.

F022 Specified module not found

The linker cannot find the specified module in the specified library in an *object_files* field specification of the form library(*module_name*...). Check the module and library names and relink.

F023 All machine names are STANDARD

There must be at least one module that is not general-purpose—that is, specifies a microcontroller name and thus a DCL file.

F024 File used in conflicting contexts

Input and output files have the same name. Use a different name for the output file.

F025 No ROM window specification

Either overtly specify a ROM window region in at least one module or specify the /ROMWIN option.

F026 ROM window mismatch

The object modules contain different ROM window specifications. Edit the source files, reassemble, and relink.

F027 Inconsistent /ROMWIN values

There is a problem with the range specified for a /ROMWIN option.

F028 NOROMWIN is specified

The command line mixes modules with NOROMWIN specified and the /ROMWIN option.

F029 ROM WINDOW attribute mismatch

The object modules contain different ROM window attribute specifications. Edit the source files, reassemble, and relink.

F030 Inconsistent memory values (VECTOR)

The object modules contain different vector region specifications. The most likely cause is mixing modules for different microcontrollers. Edit the source files so that the TYPE directives all specify the same DCL file, reassemble, and relink.

F031 Inconsistent memory values (ROMWIN)

The object modules contain different ROM window region specifications. The most likely cause is mixing modules for different microcontrollers. Edit the source files so that the TYPE directives all specify the same DCL file, reassemble, and relink.

F032 Inconsistent memory values (SFR)

The object modules contain different SFR region specifications. The most likely cause is mixing modules for different microcontrollers. Edit the source files so that the TYPE directives all specify the same DCL file, reassemble, and relink.

F033 Inconsistent memory values (INTERNAL RAM)

The object modules contain different internal RAM region specifications. The most likely cause is

mixing modules for different microcontrollers. Edit the source files so that the TYPE directives all specify the same DCL file, reassemble, and relink.

F034 Inconsistent max physical segment for Data memory '*module_name*'

This error message is displayed when the upper limit of the segment in the data memory space of the module the linker is trying to link is different.

The most likely error cause is an attempt to link the object file in which the /nofar option is specified with the startup file that does not support the /nofar option when compiling. To link with a file in which the /nofar option is specified when compiling, specify the startup file that supports the /nofar option (the file in which a NOFAR directive is described).

F035 Inconsistent max physical segment for Program memory ' *module_name* '

This error message is displayed when the upper limit of the segment in the program memory space of the module the linker is trying to link is different.

The most likely error cause is an attempt to link the object file in which the /nofar option is specified with the startup file that does not support the /nofar option when compiling. To link with a file in which the /nofar option is specified when compiling, specify the startup file that supports the /nofar option (the file in which a NOFAR directive is described).

7.9.3 Linker Error Messages

E001 Segment type mismatch

The segment types for partial segments do not match.

E002 Physical segment attribute mismatch

The physical segment attributes for partial segments do not match.

E003 Physical segment address mismatch

The physical segment addresses for partial segments do not match.

E004 Special region attribute mismatch

The special region attributes for partial segments do not match.

E005 Segment size out of range

The segment size exceeds the maximum, 64K bytes.

E006 Out of overlay area

The segments making up the overlay unit exceed the overlay range available. Try such measures as making the overlay range larger.

E007 Physical segment address out of range

The physical segment address is outside the range available.

E008 Offset out of range

The range available is 0 to 0FFFFH.

E009 Duplicate public symbol

A public symbol with the same already exists. All public symbols must have unique names.

E010 Unresolved external symbol

Unresolved external symbols remain. If the symbols are declared external, but never actually referenced, use the /EXC option to suppress these error messages.

E011 Cannot find segment

The segment specified by the option was not found.

E012 Control type mismatch

A segment specified with a /CODE, /DATA, /BIT, /NVDATA, /NVBIT, or /TABLE option has the wrong type—the name of a segment with segment type CODE appears in a /DATA option, for example.

E013 Cannot change physical segment address

A /CODE, /DATA, /BIT, /NVDATA, /NVBIT, or /TABLE option attempts to change the physical segment address from that determined by the assembler. Such changes are not allowed.

E015 Segment not allocated

The linker is unable to assign a segment because of insufficient free space or for some other reason.

E018 Out of range: physical segment address

The physical segment address after the fix-up is outside the acceptable range.

E019 Out of range: from *min* to *max*

The value after the fix-up is outside the specified acceptable range.

E020 Out of relative jump range

The jump distance value after the fix-up is outside the specified acceptable range.

E021 Out of SWI address range

The vector address is outside the SWI address range.

E022 Out of SWI number

The SWI number is outside the specified acceptable range (0 to 63).

E023 Vector address must be #0

The vector address must assigned to physical segment number #0.

E024 Out of area: memory range not available

There is no memory in the accessed region.

E025 Usage type mismatch

Two external, communal, or public symbols fail the matching checks because their usage types do not match. Both symbols must have the same usage type.

E026 Physical segment attribute mismatch

Two external, communal, or public symbols fail the matching checks because their physical segment attributes do not match. Both symbols must have the same physical segment attributes.

E027 Physical segment address mismatch

Two external, communal, or public symbols fail the matching checks because their physical segment addresses do not match. Both symbols must have the same physical segment address.

E028 CDB information might be incorrect

There is a potential problem with the C source level debugging information from the C compiler. Contact your nearest LAPIS Semiconductor sales office.

7.9.4 Warning Messages

W001 /NVRAMP is valid for only #0

The range specified with the /NVRAMP option must be in physical segment #0. To add nonvolatile memory in physical segment #1 or higher, use the /NVRAM option instead.

W002 No stack segment, so setting _$$SP to 0

The linker found no stack segment, so sets the value of the stack symbol _$$SP to 0.

W003 No stack segment, so ignoring /STACK option

The linker ignores the /STACK option because it found no stack segment.

W004 Stack size must be even

The stack segment size must be an even value. When this warning is output, one byte is added to the stack size to be adjusted to an even size.

W005 Memory information different

The object modules contain different, but not contradictory memory information, so the linker ORs the two.

This warning message appears only when the /PDIF option is in effect.

W006 Cannot access high byte

A word-sized access instruction specifies an odd-numbered address. The corresponding address must be even.

W007 Branch address must be even

A jump target address must be even.

W008 Branch to different segment

An instruction attempts to jump from the current physical segment number to a different one.

W009 Physical segment address must be #0

The physical segment address must be #0.

W010 Cannot write to ROM

An instruction attempts to write to an address in ROM. Such writes are not allowed.

W011 CODE/TABLE segments overlap

A segment of type CODE or TABLE overlaps a previously assigned segment or pseudo-segment.

W012 DATA/BIT segments overlap

A segment of type DATA or BIT overlaps a previously assigned segment or pseudo-segment.

W013 NVDATA/NVBIT segments overlap

A segment of type NVDATA or NVBIT overlaps a previously assigned segment or pseudo-segment.

W014 Overlay area overlap

An overlay region overlaps a previously assigned overlay region or segment.

W015 Memory type mismatch

The memory type actually accessed differs from that expected for the usage type.

W016 Usage type mismatch

The usage type specified in the assembly language source code differs from that expected for the operand symbols.

W017 Vector address must be even

The vector address must be an even number.

W018 Specified stack size is too big, so adjusting to *size16* (*size10*) bytes

The linker reduces the stack segment size because there is not enough room to assign the size specified in the program or with the /STACK option. The message gives the new size in both hexadecimal and decimal.

W019 Out of allocatable memory area

The specified absolute address in a /CODE, /DATA, /BIT, /NVDATA, /NVBIT, or /TABLE option either has no memory physically present or has a segment type that is incompatible with the memory type—a DATA segment is assigned to a ROM region, for example.

W020 Overlay area must RAM or NVRAM

An overlay region is assigned to ROM instead of RAM or nonvolatile memory.

W021 Overlay segment type must be CODE

The segments making up an overlay unit must be of type CODE.

W022 Outside ROM window

A segment of type TABLE assigned to physical segment #0 is assigned outside the ROM window address range.

W023 Ignoring boundary specification

The linker ignores the boundary alignment specification.

W024 0 size segment detected

The linker assigns segments with zero length to the lowest address in the corresponding memory space.

W025 Cannot find segment

The specified segment was not found.

W026 /COMB requires CODE/TABLE segments

The segments specified with a /COMB option must be of type CODE or TABLE.

W027 Ignoring /COMB subsegment

Because the specified /COMB option first (main) segment does not exist, the linker ignores subsegments, treating them as normal relocatable segments instead.

W028 /COMB segment type mismatch

The segments specified with a /COMB option must all have the same segment type (CODE or TABLE).

W029 /COMB physical segment attribute mismatch

The segments specified with a /COMB option must all have the same physical segment attributes. The linker does not merge segments that do not match the first (main) segment.

W030 /COMB physical segment address mismatch

The segments specified with a /COMB option must all have the same physical segment address. The linker does not merge segments that do not match the first (main) segment.

W031 /COMB special region attribute mismatch

The segments specified with a /COMB option must all have the same special region attributes. The linker does not merge segments that do not match the first (main) segment.

W032 Overlay segments reference each other

A segment in one overlay unit references a segment in another within the same region.

Consider the following examples.

```
/OVERLAY(OVL1, 1:8000H, 1:8FFFH){UNIT(SEG1) UNIT(SEG2)}
/OVERLAY(OVL2, 1:8000H, 1:9FFFH){UNIT(SEG3)}
```

Segments SEG1 and SEG2 are assigned to the same overlay region. Code calling SEG1 from SEG2 or vice versa triggers this warning message.

Overlay regions OVL1 and OVL2 overlap, so SEG3 code calling, say, SEG2 or vice versa also triggers this warning message.

W033 Reset vector table not allocated

Assigning all absolute segments has left the reset vector region in physical segment #0 offsets 0 through 3 unassigned. The first two bytes in this region specify the initial value for the stack pointer; the second two, the program entry point after reset input. Failure to initialize these to known values leads to erratic operation.

```
CSEG AT 0:0000H
DW      _$$SP   ; Initial value for stack pointer
DW      START   ; Program entry point after reset input
```

The above is sample source code for setting up the vector table in the reset vector region. The CSEG directive defines an absolute CODE segment containing two words: the initial value for the stack pointer and the program entry point after reset input.

W034 Ignoring /CODE or /TABLE option for ' *segment_name* '

When /CODE option or /TABLE option is specified for the second or subsequent segment (henceforth sub segment) specified by this option, the linker displays this warning. And the linker ignores /CODE option or /TABLE option for the segments.

7.9.5 Internal Processing Error Messages

These error messages indicate a problem detected within the linker itself. They have the following format.

RLU8 internal error (*position*)

The field *position* is a text string indicating the position.

These error messages should normally not appear, but, if they do, please contact your nearest LAPIS Semiconductor sales office.

8 LIBU8 Librarian

8.1 Overview

The LIBU8 librarian merges multiple object files created by the assembler into a single file called a library file. It provides facilities for both creating and modifying these library files.

The object files added to a library file are called object modules.

The linker RLU8 then accepts object modules from these files.

8.1.1 LIBU8 Functions

LIBU8 has the following functions.

(1) Creating a new library file

(2) Adding an object module to a library file

(3) Merging one library file's object modules into a different library file

(4) Deleting object modules from a library file

(5) Replacing object modules in a library file

(6) Copying object modules from a library file to object files

(7) Extracting object modules from a library file

(8) Creating a list file

8.1.2 Advantages to Using LIBU8

Dividing program development into multiple modules frequently has the side benefit of producing general-purpose modules that may be used as is in other programs. As the number of such modules grows, however, so too does the task of managing them. Merging these individual modules into libraries brings things back under control. Specifying a library puts the linker in charge of retrieving the necessary object modules.

8.1.3 From File Name to Module Name

The librarian makes the following distinction between an object file's file name and its module name in the library file.

The file name is the complete file specification including drive, directory, and file extension; the module name, assigned by the assembler, is the source code file name from the RASU8 command line less any drive, directory, and file extension specifications—in other words, the base name. The assembler stores this name in the object file output.

Module names are case sensitive because they use the base name exactly as it appears on the command line. It is thus possible to inadvertently create separate modules for the same source code file simply by changing case.

```
RASU8 MODULE

RASU8 module
```

The above two command lines use the same input file MODULE.ASM, but the module name from the first is in upper case (MODULE) while that from the second is in lower case (module).

Case sensitivity even makes it possible to store such modules in the same library file. Any intervening changes, however, introduce the risk of unexpected differences in module behavior if you get the case wrong. We therefore recommend against using case to distinguish modules. Standardize case usage—all capitals, all lower case, initial capitals, whatever—and use unique names instead.

To view the module names in a library, create a list file.

■ Example ■

```
LIBU8  MYLIB;
```

The above command line creates the list file MYLIB.LST for the library MYLIB.LIB.

Note that LIBU8 uses object file names only for adding modules. Such operations as deleting or copying modules from a library file use module names.

8.2 Running LIBU8

The following ways are available for running the librarian.

(1) From the command line

(2) Using prompts

(3) Combining command line and prompts

(4) Using a response file

8.2.1 Command Line Operation

This approach specifies everything to the library manager at the MS-DOS field prompt using the following command line syntax.

LIBU8 *library_file* [*operations*][,[*list_file*][,[*output_library_file*]]][;]

The delimiter for the first two fields, *library_file* and *operations*, is whitespace; for the rest, a comma (,).

Field	Contents
library_file	Name of library file to create or modify
operations	List of operations on the specified library file
list_file	List file name
output_library_file	Name of output library file resulting from operations

A file name specification is a Windows long file name up to 255 bytes long, including path and file extension. Note, however, that it must not contain spaces.

The first field, *library_file*, is obligatory; the rest can be skipped with commas. A semicolon (;) indicates the end of input. The library manager ignores any text after the semicolon and skips all remaining fields. Skipped fields assume the following default values.

Field	Default Value
operations	No operations
list_file	Generate list file using the output library file name with the file extension changed to .LST
output_library_file	Same as input library file name, *library_file*

The following lists the file specification defaults for each file type.

File Type	Directory	Base Name	File Extension
Input library file	Current	Must be present	.lib
Operation file	Current	Must be present	.obj
Output library file	Same directory as input library*	Base name of input library file	.lib
List file	Same directory as output library*	Base name of output library file	.lst

* Current directory on that drive if a file name with no directory is specified.

■ Example ■

```
LIBU8  MYLIB;
```

The field *operations* is blank, so the above only creates the list file. There is no command line specification, so the name defaults to MYLIB.LST.

8.2.1.1 *library_file* Field

The field *library_file* specifies the name of the library file to create or modify. It cannot be skipped. Leaving it blank on the command line displays a prompt asking for input. Additional empty input causes LIBU8 to display its usage screen and exit.

The default file extension for library files is .LIB. The library manager interprets MYLIB, for example, as MYLIB.LIB.

A file extension other than this default can also be used. If so, it must always be specified. To specify a file name with no file extension, end it with a period (.)—MYLIB., for example.

In the absence of a specification to the contrary, the drive and directory default to the current drive and directory.

8.2.1.2 *operations* Field

This field specifies the operations to perform on the specified library file. If it is blank, the only operations are an internal check of library file contents and creation of a list file.

An operation consists of an operation symbol (+, -, %, *, or &) plus a file or module name. Multiple operations must separated with spaces as shown in the following example. Spaces are optional between the operation symbol and the accompanying operand.

```
LIBU8 MYLIB +ADCON +CALC +DISPLAY;
```

These operation symbols have the following meanings.

Operation Symbol	Function	Description
+	Add	Add the specified object file or the modules in the specified library file to the library file
-	Delete	Delete the specified module from the library file
%	Replace	Replace the specified module in the library with a new version
*	Copy	Copy the specified module from the library file to an object file. Leave the original copy of the module in the library file.
&	Extract	Extract the specified module from the library file to an object file. Delete the module from the library file.

For file names, the file extension defaults to .OBJ.

Drive, directory, and file extension specifications on module names are ignored.

■ Example ■

```
LIBU8 MYLIB +GET-KEY;
LIBU8 MYLIB +GET -KEY;
```

The first example above adds the object file GET-KEY.OBJ to the library MYLIB.LIB.

The second example deletes the module KEY from the library file MYLIB.LIB and adds the object file GET.OBJ. Note how the order of operation depends on operation precedence. For further details, see Section 8.5.8 “Operation Precedence” below.

8.2.1.3 *list_file* Field

This field specifies a name for the list file, a text file listing each module in the library and, in alphabetical order, all public symbols in the module. For further details on list files, see the Section "List File Format."

The default is to create a list file. The default name for this file is the output library file name with the file extension changed to .LST. To override this default, enter a new name in this field. To suppress file output, use NUL.

■ Example ■

```
LIBU8 MYLIB +CALC, C:\WORK\LIBLIST.;
```

This example changes the list file name to C:\WORK\LIBLIST.

■ Example ■

```
LIBU8 MYLIB %CALC, NUL;
```

The NUL in this example suppresses list file output.

8.2.1.4 *output_library_file* Field

This field specifies a name for the output library file. Leaving it blank causes the library manager to use the same name as the input library file. Use this field to change the file name from this default.

If a file with the same name already exists, the library manager backs it up by changing its file extension to .LBK.

■ Example ■

```
LIBU8 MYLIB -DISPLAY, , NEWLIB
```

This example deletes the module DISPLAY from the library file MYLIB.LIB and saves the result as NEWLIB.LIB. If NEWLIB.LIB already exists, however, the library manager first renames that file to NEWLIB.LBK. MYLIB.LIB does not change. The library manager also creates the list file NEWLIB.LST.

This field only applies when operations change the library file contents. The library manager therefore issues a warning message when it ignores the specification—that is, in the following cases.

(1) No operations are specified.

(2) The only operations are copies.

(3) The library is new.

8.2.2 Interactive Prompts

Typing LIBU8 alone on the command line produces a series of prompts asking for the input fields. These prompts appear as necessary in the following order.

The library manager pauses after each prompt and does not display the next until it receives line input.

```
Library file   :
Operations     :
List file      :
Output library :
```

The following relates these prompts to the equivalent fields on the command line.

Prompt	Command Line Field
Library file :	*library_file* field
Operations :	*operations* field
List file :	*list_file* field
Output library :	*output_library_file* field

■ Example ■

```
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 also generates the list file ABC.LST.

■ Example ■

```
Library file   : ABC
Operations     : %C:XYZ
List file      : NUL
Output library : C:DEF
```

This example replaces the module XYZ in the library file ABC.LIB with the object file C:XYZ.OBJ and saves the result C:DEF.LIB. ABC.LIB does not change.

The NUL suppresses list file output.

■ **Example** ■

```
Library file    : MYLIB ;
```

or

```
Library file    : MYLIB
Operations      : ,
```

Both represent the same operation—that is, creating the list file MYLIB.LST from library file MYLIB.LIB. The latter does not itself change.

■ **Example** ■

```
Library file    : MYLIB
Operations      : -DISPLAY
List file       : ,
Output library : NEWLIB
```

This example deletes the module DISPLAY from the library file MYLIB.LIB and saves the result as NEWLIB.LIB. If NEWLIB.LIB already exists, however, the library manager first renames that file to NEWLIB.LBK. MYLIB.LIB does not change.

The library manager also creates the list file NEWLIB.LST.

8.2.3 Combining Command Line and Prompts

If the command line ends in the middle of a field with no semicolon to indicate the end of the command line, the library manager displays prompts asking for more input.

■ **Example** ■

```
LIBU8 MYLIB +CALC
```

The above command line displays the prompt asking for more input for the field operations.

LIBU8 begins processing after the user has responded to all prompts.

```
List file       :
Output library :
```

To skip these prompts, type a semicolon at the end of the command line or prompt input. The library manager interprets a semicolon as the end of input, so skips all remaining prompts and begins processing.

```
LIBU8 MYLIB +CALC;
```

8.2.4 Using a Response File

This form of librarian input uses a text file containing data command line fields. It allows you to avoid the 127-byte limit on MS-DOS command lines and to save frequently used input as a disk file.

Note that a response file specification must appear on the command line. The librarian does not recognize them in interactive prompt input.

The first step is to create the response file with a text editor and write the data necessary for the command line fields. Split the data for each field into as many lines as you wish. The library manager recognizes only commas as field delimiters. It treats line feeds as spaces. The example below uses MYLIB.RES as the name for this response file.

For the benefit of human readers, response files can contain comments. These start with a sharp (#) or // combination. The librarian skips that and all characters from there to the line feed (0AH).

```
##############################################
#  Sample library manager response file
#  Everything from '#' or '//' through to the
#  end of the line is considered a comment
##############################################
MYLIB                       // Input library file
+A +B +C                    // Operations
+D +E +F +G +H +I +K +L ,   // Operations
LISTFILE,                   // List file
OUTLIB                      // Output library file
```

To use this response file, type the following.

```
LIBU8 @MYLIB.RES
```

The next example shows how to use a response file containing data for only some input fields.

■ Example ■

The following response file, OPERATE.RES, consists solely of data for the operation field.

```
+A +B +C +D +E +F +G +H +I +K +L
```

The following is an example of how to use it in a LIBU8 command line.

```
LIBU8 MYLIB @OPERATE.RES , LISTFILE , OUTLIB
```

8.3 Redirecting Message Output

The librarian sends all its screen display messages to the standard output device. The programmer can therefore save these to a file with standard MS-DOS redirection. Alternatively, redirecting them to NUL totally suppresses them.

■ Example ■

```
LIBU8 MYLIB +ERR ; > ERRMES
```

This example redirects screen messages to the file ERRMES.

■ Example ■

```
LIBU8 MYLIB %A %B %C ; > NUL
```

This example suppresses screen messages.

8.4 Return Codes

The library manager exits with one of the following return codes indicating the status to MAKE, batch file, or other caller.

Return Code	End Status	Description
0	Normal termination	Success
1	Warning	The library manager produced at least one warning message, but continued as directed.
2	Error	The library manager produced at least one error message and ignored the problem.
3	Fatal error	The library manager encountered an error so serious that it cannot continue.

8.5 LIBU8 Functions

LIBU8 has the following functions.

(1) Creating a new library file

(2) Adding an object module to a library file

(3) Merging one library file's object modules into a different library file

(4) Deleting object modules from a library file

(5) Replacing object modules in a library file

(6) Copying object modules from a library file to object files

(7) Extracting object modules from a library file

(8) Creating a list file

This Section describes only the first seven in detail.

The default for the eighth is to create a list file. For further details on specifying a different name for this file, see Section 8.2.1.3 "*list_file* Field" above. For further details on the list file format, see Section 8.6 "List File Format."

8.5.1 Creating a New Library

To create a new library, specify a file name in the *library_file* field—on the command line, in response to the Library file prompt, or in the response file.

The librarian automatically appends the file extension .LIB if the file specification does not include one. Although other file extensions can also be used, we recommend sticking with .LIB for library files because it is the default for both the library manager and the linker.

The librarian assumes the current drive and the current directory, respectively, if the file specification does not include them.

If the command line contains nothing beside this library file specification, the librarian displays the following confirmation prompt.

```
filename.lib - File does not exist. Create ? [Y/N]
```

Pressing n or N exits the librarian without creating the file; y or Y, creates it. Hitting the Enter key has the same effect.

If the command line contains data in other fields, the librarian skips the above prompt and automatically creates the specified library file. The approach applies even if the command line specifies blank fields with commas and a semicolon.

■ **Example** ■

```
LIBU8 NEWLIB
```

This command line displays the confirmation prompt.

■ **Example** ■

```
LIBU8 NEWLIB;
```

This command line does not display the confirmation prompt. The librarian automatically creates NEWLIB.LIB and then the list file NEWLIB.LST even though NEWLIB.LIB does not contain any modules yet.

■ **Example** ■

```
LIBU8 NEWLIB +A ;
```

This command line does not display the confirmation prompt. The librarian automatically creates NEWLIB.LIB, adds module A, and then lists this module in NEWLIB.LST.

Note that, when creating a new library, the only operation that makes sense is addition. Delete, copy, replace, and extract operations produce error messages if there are no modules in the library file.

The output library file field is ignored. Specifying an output library file when creating a new library triggers a warning message.

8.5.2 Adding a Module

■ Syntax ■

+object_file

■ Description ■

The plus (+) operator adds the specified object file to the library file.

The field *object_file* contains an MS-DOS file specification. A file name specification is a Windows long file name up to 255 bytes long, including path and file extension. Note, however, that it must not contain spaces. The file extension defaults to .OBJ.

The librarian extracts the module name from the specified file and inserts the module into the library file.

The librarian stores modules in the library file in alphabetical order.

The librarian first, however, checks the module against the library file contents. The module must not

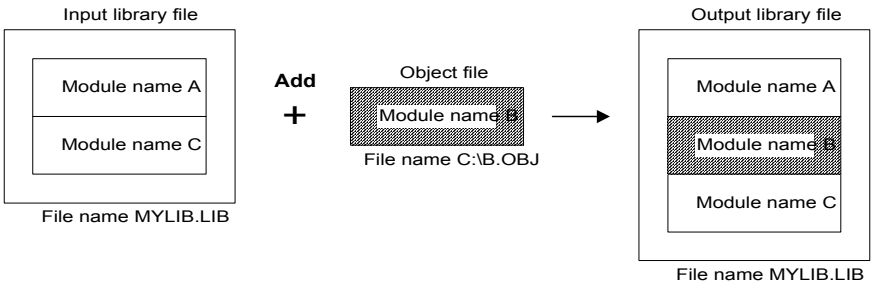
1. have the same name as one already present in the library file.

2. redefine a public symbol already defined by another module in the library file. (A public symbol is one that can be referenced from other modules because the assembler language source code makes it public with the PUBLIC directive.)

If the module fails either check, the librarian does not add it to the library file and issues an error message instead.

■ Example ■

```
LIBU8 MYLIB +C:\B.OBJ ;
```



8.5.3 Merging a Library File

■ Syntax ■

+library_file

■ Description ■

The plus (+) operator adds all modules from the specified library file to the target library file.

The field library_file contains an MS-DOS file specification. A file name specification is a Windows long file name up to 255 bytes long, including path and file extension. Note, however, that it must not contain spaces.

The file extension is obligatory because otherwise the librarian assumes .OBJ. It can be something other than .LIB, however.

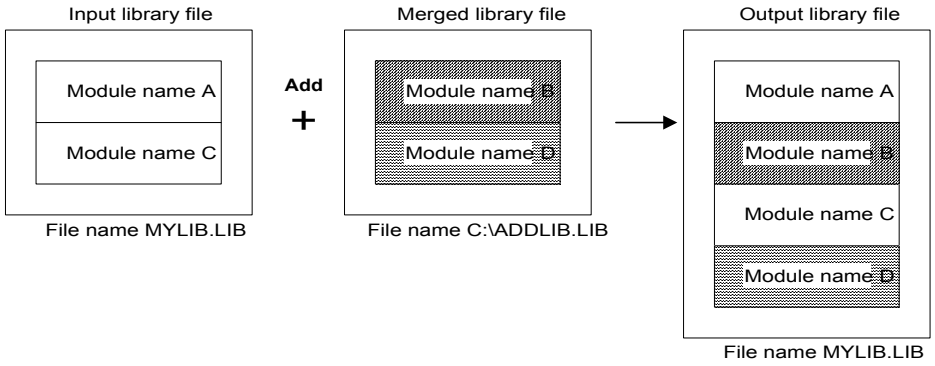
The librarian first, however, checks each module against the library file contents. The module must not

1. have the same name as one already present in the library file.
2. redefine a public symbol already defined by another module in the library file. (A public symbol is one that can be referenced from other modules because the assembler language source code makes it public with the PUBLIC directive.)

If a module fails either check, the librarian does not add it to the library file and issues an error message instead. The librarian stores modules in the library file in alphabetical order.

■ Example ■

```
LIBU8 MYLIB +C:\ADDLIB.LIB ;
```



8.5.4 Deleting a Module

■ Syntax ■

-module_name

■ Description ■

The minus (–) operator deletes the specified module from the library file.

Module names can be up to 255 bytes long.

The following example deletes module B from the library file MYLIB.LIB.

```
LIBU8 MYLIB -B ;
```

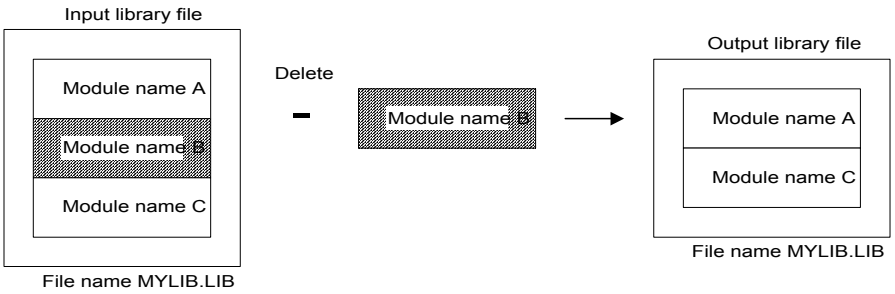
The librarian ignores any drive, directory, and file extension specifications on module names. The following command line, for example, deletes module B from MYLIB.LIB in the current directory.

```
LIBU8 MYLIB -C:\WORK\B ;
```

The librarian issues an error message if it does not find the specified module name in the library file.

■ Example ■

```
LIBU8 MYLIB -B.OBJ ;
```



8.5.5 Replacing a Module

■ Syntax ■

%object_file

■ Description ■

The replace (%) operator replaces a module in the library with the specified object file.

The field object_file contains an MS-DOS file specification. A file name specification is a Windows long file name up to 255 bytes long, including path and file extension. Note, however, that it must not contain spaces. The file extension defaults to .OBJ.

The librarian uses the base name of the object file as the module name when searching the library file. If it does not find the specified module in the library file, it aborts the operation.

The librarian first, however, checks this module against the library file contents. Any error during these checks cancels module replacement. The specified module must

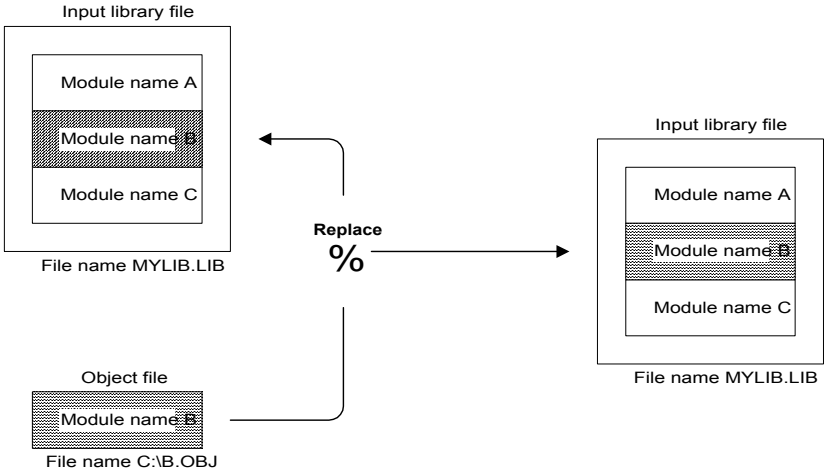
1. have a name that appears in the library file.
2. match the name in the library—case and all.
3. not redefine a public symbol already defined by another module in the library file.

If the specified module passes all the above checks without any errors, the librarian first deletes the specified module from the library file. If the delete operation succeeds, the librarian then adds the object file.

If the specified object file does not exist or there is any other type of error, the librarian leaves the current module in the library file.

■ Example ■

```
LIBU8 MYLIB %C:\B.OBJ;
```



8.5.6 Copying a Module

■ Syntax ■

**object_file*

■ Description ■

The copy (*) operator copies the specified module from the library file to an object file. Note that the original copy of the module remains in the library file.

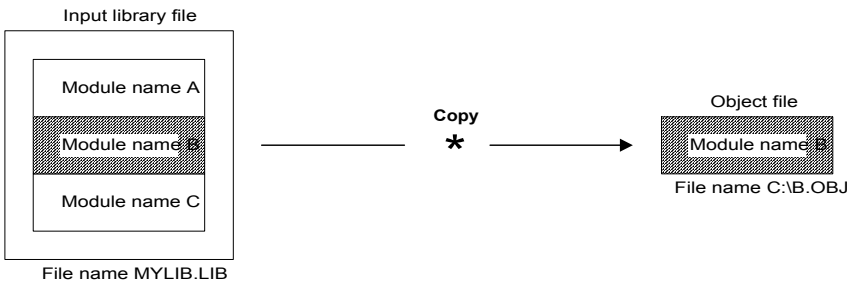
The field *object_file* contains an MS-DOS file specification. A file name specification is a Windows long file name up to 255 bytes long, including path and file extension. Note, however, that it must not contain spaces. The file extension defaults to .OBJ.

The librarian uses the base name of the object file as the module name when searching the library file. If it does not find the specified module in the library file, it aborts the operation. Otherwise, it creates the specified object file and copies the module to it. If a file with the same name exists, the librarian issues a warning message and overwrites the file.

This operation does not modify the contents of the library, so the librarian does not create an output library file if only copy operations are specified. Specifying an output library file here triggers a warning message. The librarian does not create a backup file (.LBK) either.

■ Example ■

```
LIBU8 MYLIB *C:\B.OBJ;
```



8.5.7 Extracting a Module

■ Syntax ■

&object_file

■ Description ■

The extract (&) operator copies the specified module from the library file to an object file and then deletes the original copy from the library file. This operation is thus equivalent to a copy (*) followed by a delete (-).

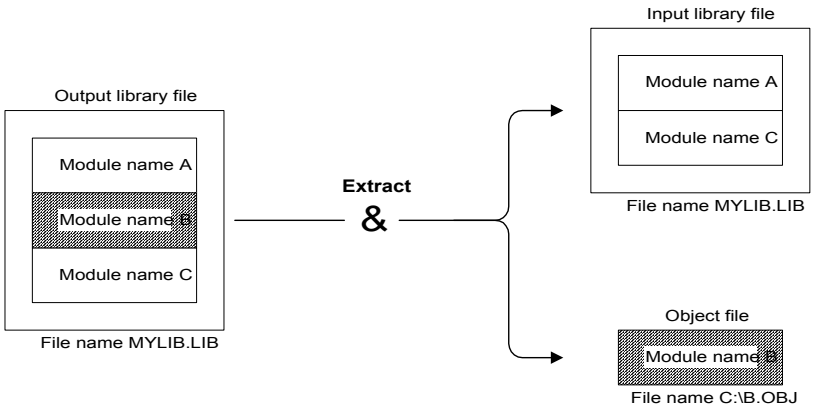
The field *object_file* contains an MS-DOS file specification. A file name specification is a Windows long file name up to 255 bytes long, including path and file extension. Note, however, that it must not contain spaces. The file extension defaults to .OBJ.

The librarian uses the base name of the object file as the module name when searching the library file. If it does not find the specified module in the library file, it aborts the operation. Otherwise, it creates the specified object file and copies the module to it. If a file with the same name exists, the librarian issues a warning message and overwrites the file.

If the copy operation succeeds, the librarian then deletes the module from the library file.

■ Example ■

```
LIBU8 MYLIB &C:\B.OBJ;
```



8.5.8 Operation Precedence

Operations have precedence levels. The librarian performs the specified operations in decreasing order of precedence regardless of the order in which they appear.

The following lists operations by precedence. The higher the precedence, the lower the priority number.

Operations with the same precedence are performed in the order in which they appear.

Precedence	Operations
1	Delete (-), copy (*), extract (&)
2	Replace (%)
3	Add (+)

■ Example ■

```
LIBU8 MYLIB +ADCON %RAMCHK -DISPLAY *CALC &EXTMEM;
```

This example first deletes the module DISPLAY from the library file MYLIB, copies CALC, and extracts EXTMEM. It then replaces the module RAMCHK. Finally, it adds ADCON.OBJ to the library file.

8.5.9 Temporary File

The librarian processes the specified operations using a temporary file ($LIBU8$.$$$) in the same directory as the output library and, just before exiting, either renames this temporary file to the name for the output library file or deletes it, depending on whether the series of operations completes successfully. This temporary file should, therefore, have disappeared when the librarian exits. It sometimes remains in the output directory, however, if the librarian exits prematurely with a fatal error or user break (Ctrl+C).

The programmer need not worry too much about such leftovers, however. They do not interfere with LIBU8 operation. The librarian also automatically overwrites any existing temporary file in the target directory the next time.

The librarian creates neither a temporary file nor an output library file if the contents of the library file do not change. There are two such cases.

1. No operations are specified.
2. Only copy operations are specified.

8.6 List File Format

A list file is a text file listing the contents of a library file. It gives information about the library file itself, a list of all modules in the library file, and lists of all public symbols defined in each module.

The default is to always create a list file. To skip the list file output, type NUL in the *list_file* field.

The following is an example of a list file. Note that the numbers in parentheses to the right have been added for use in the following discussion. They do not appear in the actual list file.

```
LIBU8 Object Librarian, Ver.1.00 Library Information
[Tue Jun 01  12:00:00  1999] ← (1)

LIBRARY FILE : test.lib ← (2)
MODULE COUNT : 1 ← (3)


MODULE NAME  : test ← (4)
DATE         : 06-01-1999  11:55:32 ← (5)
BYTE SIZE    : 0000034Eh(846) ← (6)
MEMORY MODEL : SMALL ← (7)
TRANSLATOR   : RASU8 (Ver.1.00) ← (8)
TARGET       : ML610001 ← (9)

==== PUBLIC SYMBOLS ==== ← (10)

_PubSym1    _PubSym     _PubSym3
```

The following describes the above example in the order specified by the numbers in parentheses. The first three represent information about the library file itself; the remaining seven, information about a particular module in the library file.

(1) This line gives the LIBU8 execution date and time. The fields are in the order weekday, month, day, time, and year.

(2) This line gives the name of the library file.

(3) This line gives the number of modules in the library file.

(4) This line gives the module name. Module names can be up to 255 bytes long.

(5) This line tells when the module entered the library file. The fields are in the order month, day, year, and time.

(6) This line gives the module size in bytes in both hexadecimal and decimal notation.

(7) This line gives the memory model: SMALL or LARGE.

(8) This line gives the assembler name and, in parentheses, the version number.

(9) This line gives the microcontroller target for the module.

(10) This portion lists, in alphabetical order, the public symbols defined in the module. The notation “- None -” indicates that there are none.

8.7 Error Messages

The librarian classifies errors into three types: fatal errors, errors, and warnings.

Fatal errors

A fatal error is an I/O or other critical error so serious that the library manager cannot continue. The librarian therefore aborts immediately. There are no changes in existing files. There is sometimes a temporary file left over, however.

Library manager errors

A librarian error represents an operation that is ignored. Other operations, however, are performed. The resulting file can be used. The library manager reports the number of error messages on its final screen display.

Warnings

A warning represents an operation that, while suspect, is still performed. The resulting file can be used. The librarian reports the number of warning messages on its final screen display.

8.7.1 Error Message Format

Librarian error messages have the following format.

error_level error_code: *error_message*

Some error messages feature an additional line of information about the error using one of the following formats.

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 following shows the field names used in the above formats.

Manual Notation	Screen Display
error_level	Fatal error, Error, or Warning
error_code	Error code
error_message	Descriptive message for error
library_file	Name of library file triggering error
module_name	Name of module triggering error

XXXX	Error position, a hexadecimal offset from the start of the file

The following lists librarian error messages by error type. To the left of the text is an error code giving the error type and a unique number for the error message. Following the text is a description.

8.7.2 Fatal Error Messages

001 Bad input format

There is an error in the command line.

002 Bad object filename specification

The object file name after the operation symbol either is missing or contains an invalid character.

004 Cannot open temporary file

The temporary file "$LIBU8$.$$$" is read-only. Remove this write protection with the MS_DOS ATTRIB command or equivalent.

005 Cannot rename old library

The librarian cannot change the file extension on the input library file to the backup file extension .LBK because the specified library file already has that file extension.

The librarian does not check the file extension on the input library file. If the specified operations modify the library file, it unconditionally changes the file extension on the input library file to .LBK. This rename fails, however, if the library file already has that file extension.

To use a backup file as the input library file, first change the file extension to something else with the MS-DOS REN command or equivalent.

006 Checksum error

The librarian has detected a checksum error in the current record. The library file appears to be damaged, so try rebuilding it.

008 Disk full error

There is not enough room on the disk. Delete unnecessary files or move them elsewhere to free up disk space.

009 EOF expected after module index records

There is no EOF mark at the end of the library file. The library file appears to be damaged, so try rebuilding it.

011 File name too long

The file or module name on the command line exceeds the maximum length of 255 bytes.

012 File seek error

There was a file I/O error.

013 Premature EOF

There was an EOF mark indicating the end of the redirected input before the librarian had data for all fields.

Consider the following example.

```
MYLIB  %ADCON  %CALC  %DISPLAY  , , <EOF>
```

Either end all lines with line feeds

```
MYLIB  %ADCON  %CALC  %DISPLAY  , , <line feed>
```

or specify a semicolon at the end.

```
MYLIB  %ADCON  %CALC  %DISPLAY  , ;
```

014 Insufficient memory

There is not enough memory to continue.

015 Invalid library

The specified library file contains an invalid record or invalid data.

018 I/O read error

There was an error reading a file.

019 I/O write error

There was an error writing a file.

020 ‘filename’ is write-protected

The specified file is read-only. Remove this write protection with the MS_DOS ATTRIB command or equivalent.

021 Library file might be corrupted

There is incorrect data in the library file.

031 Too many public symbols

The library file has no more room for public symbols. Either create a separate library file or cut down on the number of public symbols by eliminating the public declarations for symbols that

are not referenced from other modules. The limit, per module, when all symbols are 32 bytes long is 1983 public symbols.

032 Unable to create new library

The librarian cannot create the new library file.

034 Unable to open library

The librarian cannot open the specified library file.

037 Unable to open response file

The librarian cannot open the specified response file.

038 Unexpected end of file

The librarian was unable to read in the expected data because the file ended unexpectedly. The specified library file appears to be damaged, so try rebuilding it.

8.7.3 Library Manager Error Messages

103 Ignoring modules past 65535

There is a limit of 65,535 modules per library file. Create a separate library file.

115 Invalid library

There is a problem with the specified library file. The library file containing the module to add must have been created with LIBU8.

116 Ignoring invalid object module

There is a problem with the specified object module. The module to add must have been created with RASU8.

124 Ignoring module already in library

The specified module is already in the library file, so the librarian ignores this one.

125 Ignoring module with different name

The replace (%) operation cannot proceed because the module name in the object file does not match any module name in the library file.

126 Ignoring module not in library

The copy (*), extract (&), or delete (-) cannot proceed because the specified module name does not match any module name in the library file.

127 No room in library, so ignoring

Adding the module to the library file would exceed the 4-gigabyte limit. Either delete unnecessary modules or create a separate library file.

133 Unable to open file, so ignoring

The copy (*) or extract (&) operation cannot proceed because the librarian cannot create the specified object file.

134 Unable to open library

The librarian cannot open the library file.

135 Unable to open list file

The librarian cannot create the list file.

136 Unable to open object file, so ignoring

The operation cannot proceed because the librarian cannot open the specified object file.

140 Ignoring public symbol *public_symbol* (*file_name*) redefinition in *module_name* (*library_file*)

Adding the specified object or library file would redefine the indicated public symbol already defined in the indicated module in the library file. Change the public symbol to a unique name.

8.7.4 Warning Messages

210 Overwriting existing file in directory

The copy (*) or extract (&) operation replaces the existing file with the specified name with the object module from the library file.

226 Ignoring module not in library

The replace (%) operation cannot proceed because there is no module with the specified name in the library file.

229 Ignoring output library file specification

The librarian ignores the output library file specification from the command line because the input library file does not change. There are the following possibilities.

1. The input library file does not exist—that is, it is new.

2. No operations are specified.

3. Only copy operations are specified.

9 OHU8 Object Converter

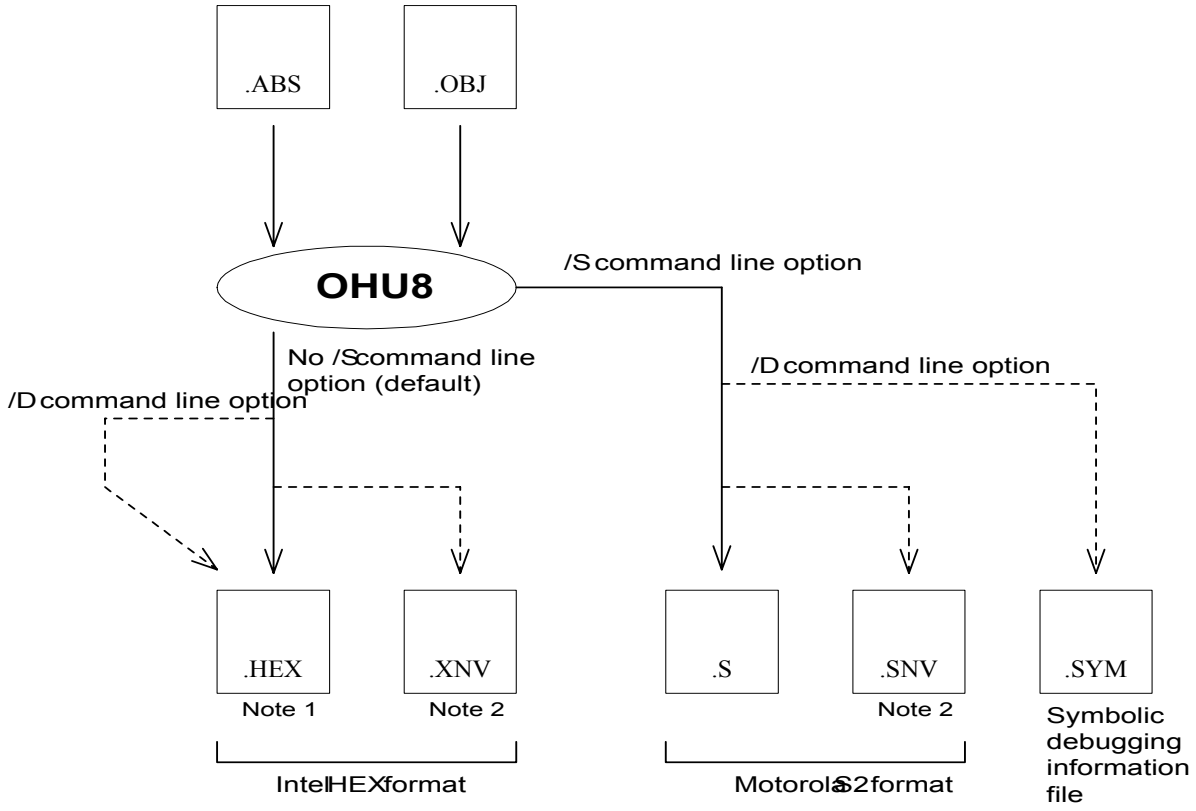
9.1 Overview

The object converter OHU8 converts object files to Intel HEX or Motorola S2 format. It only works with object files with no relocatable components—normally absolute object (.ABS) files from the linker, but also any .OBJ file from the assembler without them.

OHU8 supports two HEX file formats for output: Intel HEX and Motorola S2. This Chapter occasionally refer to such files as Intel HEX files and S2 files, respectively.

These HEX files are for use with emulators, PROM programmers, and other development tools.

The /D command line option adds debugging information for symbolic debugging with an emulator.



Note 1: For the Intel HEX format, the /D command line option adds the symbol table to the s

Note 2: This file is only produced when the user application program has object code in non\ in physical segment #0 in the data memory space.

Figure 9.1 OHU8 Data Flow

The object converter extracts only the object code from the object file and ignores all other information.

The object converter uses a separate file for any object code in nonvolatile memory in physical segment #0 in the data memory space because those addresses overlap with the code memory space.

This document uses the abbreviation NVRAM to cover EEPROM, FLASH memory, and all other types of nonvolatile memory.

The dark crosshatching in Figure 9.2 indicates object code ("program code") in physical segment #0 in the code memory space plus all object code in ROM or NVRAM in physical segments #1 or higher. The object converter saves this to a .HEX file (Intel) or an .S one (Motorola).

The lighter crosshatching in Figure 9.2 indicates object code ("NVRAM code") in nonvolatile memory. The object converter saves this to a .HNV file (Intel) or an .SNV one (Motorola).

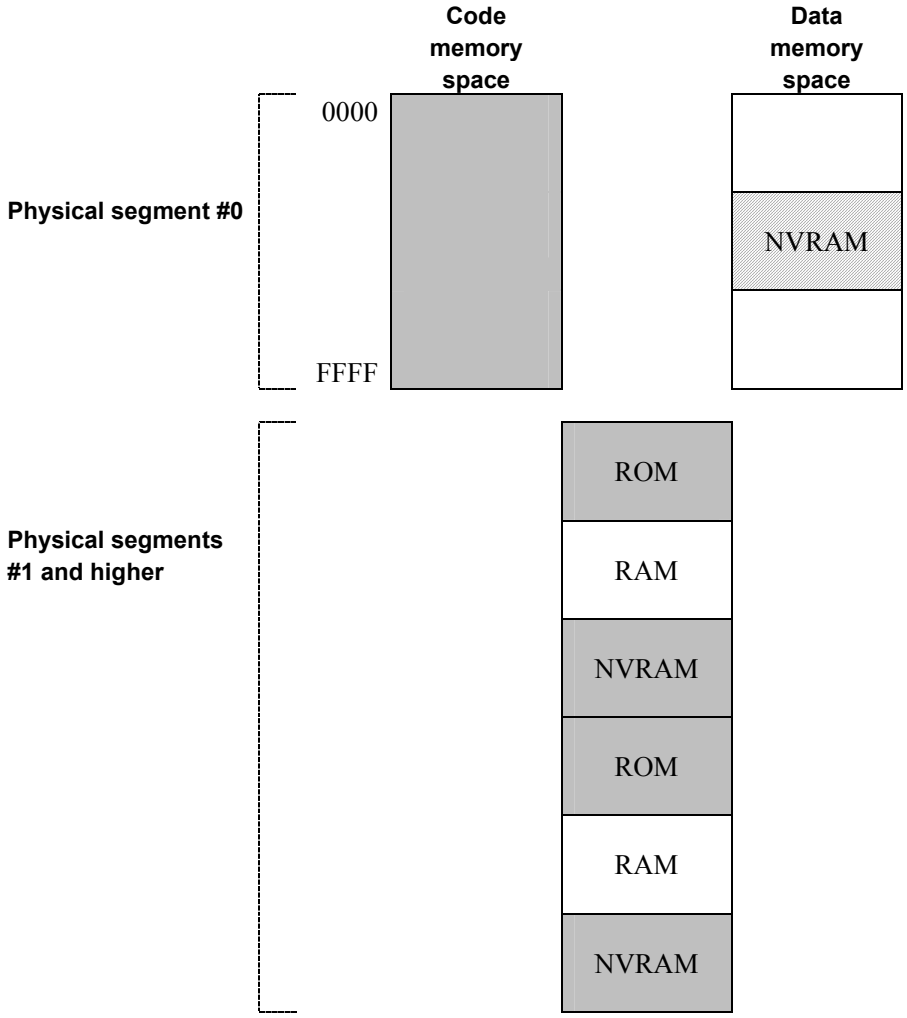


Figure 9.2 OHU8 Object Code Memory Configuration

The default is to convert this code to Intel HEX format. Adding the /S command line option switches to the Motorola S2 format.

The /D command line option adds debugging information output.

The destination files for this object code and debugging information depend on the output file format.

(1) Intel HEX format

Program code

The program code goes to a file with the same base name as the input file, but with the file extension .HEX.

NVRAM code

Any NVRAM code present goes to a file with the same base name as the input file, but with the file extension .XNV. If there is no NVRAM code, the object converter skips this output file.

Debugging information

The debugging information (if specified with the /D command line option) goes to the start of the .HEX program code file. This debugging information consists of the public symbol table.

(2) Motorola S2 format

Program code

The program code goes to a file with the same base name as the input file, but with the file extension .S.

NVRAM code

Any NVRAM code present goes to a file with the same base name as the input file, but with the file extension .SNV. If there is no NVRAM code, the object converter skips this output file.

Debugging information

The debugging information (if specified with the /D command line option) goes to a file with the same base name as the input file, but with the file extension .SYM. This debugging information consists of the public symbol table.

9.2 Execution Procedures

There are three basic ways to specify input. These can be used either alone or in combination.

(1) Directly on the command line

(2) Interactively in response to prompts

(3) Indirectly in a response file specified at the prescribed position on the command line

This Section describes these procedures individually in the above order.

9.2.1 Command Line Operation

The object converter has the following command line syntax.

OHU8 *inputfile* [*outputfile*][;]

The two fields specify the names of the input file object file and the base name for the output files, respectively.

Command line options can appear anywhere on the command line preceding the terminating semicolon.

The semicolon (;) at the end of the command line signals the end of user input. Any text after it is ignored. The object converter then skips all remaining fields, leaving them with their default settings. Omitting the semicolon produces prompts asking for input for the remaining fields.

■ Examples ■

```
OHU8 TEST;
```

This example converts TEST.ABS to Intel HEX format (TEST.HEX).

```
OHU8 /S TEST.OBJ   ;
```

This example converts TEST.ABS to Motorola S2 format (TEST.S).

The following describes the fields individually.

inputfile

This field, which specifies the name of the object file to convert, is obligatory. Leaving this field blank by simply hitting the Return/Enter key displays an input prompt. Hitting the Return/Enter key a second time displays a brief screen summarizing the command line syntax.

A file name specification is a Windows long file name up to 255 bytes long, including path and file extension. Note, however, that it must not contain spaces.

The file extension starts at the last period (.) in the file name portion. The default is .ABS, so this file extension can normally be omitted. Any other file extension used must appear explicitly.

■ Note ■

If the file name portion contains another period, .ABS is required.

```
TEST.ABS
```

The file extension .ABS is optional.

```
TEST.LONGNAME.ABS
```

Here, the file extension is required. Otherwise, the object converter uses .LONGNAME.

outputfile

This field, which specifies the base name of the output files, is optional. To skip it, type a semicolon instead.

```
OHU8 TEST.LONGNAME.ABS ;
```

Here, the output file defaults TEST.LONGNAME.HEX.

If the file name portion contains no periods, the object converter adds the default file extension: .HEX or .S. Otherwise, it uses the specified file name for the output file.

Note, however, that the object converter always uses the file extensions .XNV or .SNV for the NVRAM output file.

■ Examples ■

```
OHU8 TEST D:
```

This example converts TEST.ABS in the current directory to D:TEST.HEX.

```
OHU8 TEST    \DATA\
```

This example converts the same file to \DATA\TEST.HEX. Note how a directory specification must end with a backslash (\) Otherwise, the object converter treats the last directory portion as the base name, adding the default file extension (.HEX or .S).

```
OHU8 TEST SAMPLE /S
```

This example converts the same file to SAMPLE.S.

```
OHU8 TEST SAMPLE.HHH
```

This example converts the same file to SAMPLE.HHH. Any NVRAM code goes to SAMPLE.XNV.

```
OHU8 TEST.ABC.DEF SAMPLE.GHI.XYZ /S
```

This example converts TEST.ABC.DEF to SAMPLE.GHI.XYZ. Any NVRAM code goes to SAMPLE.GHI.SNV.

9.2.2 Interactive Prompts

Ending the command line in the middle of a field with no semicolon to indicate the end of the command line displays prompts asking for more input. These prompts appear as necessary in the following order.

Typing only OHU8 on the command line and hitting the Return/Enter key produces the following prompt.

```
INPUT FILE [.ABS] :
```

The brackets indicate the default file extension (.ABS) added if necessary.

Hitting the Return/Enter key a second time without specifying a file name displays a summary of command line syntax.

Entering an input file name displays the second prompt.

```
OUTPUT FILE(S) [input.ext] :
```

The brackets indicate the default name for the output file, the base name from the input file name plus the appropriate file extension for the output file format.

To accept this name, simply hit the Return/Enter key. Otherwise, enter a different name. At this point, you can also change the output file format with a command line option.

Processing starts after this input.

If the command line specifies only the input file name, the object converter starts with the output file name prompt.

9.2.3 Using a Response File

This form of input uses a text file containing data for command line fields. It allows you to avoid the 127-byte limit on MS-DOS command lines and to save frequently used input as a disk file.

Note that a response file specification must appear on the command line—never in interactive prompt input.

The first step is to create the response file with a text editor and write the data necessary for the command line fields. Split the data for each field into as many lines as you wish. The only field delimiters are commas. Line feeds are treated as spaces.

The following is the modified OHU8 command line syntax for using a response file.

OHU8 [*inputfile*] *@response_file* [*outputfile*][;]

The fields inputfile and outputfile have the same meanings as above—that is, the input file object file and the base name for the output files, respectively.

Command line options can appear anywhere on the command line preceding the terminating semicolon.

The semicolon (;) at the end of the command line signals the end of user input. Any text after it is ignored. The object converter then skips all remaining fields, leaving them with their default settings. Omitting the semicolon produces prompts asking for input for the remaining fields.

For the benefit of human readers, response files can contain comments. These start with a sharp (#) or // combination. The object converter skips that and all characters from there to the line feed (0AH).

The example below uses MYOH.RES as the name for this response file.

■ Example ■

The following is an example of a response file (MYOH.RES).

```
####################################################
# Sample object converter response file
# Everything from '#' or '//' through to the end
# of the line is considered a comment
####################################################
input_file        // Input absolute object file
output_file       // Output HEX file
/D                // Include debugging information in output
```

To use this response file, type the following command line.

```
OHU8 @MYOH.RES
```

The next example shows that a response file need not provide input for all fields.

■ Example ■

The following response file (MYHEX.RES) specifies only the output HEX file and debugging information output.

```
output_file /D
```

To use this response file, type the following command line.

```
OHU8 INPUT_FILE @MYHEX.RES
```

This command line produces the same results as the one above.

9.3 OHU8 Output Messages

This Section describes messages from the object converter: the sign-on message, the final summary, and any error messages arising during processing.

The object converter sends all its display messages to the standard output device. Standard MS-DOS redirection is thus available for saving them to a file, printing them (>PRN), or even suppressing them entirely (>NUL).

9.3.1 Sign-On Message

The object converter starts by displaying the following message on the screen.

```
OHU8 Object Converter, Ver.1.10
Copyright  2008-2011 LAPIS Semiconductor Co., Ltd.
```

It then reports the tool (assembler or linker) that created the input object file.

```
Object was created by translator
```

The field *translator* gives the name of the tool that created the input object file: RASU8 or RLU8.

If the object converter is unable to determine this information, it displays the following message instead.

```
Undefined translation id
```

9.3.2 Final Summary

If the /R command line option is in effect, the object converter also displays the starting and end addresses for the range converted.

```
Converted range = a:bbbb – c:dddd
```

a:bbbb ; Starting address for range converted
c:dddd ; End address for range converted

The following message indicates successful completion.

```
Convert end.
```

Error messages have the following two-line format.

```
Error : error message
File Offset : offset
```

The field *error message* is an error code and text indicating the nature of the error; *offset*, the record position in the file.

9.3.3 Return Codes

The object converter exits with one of the following return codes indicating the status to MAKE, batch file, or other caller.

Return Code	Description
0	Success
3	The object converter encountered a fatal error.

These are the only two values returned.

9.4 Options

Command line options specify the output file format (Intel HEX or Motorola S2), control debugging information output for symbolic debugging with an emulator, and specify the address range for conversion.

Command line options can appear anywhere on the command line preceding the terminating semicolon.

Option names are not case sensitive. /D and /d, for example, both produce the same results.

Multiple options must separated with spaces.

The following is a list of OHU8 command line options.

/H

Use Intel HEX format for the output file. This is the default format, so is optional. The Intel HEX format only supports addresses between 0H:0H and 0FH:0FFFFH. Addresses outside this range cause the object converter to abort with a fatal error.

/S or /HS

Use Motorola S2 format for the output file.

/D

Include debugging information in output. The destination for this debugging information depends on the output file format: the start of the .HEX program code file for the Intel HEX format and a separate .SYM file for the Motorola S2 format. If the input file does not contain debugging information, the only output record indicates the end of debugging information.

/R(*start_addr*, *end_addr*)

Limit output to the specified range. This option sends only the object code between the specified addresses to the HEX output file. Ranges must fall on 4K boundaries. Specify physical segment addesses as a number and colon before the corresponding offset. Otherwise, they default to #0.

■ Example ■

```
/R(8000h, 1:7fffh)
```

This example specifies a 64K range from 0:8000h to 1:7FFFh.

The object converter always adjusts the range so that its endpoints fall on 4K boundaries.

■ Example ■

```
/R(2:2222h, 3:3333h)
```

The new range, after adjustment, is 2:2000h to 3:3FFFh.

This example sends the object data from all content records in the adjusted address range to the output file.

The object converter accepts multiple /R command line options on the command line, but ignores all but the first.

This option affects only program code output. If there is any NVRAM code, that file contains all such code, not just that within the range specified by this option.

9.5 Files Used by OHU8

This Section describes the input and output files used by the object converter.

9.5.1 Input Files

The object converter accepts the following two types of input files.

(1) Absolute object files created by the linker

(2) Object files with no relocatable components created by the assembler. Relocatable components in the input file produce error messages.

The object converter displays a screen message indicating which tool created the input file.

9.5.2 Output Files

The object converter creates HEX files in two formats: Intel HEX and Motorola S2.

The object converter creates an NVRAM code file only if there is object code in nonvolatile memory in physical segment #0 in the data memory space. It always creates a program code file.

To prepare for symbolic debugging with an emulator, add the /D command line option to include debugging information in the output. Although both Intel HEX and Motorola S2 use the same format, the destination for this debugging information depends on the output file format: the start of the .HEX program code file for the Intel HEX format and a separate .SYM file for the Motorola S2 format.

The following describes the two HEX file formats and the shared format used for debugging information. The descriptions first give the file structure and then describe the individual records. The record descriptions start with examples of actual field output and then give field descriptions.

9.5.3 Intel HEX Files

Figure 9.3 shows the three record types for the Intel HEX format: code segment records, data records, and the file end record.

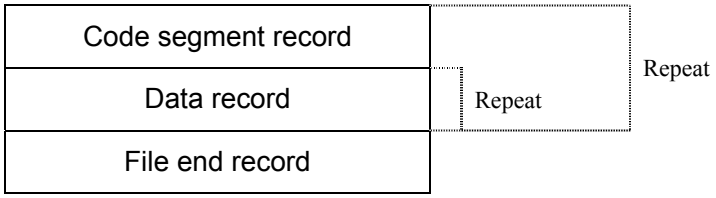


Figure 9.3 Intel HEX Format Components

■ **Note** ■

The format extends the original Intel HEX format to include physical segment addresses because the latter only has room for the offsets (0 to 0FFFFH). This extension accepts only physical segment addresses from 0H to 0FH, however, so the object converter cannot convert files with addresses above 0FH:0FFFFH.

A code segment record appears each time that the physical segment address changes. Its purpose is to indicate the physical segment address for the data records that follow. In the absence of any code segment records, the physical segment address defaults to #0.

A data record contains object code for a ROM or nonvolatile memory region.

The file end record indicates the end of the file, so appears only once, at the end of the file.

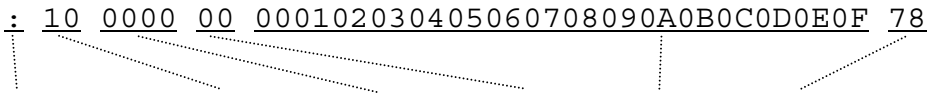
The following describes the individual record formats.

9.5.3.1 Code Segment Records

```
: 02 0000 02 F000 0C
REC_MARK  REC_LEN  LOAD_ADR  REC_TYP   DATA   CHK_SUM
```

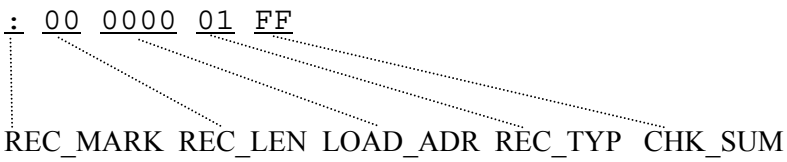
Field	Description
REC_MARK	Always a single colon
REC_LEN	Always 02
LOAD_ADR	Always 0000
REC_TYP	Always 02 to indicate a code segment record
DATA	Physical segment address in top four bits and 000 in the other twelve bits (Range: 0000 to F000)
CHK_SUM	Checksum. Twos complement of the byte formed by adding the bytes corresponding to the pairs of hexadecimal digits in the fields REC_LEN through DATA

9.5.3.2 Data Records



Field	Description
REC_MARK	Always a single colon
REC_LEN	Number of bytes of object code in the DATA field
LOAD_ADR	Load address (offset) for the first byte of object code in the DATA field
REC_TYP	Always 00 to indicate a data record
DATA	Object code bytes in hexadecimal
CHK_SUM	Checksum

9.5.3.3 File End Record



Field	Description
REC_MARK	Always a single colon
REC_LEN	Always 00
LOAD_ADR	Always 0000
REC_TYP	Always 01 to indicate a file end record
CHK_SUM	Always FF

9.5.4 Motorola S2 Format

Figure 9.4 shows the three record types for the Motorola S2 format: S0, S2, and S8.

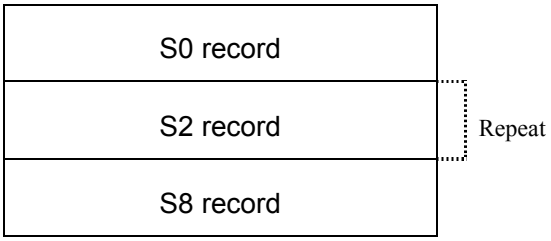


Figure 9.4 Motorola S2 Format Components

There is only one S0 record, at the start of the file.

The S2 records contain the object code for the ROM space or NVRAM region.

There is only one S8 record, at the end of the file.

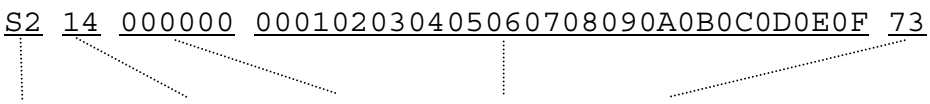
9.5.4.1 S0 Record

```
S0 0E 0000 553820444542554747 4552 FF
```

REC_TYP REC_LEN LOAD_ADR DATA CHK_SUM

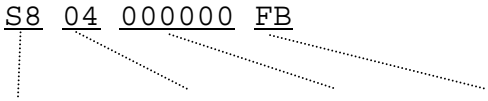
Field	Description
REC_TYP	Always S0
REC_LEN	Always 0E, the number of bytes of object code represented by the pairs of hexadecimal digits in the fields following REC_LEN (LOAD_ADR through CHK_SUM). This number is fixed here because the fields in this record have fixed widths.
LOAD_ADR	Always 0000
DATA	553820444542554747 4552, a constant reserved for use by U8 Development tools software. There is no connection with user application programs.
CHK_SUM	Always FF, the ones complement of the byte formed by adding the bytes corresponding to the pairs of hexadecimal digits in the fields REC_LEN through DATA. This number is fixed here because the fields in this record always have the same values.

9.5.4.2 S2 Records



Field	Description
REC_TYP	Always S2
REC_LEN	Same as for S0 record
LOAD_ADR	Load address for the first byte of object code in the DATA field
DATA	Object code bytes in hexadecimal
CHK_SUM	Same as for S0 record

9.5.4.3 S8 Record



Field	Description
REC_TYP	Always S8
REC_LEN	Same as for S0 record. Note that the result is always 04 because the fields in this record always have the same values.
LOAD_ADR	Always 000000
CHK_SUM	Same as for S0 record. Note that the result is always FB because the fields in this record always have the same values.

9.6 Debugging Information

This debugging information consists of a symbol table in a specific format for use in symbolic debugging. Figure 9.5 shows the two record types: debugging symbol record and debugging information end record.

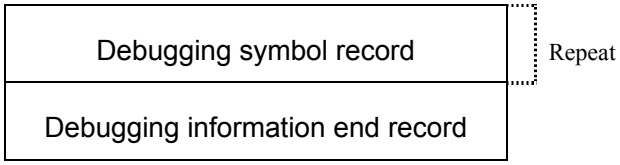


Figure 9.5 Debugging Information Components

A debugging symbol record contains information about a public symbol in the module being converted.

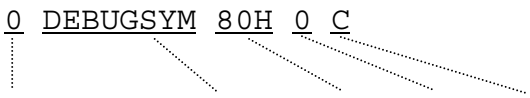
The debugging information end record indicates the end of the debugging information.

The debugging information format does not depend on the code output format.

■ Note ■

This Section describes the format of debugging information for assembly language programs. It does not cover C source level debugging information.

9.6.1 Debugging Symbol Records



Field	Description
REC_MARK	Digit 0 indicating a debugging symbol record
SYMBOL	Public symbol from module
VALUE	Symbol value in hexadecimal. For an address symbol (type other than NUMBER), this represents the offset within the specified physical segment. The maximum is thus FFFFH for a byte address and 7FFFFH for a bit address.
SEG	Physical segment address, a hexadecimal value between 0 and FFH. For a symbol of type NUMBER, this is 0.
USAGE	Usage type for symbol C : CODE D : DATA B : BIT ND : NVDATA NB : NVBIT T : TABLE TB : TBIT N : NUMBER NO : NONE

* A single space (20H) separates the fields.

9.6.2 Debugging Information End Record

_ $

This record indicates the end of the debugging information. It consists of a space (20H) and a dollar sign $ (24H).

9.7 I/O File Example

This section describes the conversion process using an actual source code file.

The first step is to assemble the source code file TEST.ASM to TEST.OBJ including the /D command line option.

```
RASU8 TEST /D
```

Source code file: TEST.ASM

```
TYPE    (M610001)
        MODEL   LARGE
        ROMWINDOW   0,3FFFH
NUM     EQU     100H
    TSEG
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
TBIT_SYM1       TBIT    3000H.0
TBIT_SYM2       TBIT    3000H.1
        DSEG AT 0E000H
DATA_SYM1:
        DS      2
        DSEG    AT      1:9000H
DATA_SYM2:
        DS      1
        BSEG AT 0E100H.0
BIT_SYM1:
        DBIT    1
        BSEG    AT      1:0A000H.0
BIT_SYM2:
        DBIT    1
        NVSEG
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
        NVBSEG
EBIT_SYM1:
        DBIT    1
EBIT_SYM2:
        DBIT    1
        PUBLIC  NUM CODE_SYM1 TBIT_SYM1 TBIT_SYM2 DATA_SYM1 DATA_SYM2
        PUBLIC  EDATA_SYM1 EDATA_SYM2 EBIT_SYM1 EBIT_SYM2 BIT_SYM1
        PUBLIC  BIT_SYM1 BIT_SYM2
       END
```

Next, use the object converter to convert TEST.OBJ to both types of HEX file.

```
OHU8 TEST.OBJ   /D;
```

This command line creates two files: the Intel HEX file TEST.HEX, which includes the debugging information, and the NVRAM code file TEST.XNV.

Intel HEX format program code file with debugging information: TEST.HEX

```
0 TBIT_SYM1 18000H 0 TB
0 TBIT_SYM2 18001H 0 TB
0 EDATA_SYM1 8000H 0 ND
0 EDATA_SYM2 8010H 0 ND
0 CODE_SYM1 0H 0 T
0 EBIT_SYM1 40000H 0 NB
0 EBIT_SYM2 40001H 0 NB
0 NUM 100H 0 N
0 BIT_SYM1 70800H 0 B
0 BIT_SYM2 50000H 1 B
0 DATA_SYM1 E000H 0 D
0 DATA_SYM2 9000H 1 D
 $
:100000000001020304050607080 90A0B0C0D0E0F78
:020000021000EC
:10000000000001000200030004000 5000600070 0D4
:10001000080009000A000B000C000D000E000F0084
:00000001FF
```

Intel HEX format NVRAM code file: TEST.XNV

```
:10800000000102030405060708090A0B0C0D0E0FF8
:10801000000001000200030004000 5000600070044
:10802000080009000A000B000C000D000E000F00F4
:00000001FF
```

```
OHU8  TEST.OBJ  /S  /D;
```

This command line creates three files: the S2 files TEST.S and TEST.SNV plus the public symbol file TEST.SYM.

Motorola S2 format program code file: TEST.S

```
S00F00004F4B492C30312C30302C55383B
S2140000000001020304050607080 90A0B0C0D0E0F73
S2140100000000010002000300040005000 6000700CE
S214010010080009000A000B000C000D000E000F007E
S804000000FB
```

Motorola S2 format NVRAM code file: TEST.SNV

```
S00F00004F4B492C30312C30302C55383B
S2140080000001020304050607080 90A0B0C0D0E0FF3
S2140080100000010002000300040005000 60007003F
S214008020080009000A000B000C000D000E000F00EF
S804000000FB
```

Debugging information file (public symbols only): TEST.SYM

```
0 TBIT_SYM1 18000H 0 TB
0 TBIT_SYM2 18001H 0 TB
0 EDATA_SYM1 8000H 0 ND
0 EDATA_SYM2 8010H 0 ND
0 CODE_SYM1 0H 0 T
0 EBIT_SYM1 40000H 0 NB
0 EBIT_SYM2 40001H 0 NB
0 NUM 100H 0 N
0 BIT_SYM1 70800H 0 B
0 BIT_SYM2 50000H 1 B
0 DATA_SYM1 E000H 0 D
0 DATA_SYM2 9000H 1 D
 $
```

9.8 Temporary Files

Conversions with the object converter use up to three temporary files: $_$, $__$, and $___$. The number depends on the output file format.

The object converter stores its output to temporary files as it reads in data from the input file. If the conversion completes successfully, it then writes the data from the temporary files to the output files. Just before exiting, it deletes the temporary files.

These temporary files are created in the current directory, so do not use files with these names in the current directory.

9.9 Error Messages

The only errors reported are all fatal ones. The object converter then aborts the conversion and skips the output files.

9.9.1 Error Message Formats

There are separate message formats for two error types: during conversion and at other times. *

1. during conversion

   Error *error_code*:*error_message*

   File Offset *hhhhhhhhH(dddddddd)*

2. at other times

   Error : *error_code*: *error_message*

The following describes the notations used in the above formats.

Manual Notation	Screen Display
hhhhhhhhH	File offset for error in hexadecimal
(*dddddddd*)	File offset for error in decimal
error_code	Error code
error_message	Descriptive message for error

The following lists object converter error messages by error code. Following the message text is a description.

001 Bad syntax on command line

There is an error in a command line specification.

002 Unable to open OHU8 temporary file

The object converter cannot create a temporary file.

003 Checksum failure

The object converter has detected a checksum error in the current record. The absolute object file appears to be damaged, so try rebuilding it.

004 Command option duplicated

The same option appears more than once on the command line.

006 File not absolute

The input file contains relocatable components. If it is an .OBJ file produced by the assembler, try converting it to an .ABS file with the linker.

007 File offset error

The object converter was unable to determine the file offset for a record read from the input file.

008 File read error

There was an error reading the input file.

009 File remove error

There was an error deleting a temporary file.

010 Illegal command option

The command line contains an unknown option.

011 Input file not specified

The command line specifies something other than a file name in the field for specifying the input file name.

012 Insufficient disk space

File output failed because the disk was full.

013 Insufficient memory

There is not enough memory to continue.

014 Invalid family id

The input file is not for the U8 architecture. The object converter supports only absolute object files created by RASU8 or RLU8.

015 Invalid object module

There is a problem with the input file.

016 Invalid record type

The input file contains a record type not recognized by the object converter. Make sure that the assembler RASU8, linker RUL8, and object converter OHU8 are all from the same package.

017 Invalid segment type

A debugging information symbol in the input file contains an unsupported segment type.

018 Invalid target

The input file was created as a general-purpose module.

019 Invalid version number

The version number for the software creating the object file is invalid.

020 I/O error

There was an error closing a file.

021 Unable to use Intel HEX format, so specify /S

The input file contains object code for an address beyond 0FH:0FFFFH, the maximum for the Intel HEX format. The only output format possible for this file is thus the Motorola S2 format (/S).

023 Bad constant, illegal character

The /R command line option address range specification contains an invalid character.

024 Bad constant, out of range

The /R command line option address range specification exceeds the 0FH:0FFFFH limit.

025 Invalid conversion range

There is a problem with the range specified with the /R command line option.

10 Using Overlays

10.1 Overview

The overlay function divides a portion of the program into overlay units, stores them in individual data memory space regions, and loads them as necessary one at a time into an overlay region for execution.

The following illustrates overlay operation.

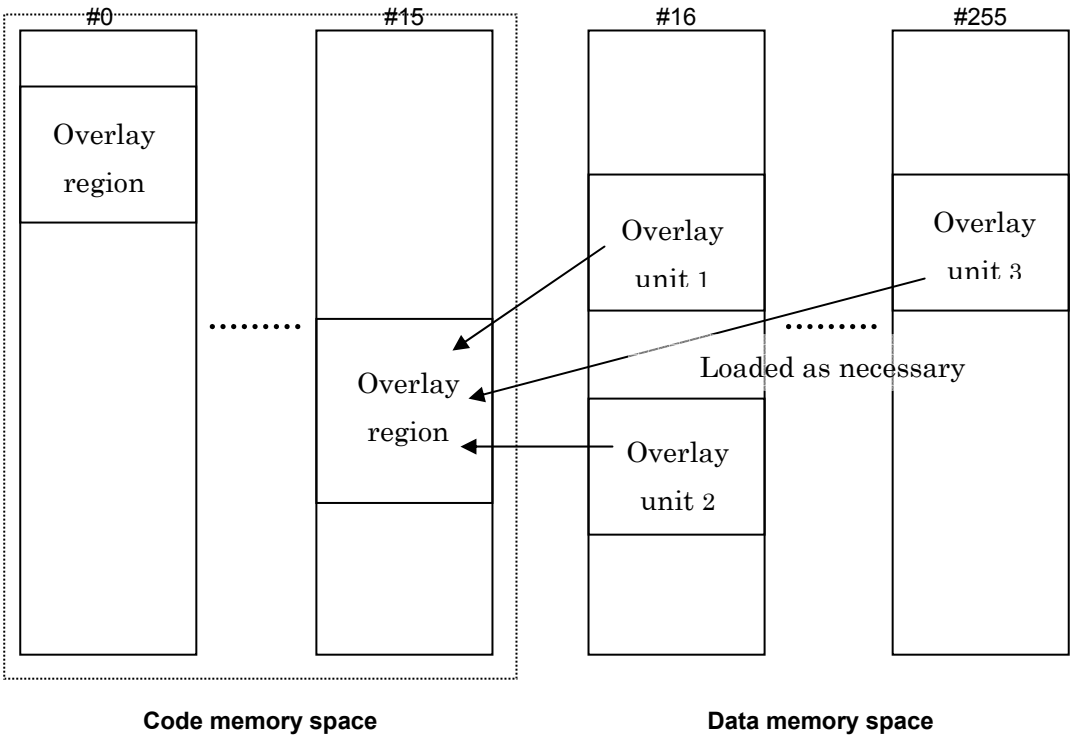


Figure 10.1 Overlay Operation

The overlay region be in any physical segment in the code memory space (#0 to #15). The memory region must, of course, be writable (RAM or nonvolatile memory).

The overlay units consist of one or more CODE segments each. They go in the data memory space. The Figure above has them all in physical segments above #15, but they can go anywhere in the data memory space starting at physical segment #1. The only restriction is that the memory region must contain ROM or nonvolatile memory.

To actually execute the object code making up an overlay unit, the user application program must first load it into the overlay region. The programmer is responsible for writing the loader, the code for the task.

10.1.1 Actual Assigned Address and Execution Address

Overlay units differ from normal program code in that the object code making up the overlay unit is stored in the data memory space at the actual assigned address, but runs in code memory at the execution address. Normal program code uses the same address for both purposes.

The address values for function entry point labels and other symbols referenced during program execution are therefore calculated using execution addresses; the ones for allocating the object to memory, actual assigned addresses.

```
      CSEG AT 1:8000H OVL 16:1000H
Overlay_func:
      MOV     ER0,    #00H
      .
      .
      .
      RT
```

The above example defines an absolute CODE segment for use as an overlay. The label *Overlay_func* has 1:8000H as its execution address, but the actual object code starts at offset 1000H in physical segment #16.

10.1.2 Limits on Overlay Regions

The overlay region must be in a memory region that is both executable (code memory space) and writable (RAM or nonvolatile memory).

The following lists the code memory space RAM and nonvolatile memory candidates available.

Memory Model	Available for Overlay Regions
SMALL	Code memory space RAM or nonvolatile memory in physical segment #0
LARGE	Code memory space RAM or nonvolatile memory in physical segments #0 to #15

10.1.3 Regions for Overlay Units

An overlay unit consists of one or more CODE segments. Unlike normal CODE segments, which must be assigned to the code memory space, these CODE segments can go anywhere because they are always loaded into and then run from a specific overlay region.

The following summarizes the differences between the ROM and nonvolatile memory regions available for assigning normal and overlay CODE segments.

CODE Segment Type	Regions Available
Normal	Code memory space ROM or nonvolatile memory in physical segment #0 (SMALL memory model) or in physical segments #0 to #15 (LARGE memory model)
Overlay	ROM or nonvolatile memory in physical segment #0 in the code memory space or in physical segments #1 to #255 in the data memory space

In the absence of any /CODE command line options specifying addresses for relocatable CODE segments, all ROM and nonvolatile memory regions in physical segments #1 to #255 are assigned to the data memory space.

Normally, a relocatable CODE segment can be assigned to nonvolatile memory only if its segment symbol definition specifies the special region attribute NVRAM. This restriction does not apply, however, if a /CODE command line option specifies an absolute address.

10.2 Creating Overlay Units

There are two ways to create overlay units: from absolute segments and from relocatable ones.

10.2.1 From Absolute Segments

The following is the syntax for defining absolute segments for use in overlay units.

■ Syntax ■

CSEG [#*pseg_addr*] AT *overlay_address* OVL *allocation_address*

■ Description ■

The OVL specifier in the definition defines the absolute segment for use in an overlay unit. This specifier is only available in CSEG directives defining absolute CODE segments. The first two fields together specify the physical segment address and offset for the execution address; the address field at the end, the actual assigned address.

If the OVL specifier is present, the execution address specification in the first two fields must include a physical segment address.

The DSEG or NVSEG directive defining the overlay region must reserve enough space to hold the largest overlay unit in the group.

```
; Start CODE segment for overlay unit
    CSEG AT 1:8000H OVL 10H:1000H
Overlay_func:
    MOV     ER0,    #00H
    .
    .
    .
    RT

; Reserve overlay region
    DSEG AT 1:8000H
Overlay_area:
    DS      1000H
```

The above example specifies overlay execution at 1:8000H for object code starting at 10H:1000H. The DSEG directive reserves space for this overlay region.

Note that the program is responsible for loading the object code into the overlay region before executing it.

10.2.2 From Relocatable Segments

The following is the RLU8 command line option syntax for assigning relocatable segments to overlay units.

■ Syntax ■

/OVERLAY(*area_name*, *start_address*, *end_address*){[*overlay_unit* [*overlay_unit* …]]}

■ Description ■

The /OVERLAY command line option specifies a name and address range for the overlay region and then lists the overlay units and their segments assigned to that overlay region.

■ *overlay_unit* Syntax ■

UNIT(*segment* [*segment* …])

An overlay unit specification consists of the keyword UNIT followed by, in parentheses and delimited with spaces, a list of segments making up the overlay unit.

Note that the linker assigns the segments to the overlay region in the order specified.

■ Example ■

```
# Response file (LINK.RES) for overlay specifications
/OVERLAY(OVL1, 1:8000H, 1:9FFFH)
{
    UNIT(segA segB)      // Overlay unit 1
    UNIT(segC segD)      // Overlay unit 2
    UNIT(segE segF segG) // Overlay unit 3
}
/CODE(10H:0H segA segB segC segD segE segF segG)
# EOF
```

This example uses an RLU8 response file LINK.RES to hold /OVERLAY specifications for overlay region OVL1 with the address range 1:8000H to 1:9FFFH and three overlay units.

The /CODE command line option specifies a base address for the actual assigned addresses for the overlay units and the relocatable CODE segments in them.

To use this response file, type the following.

```
RLU8 OVLSAMPLE @LINK.RES ;
```

The following illustrates how the segments in the overlay units fit into the overlay region for execution.

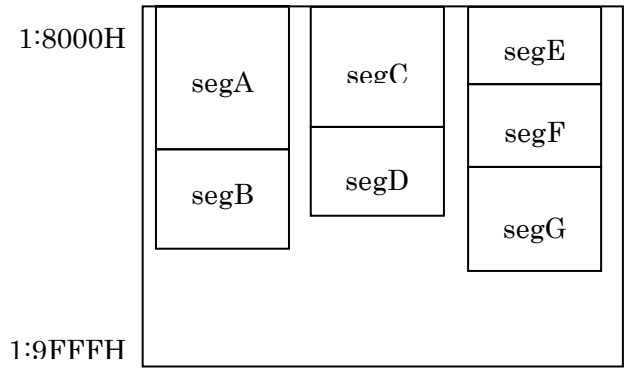


Figure 10.2 Loading Overlay Units into Overlay Region

The RLU8 map file provides the following output detailing the addresses assigned to the individual segments in overlay region OVL1.

```
Link Map - Overlay Area "OVL1" (01:8000 - 01:9FFF)
```

Overlay region OVL1

```
      Type      Start       Stop        Size                Name
     ------------------------------------------------------
      S CODE    01:8000     01:89FF     0A00(2560)          segA
      S CODE    01:8A00     01:9133     0734(1844)          segB
 ---
      S CODE    01:8000     01:8965     0966(2406)          segC
      S CODE    01:8966     01:8F9D     0638(1592)          segD
 ---
      S CODE    01:8000     01:8595     0596(1430)          segE
      S CODE    01:8596     01:8D2B     0796(1942)          segF
      S CODE    01:8D2C     01:9583     0858(2136)          segG
```

This listing separates the groups of segments making up each overlay unit with three hyphens.

Note how the address ranges use execution addresses.

The actual assigned addresses for the segments appear in a separate listing in the map file.

```
      Type      Start       Stop        Size                Name
     ------------------------------------------------------------
>GAP<           01:0000.0   01:7FFF.7   8000.0(32768.0)     (RAM)
      Q OVRLAY  01:8000     01:9FFF     2000(8192)          (OVL1)
 ---
      S CODE    10:0000     10:09FF     0A00(2560)          segA *
      S CODE    10:0A00     10:1133     0734(1844)          segB *
      S CODE    10:1134     10:1A99     0966(2406)          segC *
      S CODE    10:1A9A     10:20D1     0638(1592)          segD *
      S CODE    10:20D2     10:2667     0596(1430)          segE *
      S CODE    10:2668     10:2DFD     0796(1942)          segF *
      S CODE    10:2DFE     10:3655     0858(2136)          segG *
```

This example is an extract from the segment assignment information for physical segments #1 and higher.

The overlay region OVL1 defined with the /OVERLAY command line option appears in the map file as a pseudo-segment. The /CODE command line option specifies assignment of relocatable segments segA through segG to addresses at or above offset 0 in physical segment #16.

An asterisk (*) to the right of a segment name indicates an overlay segment.

10.3 Overlay Loader

The overlay loader must be provided by the programmer. The simplest loader simply copies the desired object code from the actual assigned address to the overlay region.

Doing so requires at least two addresses: source and destination. Normal symbol references use the execution address, so the assembler provides the following operators for accessing the actual assigned address.

Operator	Description
OVL_ADDRESS *symbol*	Full actual assigned address for segment *symbol*
OVL_SEG *symbol*	Physical segment address portion of the actual assigned address for segment *symbol*
OVL_OFFSET *symbol*	Offset portion of the actual assigned address for segment *symbol*

In the interest of maximizing code reusability, the following overlay loader takes such particulars as copy source and destination addresses from a table. The following shows the relevant portion for the overlay CODE segment Unit1Seg.

```
; Overlay loader parameter table
UnitAdrTable    segment table word any
        rseg    UnitAdrTable
Unit1p:
        dw      offset      Unit1Seg    ; Offset portion of execution
                                        ; address
        dw      ovl_offset  Unit1Seg    ; Offset portion of actual
                                        ; assigned address
        dw      size        Unit1Seg    ; Segment size
        db      seg         Unit1Seg    ; Physical segment address
                                        ; portion of execution
                                        ; address
        db      ovl_seg     Unit1Seg    ; Physical segment address
                                        ; portion of actual
                                        ; assigned address
```

This example uses the overlay operators OVL_OFFSET and OVL_SEG to obtain the actual assigned address for the overlay CODE segment Unit1Seg.

The following is sample code for copying object code between the source and destination addresses from this table.

```
; Sample overlay loader
; OvlsegLoad accepts the following parameters
; ER0: Offset portion of overlay loader parameter table address
; R2: Physical segment address portion of overlay loader parameter
; table address
;
OverlayLoader   segment code  any
        rseg    OverlayLoader
OvlsegLoad:
        push    qr0
        push    qr8
        l       er4,    r2:[er0]    ; Get offset portion of execution
                                    ; address (destination)
        l       er6,    r2:2[er0]   ; Get offset portion of actual
                                    ; assigned address (source)
        l       er8,    r2:4[er0]   ; Get overlay segment
        l       er10,   r2:6[er0]   ; Get physical segment addresses
                                    ; for source and destination
                                    ; addresses
copy_loop:
        cmp     r8,     #0
        cmpc    r9,     #0
        beq     copy_end
        l       er12,   r11:[er6]
        st      er12,   r10:[er4]
        add     er4,    #2
        add     er6,    #2
        add     er8,    #-2
        bal     copy_loop
copy_end:
        pop     qr8
        pop     qr0
        rt
public  OvlsegLoad
```

The following is source code for actually using the above code to load an overlay.

```
; Load overlay Unit1Seg
        mov     r0,     #byte1 offset Unit1p
        mov     r1,     #byte2 offset Unit1p
        mov     r2,     #seg Unit1p
        bl      OvlsegLoad
```

This completes the loading of the overlay. Once loading is complete, the overlay runs the same as normal object code.

Note, however, that this sample overlay unit consists of a single segment. The program must repeat the loading process for all other CODE segments in the overlay unit as well. For this reason, it is obviously better to keep the number of CODE segments making up an overlay unit to a minimum. If these CODE segments are in multiple files, one technique is to make them into partial segments by using the same segment symbol for the entire set.

11 Absolute Print File Generation

11.1 Overview

The assembler's absolute print file function generates a print file that does not contain any indeterminate machine code or addresses—in other words, with only absolute addresses.

Splitting a program into multiple modules, using relocatable segments, and other actions render the assembler incapable of finalizing all machine code and addresses in normal print files. Debugging with such files therefore requires flipping back and forth to such reference materials as the symbol table in the map file from the linker.

The MACU8 assembler package eliminates this tedium by feeding back such information from the linker, so that a second assembler run can replace the indeterminate portions with the proper final values.

11.2 Generating an Absolute Print File

This section describes the procedure for creating an absolute listing file using a sample program consisting of three source code files: FOO1.ASM, FOO2.ASM, and FOO3.ASM.

The first step is to assemble the three source code files normally.

```
RASU8 FOO1
RASU8 FOO2
RASU8 FOO3
```

The second step is to link the three object files with the /A command line option.

```
RLU8 FOO1 FOO2 FOO3 /A;
```

This option causes the linker to generate an extra file with the name FOO1.ABL. This ABL file lists such things as absolute addresses for symbols and final machine code.

The third step is to reassemble the three source code files, this time with the option /AFOO1. This option takes as its argument the name of an ABL file created by the linker. The file extension (.ABL) is optional.

```
RASU8 FOO1 /AFOO1
RASU8 FOO2 /AFOO1
RASU8 FOO3 /AFOO1
```

Reassembly produces three files with the file extension .APR: FOO1.APR, FOO2.APR, and FOO3.APR. These are absolute print files.

The following Figure shows the entire process schematically.

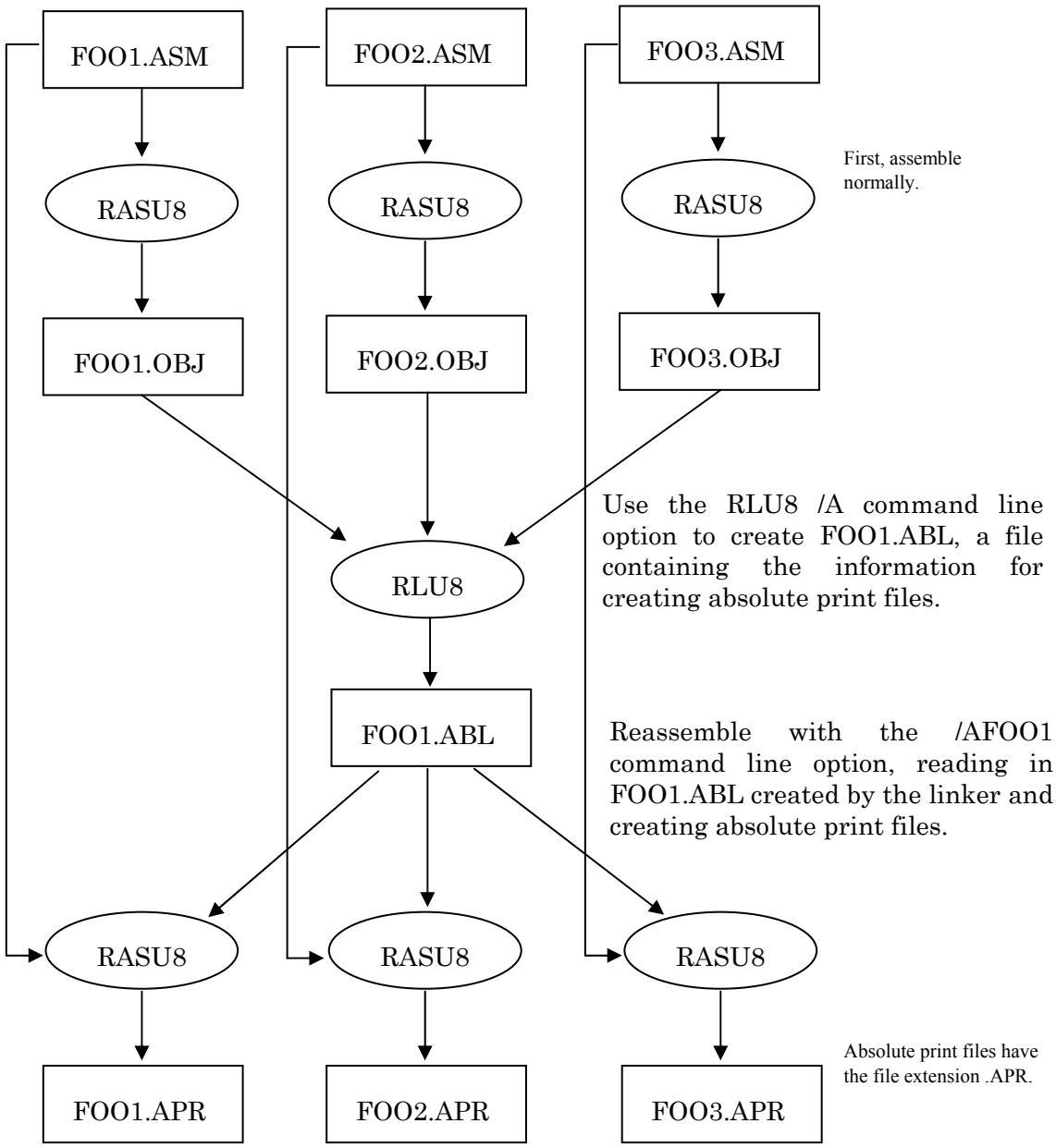


Figure 11.1 Generating an Absolute Print File

11.3 Linker /A Command Line Option

Creating absolute print files requires specifying the /A command line option to the linker. Specifying this option causes the linker to generate an ABL file, an extra file listing such things as absolute addresses for symbols and final machine code.

The following is the syntax for the linker's /A command line option.

■ Syntax ■

/A[(*abl_file*)]

The argument specifies a name for the ABL file. This name defaults to the absolute object (.ABS) file name with the file extension changed to .ABL. The file extension defaults to .ABL.

■ Example ■

```
RLU8 FILE1 FILE2 FILE3 /A;
```

Here the ABL file name defaults to FILE1.ABL.

■ Example ■

```
RLU8 FILE1 FILE2 FILE3 /A(PRNDATA);
```

Omitting the file extension produces the default, .ABL.

■ Example ■

```
RLU8 FILE1 FILE2 FILE3 /A(PRNDATA.DAT);
```

A full name specification (PRNDATA.DAT) is used as is.

11.4 Assembler /A Command Line Option

To reassemble files and generate absolute print files, add the /A command line option.

The following is the syntax for the assembler's /A command line option.

■ Syntax ■

/A[*abl_file*]

The argument specifies an ABL file created by the linker. There must not be any spaces between it and the option name. The ABL file name defaults to the source code file name with the file extension changed to .ABL. The file extension defaults to .ABL.

Specifying the /A command line option causes the assembler to read in the ABL file and generate an absolute print file.

The default name for the output file is the source code file name with the file extension changed to .APR. The /PR command line option allows the programmer to change the name from the default.

Reassembly features the following differences from normal assembly.

(1) It does not create an object file. The /O command line option and OBJ directives are ignored.

(2) It does not process C source level debugging information. The /SD command line option is ignored.

(3) It does not create an external symbol declaration file. The /X command line option is ignored.

(4) It always generates an absolute print file—even in the presence of the /NPR command line option or NOPRN directives.

Note that, while certain command line options and directives are ignored, that does not mean that including them represents errors.

The following is a list of command line options that must remain unchanged between the two assembler runs.

Option	Function
/MS and /ML	Memory model. We recommend using the MODEL directive whenever possible.
/DN and /DF	Data model. We recommend using the MODEL directive whenever possible.
/CD and /NCD	Case sensitivity for symbols
/I*include_path*	Include file path

The simplest way to enforce this is to use exactly the same command line options both times and just add the /A command line option for the second. Suppose that the first assembler run uses the following command line options, for example.

```
RASU8 FOO1 /D /R /PW120 /NPR /T4 /IHEADER
```

The second run should just add the /A command line option.

```
RASU8 FOO1 /D /R /PW120 /NPR /T4 /IHEADER /AFOO1
```

11.5 Reassembly Errors

Adding the /A command line option and reassembling a source code file that produced no error messages during normal assembly sometimes produces fresh error and warning messages. The reason is that the assembler is now able to apply full error checking instead of having to skip operands because they are based on indeterminate addresses.

Consider the following example.

```
EXTRN DATA:DATA_TBL
    L       ER0,    DATA_TBL
```

Just because normal assembly does not produce an error message does not necessarily mean that the above code is correct. Only that the assembler does not know the address for DATA_TBL, so is unable to check it.

Suppose, for example, that another source code file specifies 8001H as the address for DATA_TBL. Accessing this odd-numbered address represents a word boundary error. The linker alerts the programmer to this problem with the following warning message, which says that the instruction triggering the warning is apparently at the address 200H.

```
Warning W006: Cannot access high byte, 0200/(absolute)/foo1
```

This warning message flags the problem, but leaves it up to the programmer to find the location in the actual source program corresponding to the address provided. The task only gets harder as the program grows larger.

The absolute print file function can help here. Reassembling the specified source code file with the /A command line option produces a similar warning message.

```
foo1.asm(206):206: Warning 28 : cannot access high byte
```

Unlike the linker version, however, this one gives the programmer a precise location, the line number.

As the above illustrates, one objective served by absolute print files is pinpointing the sources of linker warnings related to addressing in the source code files.

Unfortunately, this usefulness does not extend to all linker error messages. Reassembly only helps with linker warning messages. Reassembling a source code file producing any other type of error message does not produce a proper absolute print file. Sometimes it even produces a fatal error message.

11.6 Example of Absolute Print File

The format of absolute print file is the same as that of the print file output at the assembling time. For the format of the print file, see Section “6.7.1 Assembly Listing”.

In addition, the functions or tables which are not referred to are excepted from the candidate for allocation. The location field is displayed by an asterisk (**:****) to such function or table.

The output example of an absolute print file is shown below. For explanation, the left-hand side of an absolute print file is numbered.

```
    ## Loc. Object                    Line   Source Statements
                                        :
                                        :
      --------------------------       14       rseg $$ref_func$tapr
                                       15
 ┌─ 00:00A8                            16   _ref_func   :
 │                                     17
 │                                     18   ;;{
 │                                     19
 │                                     20   ;;    var += 10;
(1) 00:00A8 12-90 FE-E7                21       l   er0,    NEAR _var
 │  00:00AC 8A E0                      22       add er0,    #10
 │  00:00AE 13-90 FE-E7                23       st  er0,    NEAR _var
 │                                     24
 │                                     25   ;;}
 └─ 00:00B2 1F-FE                      26       rt
                                       27
                                       28
      --------------------------       29       rseg $$noref_func$tapr
                                       30
 ┌─ **:****                            31   _noref_func :
 │                                     32
 │                                     33   ;;{
 │                                     34
 │                                     35   ;;    var -= 10;
(2) **:**** 12-90 FE-E7                36       l   er0,    NEAR _var
 │  **:**** F6 E0                      37       add er0,    #-10
 │  **:**** 13-90 FE-E7                38       st  er0,    NEAR _var
 │                                     39
 │                                     40   ;;}
 └─ **:**** 1F-FE                      41       rt
                                       42
                                        :
                                        :
```

The above (1) in the example shows the function which was linked. And the above (2) in the example shows the function which was not linked.

11.7 Fatal Error 11

Sometimes reassembly aborts with the following fatal error message.

```
MACRO.asm 29 : Fatal Error 11: illegal reading binary file :
ABL file : error_message
```

Almost always the reason for this error message is a problem with the contents of the ABL file. Start by checking the following.

(1) Did the first assembly run produce any error messages? (Warning messages can be ignored.)

(2) Do the specifications for memory and data models (/MS, /ML, /DN, and /DF command line options and MODEL directives), symbol case sensitivity (/CD and /NCD command line options), and include path (/Iinclude_path command line option) match completely for both runs?

(3) Did the linker produce any messages other than warning messages?

(4) Did you specify the linker's /A command line option?

If the source program passes all the above checks, yet this error message persists, contact your nearest LAPIS Semiconductor sales office.

Appendices

Appendix A. Directive List

The following is a list of directives.

Directive	**Syntax** **Function**
TYPE	TYPE (*dcl_name*) Specifies the base name of the DCL file for the target microcontroller.
ROMWINDOW	ROMWINDOW *base_address*, *end_address* Specifies the address range for the ROM window region.
NOROMWIN	NOROMWIN Specifies that the program does not use the ROM window function.
MODEL	MODEL *memory_model* [, *data_model*] or MODEL *data_model* [, *memory_model*] Specifies the data and memory models.
END	END Indicates the end of current file.
EQU SET	*symbol* EQU *simple_expression* *symbol* SET *simple_expression* Defines a general-purpose local symbol.
CODE	*symbol* CODE *simple_expression* Defines a CODE local symbol.
TABLE	*symbol* TABLE *simple_expression* Defines a TABLE local symbol.
TBIT	*symbol* TBIT *simple_expression* Defines a TBIT local symbol.
DATA	*symbol* DATA *simple_expression* Defines a DATA local symbol.
BIT	*symbol* BIT *simple_expression* Defines a BIT local symbol.

Directive	**Syntax** **Function**
NVDATA	*symbol* NVDATA *simple_expression* Defines an NVDATA local symbol.
NVBIT	*symbol* NVBIT *simple_expression* Defines an NVBIT local symbol.
SEGMENT	*segment_symbol* SEGMENT *segment_type* [*boundary_attr*] [*seg_attr*] [*relocation_attr*] Defines a relocatable segment.
STACKSEG	STACKSEG *stack_size* Defines the stack segment.
CSEG	CSEG [#*pseg_addr*][AT *start_address*] Declares the start of an absolute CODE segment.
TSEG	TSEG [#*pseg_addr*][AT *start_address*] Declares the start of an absolute TABLE segment.
DSEG	DSEG [#*pseg_addr*][AT *start_address*] Declares the start of an absolute DATA segment.
BSEG	BSEG [#*pseg_addr*][AT *start_address*] Declares the start of an absolute BIT segment.
NVSEG	NVSEG [#*pseg_addr*][AT *start_address*] Declares the start of an absolute NVDATA segment.
NVBSEG	NVBSEG [#*pseg_addr*][AT *start_address*] Declares the start of an absolute NVBIT segment.
RSEG	RSEG *segment_symbol* Starts a relocatable segment.
ORG	ORG *address* Resets the location counter for the current logical segment.
DS	[*label* :] DS *size* Reserves the specified number of bytes in the current logical segment.

Directive	**Syntax** **Function**
DBIT	[*label* :] DBIT *size* Reserves the specified number of bits in the current logical segment.
ALIGN	ALIGN Makes the current location even.
DB	[*label* :] DB { *expression* | *string_constant* | *duplicate_expression* } [, { *expression* | *string_constant* | *duplicate_expression* }] … Initializes CODE memory one byte at a time.
DW	[*label* :] DW { *expression* | *duplicate_expression* } [, { *expression* | *duplicate_expression* }] … Initializes CODE memory one word at a time.
GJMP	[*label* :] GJMP *symbol* Converts to the optimal branch instruction.
GB*cond*	[*label* :] GB*cond symbol* Converts to the optimal conditional branch instruction.
EXTRN	EXTRN *usage_type* [*attribute*] : *symbol* [*symbol* …] [*usage_type* [*attribute*] : *symbol* [*symbol* …]] … Declares an external symbol.
PUBLIC	PUBLIC *symbol* [*symbol* …] Makes a symbol public.
COMM	*communal_symbol* COMM *segment_type size* [*relocation_attr*] Defines a communal symbol.
INCLUDE	INCLUDE (*include_file*) Reads in a file.
DEFINE	DEFINE *symbol* ”*macro_body*” Assigns a text string to a macro symbol.

Directive	**Syntax** **Function**

IF
IFDEF
IFNDEF
ELSE
ENDIF
	IF*xxx conditional_operand* (IFxxx is IF, FDEF, or IFNDEF)
	true_conditional_body
	[ELSE
	false_conditional_body]
	ENDIF
	Conditionally assembles block.

CFILE	CFILE *file_id total_line* "*filename*" Gives the name and other information on a C source code file.
CFUNCTION	CFUNCTION *fn_id* Indicates the start of a C function.
CFUNCTIONEND	CFUNCTIONEND *fn_id* Indicates the end of a C function.
CARGUMENT	CARGUMENT *attrib size offset* "*variable_name*" *hierarchy* Gives information on a C function argument.
CBLOCK	CBLOCK *fn_id block_id c_source_line* Indicates the start of a C block.
CBLOCKEND	CBLOCKEND *fn_id block_id c_source_line* Indicates the end of a C block.
CLABEL	CLABEL *label_no* "*label_name*" Links labels between C and assembly language.
CLINE	CLINE *line_attr line_no start_column end_column* Gives C source code file line number information.
CLINEA	CLINEA *file_id line_attr line_no start_column end_column* Gives C source code file line number information.
CGLOBAL	CGLOBAL *usg_typ attrib size* "*variable_name*" *hierarchy* Gives the name and other information on a global variable defined in the C source program.

Directive	**Syntax** **Function**
CSGLOBAL	CSGLOBAL *usg_typ attrib size* "*variable_name*" *hierarchy* Gives the name and other information on a static global variable defined in the C source program.
CLOCAL	CLOCAL *attrib size offset block_id* "*variable_name*" *hierarchy* Gives the name and other information on a local variable defined in the C source program.
CSLOCAL	CSLOCAL *attrib size alias_no block_id* "*variable_name*" *hierarchy* Gives the name and other information on a static local variable defined in the C source program.
CSTRUCTTAG	CSTRUCTTAG *fn_id block_id st_id total_mem total_size* "*tag_name*" Gives the name and other information on a structure defined in the C source program.
CSTRUCTMEM	CSTRUCTMEM *attrib size offset* "*member_name*" *hierarchy* Gives the name and other information on a structure member defined in the C source program.
CUNIONTAG	CUNIONTAG *fn_id block_id un_id total_mem total_size* "*tag_name*" Gives the name and other information on a union defined in the C source program.
CUNIONMEM	CUNIONMEM *attrib size* "*member_name*" *hierarchy* Gives the name and other information on a union member defined in the C source program.
CENUMTAG	CENUMTAG *fn_id block_id emu_id total_mem* "*tag_name*" Gives the name and other information on an enumeration defined in the C source program.
CENUMMEM	CENUMMEM *value* "*member_name*" Gives the name and other information on an enumeration member defined in the C source program.
CTYPEDEF	CTYPEDEF *fn_id block_id attrib* "*type_name*" *hierarchy* Gives the name and other information on a user defined type defined with a typedef in the C source program.

Directive	Syntax Function
FASTFLOAT	FASTFLOAT Specifies to the linker the faster emulation library for floating point arithmetic.
OBJ	OBJ [(*object_file*)] Generates an object file.
NOOBJ	NOOBJ Skips the object file.
PRN	PRN [(*print_file*)] Generates a print file.
NOPRN	NOPRN Skips the print file.
ERR	ERR [(*error_file*)] Generates an error file.
NOERR	NOERR Skips the error file.
DEBUG	DEBUG Includes debugging information in the object file.
NODEBUG	NODEBUG Skips debugging information in the object file.
LIST	LIST Enables assembly listing output to the print file.
NOLIST	NOLIST Disables assembly listing output to the print file.
SYM	SYM Enables symbol table output to the print file.
NOSYM	NOSYM Disables symbol table output to the print file.

Directive	**Syntax** **Function**
REF	REF Enables cross-reference listing output to the print file.
NOREF	NOREF Disables cross-reference listing output to the print file.
PAGE	PAGE PAGE [*page_length*][, *page_width*] 1. Forces a page break in the print file. 2. Specifies the print file page length in lines and the page width in columns.
NOPAGE	NOPAGE Removes page length and page width limits in the print file.
DATE	DATE ”*charcter_string*” Specifies a text string for the print file's date field.
TITLE	TITLE ”*character_string*” Specifies a text string for the print file's title field.
TAB	TAB [*tab_width*] specifies the tab width for formatting the print file output.
NOFAR	NOFAR Limits data memory spaces to only physical segment #0.

Appendix B. Reserved Word List

The following is an alphabetical list of reserved words with their applications. If a reserved word has more than one application, they are separated with slashes.

	Reserved Word	Application	Notes
A	ADD	Basic instruction	
	ADDC	Basic instruction	
	AL	Condition for BC instruction	
	ALIGN	Directive	
	AND	Basic instruction	
B	B	Basic instruction	
	BAL	Basic instruction	
	BC	Basic instruction	
	BCY	Basic instruction	
	BEQ	Basic instruction	
	BGE	Basic instruction	
	BGES	Basic instruction	
	BGT	Basic instruction	
	BGTS	Basic instruction	
	BIT	Directive / directive operand	
	BL	Basic instruction	
	BLE	Basic instruction	
	BLES	Basic instruction	
	BLT	Basic instruction	
	BLTS	Basic instruction	
	BNC	Basic instruction	
	BNE	Basic instruction	
	BNS	Basic instruction	
	BNV	Basic instruction	

	Reserved Word	Application	Notes
B	BNZ	Basic instruction	
	BOV	Basic instruction	
	BP	Register name	
	BPOS	Operator	
	BPS	Basic instruction	
	BRK	Basic instruction	
	BSEG	Directive	
	BYTE1	Operator	
	BYTE2	Operator	
	BYTE3	Operator	
	BYTE4	Operator	
	BZ	Basic instruction	
C	CARGUMENT	Directive	
	CBLOCK	Directive	
	CBLOCKEND	Directive	
	CENUMMEM	Directive	
	CENUMTAG	Directive	
	CER0	Register name	
	CER2	Register name	
	CER4	Register name	
	CER6	Register name	
	CER8	Register name	
	CER10	Register name	
	CER12	Register name	
	CER14	Register name	
	CFILE	Directive	
	CFUNCTION	Directive	

	Reserved Word	Application	Notes
C	CFUNCTIONEND	Directive	
	CGLOBAL	Directive	
	CLABEL	Directive	
	CLINE	Directive	
	CLINEA	Directive	
	CLOCAL	Directive	
	CMP	Basic instruction	
	CMPC	Basic instruction	
	CODE	Directive / directive operand	
	COMM	Directive	
	CPL	Basic instruction	Not allowed
	CPLC	Basic instruction	
	CQR0	Register name	
	CQR8	Register name	
	CR0	Register name	
	CR1	Register name	
	CR2	Register name	
	CR3	Register name	
	CR4	Register name	
	CR5	Register name	
	CR6	Register name	
	CR7	Register name	
	CR8	Register name	
	CR9	Register name	
	CR10	Register name	
	CR11	Register name	
	CR12	Register name	

	Reserved Word	Application	Notes
C	CR13	Register name	
	CR14	Register name	
	CR15	Register name	
	CSEG	Directive	
	CSGLOBAL	Directive	
	CSLOCAL	Directive	
	CSTRUCTMEM	Directive	
	CSTRUCTTAG	Directive	
	CTYPEDEF	Directive	
	CUNIONMEM	Directive	
	CUNIONTAG	Directive	
	CXR0	Register name	
	CXR4	Register name	
	CXR8	Register name	
	CXR12	Register name	
	CY	Condition for BC instruction	
D	DAA	Basic instruction	
	DAS	Basic instruction	
	DATA	Directive / directive operand	
	DATE	Directive	
	DB	Directive	
	DBIT	Directive	
	DEBUG	Directive	
	DEC	Basic instruction	
	DEFINE	Directive	
	DI	Basic instruction	
	DIV	Basic instruction	

	Reserved Word	Application	Notes
D	DS	Directive	
	DSEG	Directive	
	DSR	Register name / SFR symbol	
	DUP	Directive operand	
	DW	Directive	
	DYNAMIC	Directive operand	
E	EA	Register name	
	ECSR	Register name	
	EDSR	Basic instruction	Not allowed
	EI	Basic instruction	
	ELR	Register name	
	ELSE	Directive	
	END	Directive	
	ENDIF	Directive	
	EPSW	Register name	
	EQ	Condition for BC instruction	
	EQU	Directive	
	ER0	Register name	
	ER2	Register name	
	ER4	Register name	
	ER6	Register name	
	ER8	Register name	
	ER10	Register name	
	ER12	Register name	
	ER14	Register name	
	ERR	Directive	
	EXTBW	Basic instruction	

	Reserved Word	Application	Notes
E	EXTRN	Directive	
F	FAR	Addressing type specifier / directive operand	
	FASTFLOAT	Directive	
	FP	Register name	
G	GBCY	Directive	
	GBEQ	Directive	
	GBGE	Directive	
	GBGES	Directive	
	GBGT	Directive	
	GBGTS	Directive	
	GBLE	Directive	
	GBLES	Directive	
	GBLT	Directive	
	GBLTS	Directive	
	GBNC	Directive	
	GBNE	Directive	
	GBNS	Directive	
	GBNV	Directive	
	GBNZ	Directive	
	GBOV	Directive	
	GBPS	Directive	
	GBZ	Directive	
	GE	Condition for BC instruction	
	GES	Condition for BC instruction	
	GJMP	Directive	
	GT	Condition for BC instruction	
	GTS	Condition for BC instruction	

	Reserved Word	Application	Notes
H			
I	ICESWI	Basic instruction	
	IF	Directive	
	IFDEF	Directive	
	IFNDEF	Directive	
	INC	Basic instruction	
	INCLUDE	Directive	
J			
K			
L	L	Basic instruction	
	LARGE	Directive operand	
	LE	Condition for BC instruction	
	LEA	Basic instruction	
	LES	Condition for BC instruction	
	LIST	Directive	
	LR	Register name	
	LT	Condition for BC instruction	
	LTS	Condition for BC instruction	
M	MODEL	Directive	
	MOV	Basic instruction	
	MUL	Basic instruction	
N	NC	Condition for BC instruction	
	NE	Condition for BC instruction	
	NEAR	Addressing type specifier / directive operand	
	NEG	Basic instruction	
	NODEBUG	Directive	
	NOERR	Directive	

	Reserved Word	Application	Notes
N	NOFAR	Directive	
	NOLIST	Directive	
	NOOBJ	Directive	
	NOP	Basic instruction	
	NOPAGE	Directive	
	NOPRN	Directive	
	NOREF	Directive	
	NOROMWIN	Directive	
	NOSYM	Directive	
	NS	Condition for BC instruction	
	NV	Condition for BC instruction	
	NVBIT	Directive / directive operand	
	NVBSEG	Directive	
	NVDATA	Directive / directive operand	
	NVRAM	Directive operand	
	NVSEG	Directive	
	NZ	Condition for BC instruction	
O	OBJ	Directive	
	OFFSET	Operator	
	OR	Basic instruction	
	ORG	Directive	
	OV	Condition for BC instruction	
	OVL_ADDRESS	Operator	
	OVL_OFFSET	Operator	
	OVL_SEG	Operator	
P	PAGE	Directive	
	PC	Register name	

	Reserved Word	Application	Notes
P	POP	Basic instruction	
	PRN	Directive	
	PS	Condition for BC instruction	
	PSW	Register name	
	PUBLIC	Directive	
	PUSH	Basic instruction	
Q	QR0	Register name	
	QR8	Register name	
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	
	R8	Register name	
	R9	Register name	
	R10	Register name	
	R11	Register name	
	R12	Register name	
	R13	Register name	
	R14	Register name	
	R15	Register name	
	RB	Basic instruction	
	RC	Basic instruction	
	REF	Directive	

	Reserved Word	Application	Notes
R	ROMWINDOW	Directive	
	RSEG	Directive	
	RT	Basic instruction	
	RTI	Basic instruction	
	RTICE	Basic instruction	
	RTICEPSW	Basic instruction	Not allowed
S	SB	Basic instruction	
	SC	Basic instruction	
	SEG	Operator	
	SEGMENT	Directive	
	SET	Directive	
	SIZE	Operator	
	SLL	Basic instruction	
	SLLC	Basic instruction	
	SMALL	Directive operand	
	SP	Register name	
	SRA	Basic instruction	
	SRL	Basic instruction	
	SRLC	Basic instruction	
	ST	Basic instruction	
	STACKSEG	Directive	
	SUB	Basic instruction	
	SUBC	Basic instruction	
	SWI	Basic instruction	
	SYM	Directive	
T	TAB	Directive	
	TABLE	Directive / directive operand	

	Reserved Word	Application	Notes
T	TB	Basic instruction	
	TBIT	Directive / directive operand	
	TITLE	Directive	
	TSEG	Directive	
	TYPE	Directive	
U	UNIT	Directive operand	
V			
W	WORD	Directive operand	
	WORD1	Operator	
	WORD2	Operator	
X	XOR	Basic instruction	
	XR0	Register name	
	XR4	Register name	
	XR8	Register name	
	XR12	Register name	
Y			
Z	ZF	Condition for BC instruction	