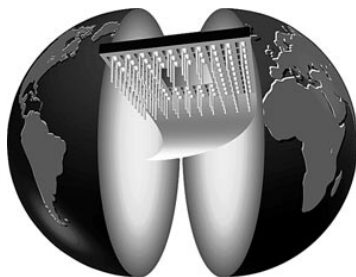


# **80C196 v6.1**

## **UTILITIES USER'S GUIDE**



A publication of  
TASKING  
Documentation Department  
Copyright © 1999 TASKING, Inc.

All rights reserved. Reproduction in whole or part is prohibited  
without the written consent of the copyright owner.

The following trademarks are acknowledged:

FLEXlm is a registered trademark of Globetrotter Software, Inc.  
HP and HP-UX are trademarks of Hewlett-Packard Co.  
Intel, MCS and ICE are trademarks of Intel Corporation.  
MS-DOS and Windows are registered trademarks of Microsoft Corporation.  
SUN is a trademark of Sun Microsystems, Inc.  
UNIX is a registered trademark of X/Open Company, Ltd.

All other trademarks are property of their respective owners.

Data subject to alteration without notice.

E-mail: [support@tasking.com](mailto:support@tasking.com)  
WWW: <http://www.tasking.com>

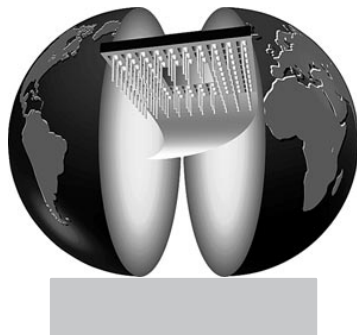
*The information in this document has been carefully reviewed and is believed to be accurate and reliable. However, TASKING assumes no liabilities for inaccuracies in this document. Furthermore, the delivery of this information does not convey to the recipient any license to use or copy the software or documentation, except as provided in an executed license agreement covering the software and documentation.*

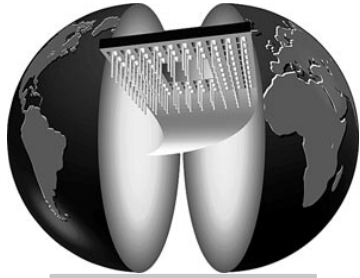
*TASKING reserves the right to change specifications embodied in this document without prior notice.*

# CONTENTS

## TABLE OF CONTENTS

---





---

# CONTENTS

---

## **OVERVIEW** **1-1**

1.1	80C196 Utilities and the Development Process . . . . .	1-3
1.1.1	Relocation and Linkage . . . . .	1-4
1.1.2	Use of Libraries . . . . .	1-4
1.1.3	FPAL96 Functions . . . . .	1-5
1.1.4	Audience Description . . . . .	1-6
1.1.5	ROM and PROM Versions . . . . .	1-7
1.1.6	Keeping Track of Files . . . . .	1-7
1.2	Conventions . . . . .	1-8
1.3	Customer Support . . . . .	1-8
1.3.1	If You Have a Problem Using the Software . . . . .	1-9

## **RL196 LINKER** **2-1**

2.1	Overview . . . . .	2-3
2.2	Resolving External References . . . . .	2-5
2.2.1	Type Checking . . . . .	2-5
2.2.1.1	Segment Type Matching . . . . .	2-6
2.2.1.2	Symbol Type Matching . . . . .	2-6
2.2.1.3	Mismatched Types . . . . .	2-8
2.2.2	Performing Fix-ups . . . . .	2-9
2.3	Variable Initialization . . . . .	2-9
2.4	Combining Different OMF96 Formats . . . . .	2-10
2.4.1	Global Initialization . . . . .	2-10
2.4.2	OMF96 Version 3.0 Limitations . . . . .	2-11
2.5	Memory Allocation . . . . .	2-11
2.5.1	Stack Segment . . . . .	2-12
2.5.2	Stack Overflow . . . . .	2-13
2.5.3	Register Overlaying . . . . .	2-14
2.5.4	Providing Means for Dynamic Memory Allocation . . . . .	2-15
2.5.5	Overlapping ROM and RAM Memory Using the INST Pin . . . . .	2-16
2.5.5.1	INST Pin Behavior . . . . .	2-17
2.5.5.2	Overlapping Memory Scheme Example . . . . .	2-17
2.5.5.3	Guidelines for Hardware Development . . . . .	2-18

2.5.5.4 Linker/Locator Example Invocation Line ..... 2-19

2.5.5.5 Summary of RL196 INST Usage ..... 2-20

2.5.6 Support for Vertical Windows ..... 2-20

2.6 Invoking RL196 ..... 2-22

2.6.1 Options ..... 2-23

2.6.2 Input List ..... 2-26

2.6.2.1 Ordinary Object File ..... 2-27

2.6.2.2 Object Library File ..... 2-28

2.6.2.3 Publiconly Object File ..... 2-29

2.6.3 Output Files ..... 2-30

2.6.3.1 Print File ..... 2-30

2.6.3.2 Output Object File ..... 2-37

2.7 Automatically Invoking Multiple Commands ..... 2-38

2.7.1 Using Make Utility mk196 ..... 2-38

2.7.2 Using Batch Files ..... 2-39

2.7.3 Using Command Files ..... 2-41

2.8 RL196 Controls ..... 2-43

**OH196 CONVERTER** **3-1**

3.1 Invocation Syntax ..... 3-3

3.2 Examples ..... 3-4

3.3 Output File ..... 3-4

**LIB196 LIBRARIAN** **4-1**

4.1 Invoking LIB196 ..... 4-3

4.1.1 Options ..... 4-3

4.1.2 Character Set ..... 4-4

4.2 LIB196 Commands ..... 4-4

4.3 Command Descriptions ..... 4-5

## **USING THE FPAL96 LIBRARY** **5-1**

5.1	Data Formats Supported . . . . .	5-3
5.1.1	Floating Point Numbers . . . . .	5-3
5.1.1.1	Special Floating Point Numbers . . . . .	5-5
5.1.2	Integers . . . . .	5-11
5.1.3	Decimals . . . . .	5-11
5.2	Conventions . . . . .	5-12
5.2.1	Naming Conventions . . . . .	5-12
5.2.2	Parameter Passing . . . . .	5-12
5.3	FPAL96 Control Variables . . . . .	5-13
5.3.1	Floating-point Accumulator . . . . .	5-13
5.3.2	Built-in Variables . . . . .	5-13
5.3.2.1	Control Word . . . . .	5-14
5.3.2.2	Status Word . . . . .	5-16
5.4	Declaration and Linkage . . . . .	5-18
5.4.1	Declaring Floating-point Functions . . . . .	5-18
5.4.1.1	In an ASM196 Program . . . . .	5-18
5.4.1.2	In an C196 Program . . . . .	5-19
5.4.2	Selecting the Correct Library . . . . .	5-19
5.4.3	Initializing the Floating-point Library . . . . .	5-20
5.4.4	Linking the Floating-point Library . . . . .	5-20
5.5	Examples Using FPAL96 Routines . . . . .	5-21

## **FPAL96 FUNCTIONS REFERENCE** **6-1**

6.1	Introduction . . . . .	6-3
6.2	Administrative Operations . . . . .	6-3
6.3	Load Operations . . . . .	6-3
6.4	Store Operations . . . . .	6-4
6.5	Unary Operations . . . . .	6-4
6.6	Binary Operations . . . . .	6-5
6.7	Functions List . . . . .	6-6



**EXCEPTIONS AND EXCEPTION HANDLING** **7-1**

7.1	Introduction . . . . .	7-3
7.2	Invalid-operation Exception . . . . .	7-5
7.3	Zero-divide Exception . . . . .	7-6
7.4	Overflow Exception . . . . .	7-6
7.5	Underflow Exception . . . . .	7-7
7.6	Precision Exception . . . . .	7-7
7.7	Denormalized-number Exception . . . . .	7-8
7.8	Creating Your Own Exception Handler . . . . .	7-8

**MK196 MAKE UTILITY** **8-1**

8.1	Invocation Syntax . . . . .	8-3
8.2	Description . . . . .	8-3
8.3	Usage . . . . .	8-5
8.4	Example . . . . .	8-14
8.5	Files . . . . .	8-15
8.6	Diagnostics . . . . .	8-15

**MESSAGES AND ERROR RECOVERY** **9-1**

9.1	RL196 Messages . . . . .	9-3
9.1.1	Console Messages . . . . .	9-3
9.1.2	Fatal Errors . . . . .	9-4
9.1.2.1	RL196 Error Messages . . . . .	9-4
9.1.2.2	Argument Error Messages . . . . .	9-11
9.1.2.3	Memory Error Messages . . . . .	9-12
9.1.2.4	I/O Error Messages . . . . .	9-13
9.1.3	Error Messages . . . . .	9-14
9.1.4	Warnings . . . . .	9-18
9.2	OH196 Error Messages . . . . .	9-22
9.3	LIB196 Error Messages . . . . .	9-23

**GLOSSARY** **A-1**

**INDEX**



CONTENTS

## **MANUAL PURPOSE AND STRUCTURE**

### **PURPOSE**

This manual describes how to use the TASKING utilities RL196, OH196, LIB196 and MK196 and the FPAL96, a floating-point object module library for 80C196 microcontrollers. To effectively use the 80C196 utilities, you must be familiar with the 80C196 architecture, programming in assembly language or a high-level language, and the software development process.

### **INSTALLING THE UTILITIES**

To install the 80C196 utilities, see the *Software Installation* chapter in the *80C196 Assembler User's Guide* or the *80C196 C Compiler User's Guide*. That chapter also explains the environment variable settings, and directory structure to set up your system for the translator and utilities.

### **MAKING EFFICIENT USE OF RL196**

To understand how RL196 operates on your modules, read Chapter 2. This chapter explains the default settings of the linker, and it can help you use the correct linking and locating controls in the linker invocation line.

### **USING FLOATING-POINT FUNCTIONS**

To learn about the major functions of FPAL96, see Chapter 1. To learn how to control the behavior of the FPAL96 library, read Chapter 5. To see examples on how to use each function in ASM196, and C196, see Chapter 6.

### **MANUAL STRUCTURE**

Related Publications  
Conventions Used In This Manual

1. Overview  
Summarizes the 80C196 utilities and introduces you to the 80C196 floating-point library (FPAL96).
2. RL196 Linker  
Deals with linker invocation, output files and contains a detailed description of the linking and locating controls.
3. OH196 Converter  
Third-party vendors' PROM programmers do not always accept the 80C196 Absolute Module Format. In this case, you can run the absolute object file created by **rl196** through the **oh196** utility which produces a file in hexadecimal format.
4. LIB196 Librarian  
Describes how to create and maintain libraries.
5. Using the FPAL96 Library  
Describes the 80C196 floating-point library (FPAL96).
6. FPAL96 Functions Reference  
Explains how to use the floating-point functions in ASM196 and C196.
7. Exceptions and Exception Handling  
Describes how to create your own exception handler and explains the different types of exceptions FPAL96 can generate while running.
8. MK196 Make Utility  
Describes how to maintain, update, and reconstruct your application software.
9. Messages and Error Recovery  
Describes the error/warning messages of the linker and utilities.

## **APPENDICES**

- A. Glossary  
Contains an explanation of terms.

## **INDEX**

## **RELATED PUBLICATIONS**

- IEEE Standard for Floating-point Arithmetic 754–1985

### ***TASKING publications***

- 80C196 C Compiler User's Guide [TASKING, MA006022]
- 80C196 Assembler User's Guide [TASKING, MA006020]
- 80C196 Utilities User's Guide [TASKING, MA006009]

### ***Intel publications***

- Embedded Microcontrollers and Processors Handbook [270645]
- 8XC196xx User's Manuals

## CONVENTIONS USED IN THIS MANUAL

The notation used to describe the format of call lines is given below:

`{ }` Items shown inside curly braces enclose a list from which you must choose an item.

`[ ]` Items shown inside square brackets enclose items that are optional.

`|` The vertical bar separates items in a list. It can be read as OR.

*italics* Items shown in italic letters mean that you have to substitute the item. If italic items are inside square brackets, they are optional. For example:

*filename*

means: type the name of your file in place of the word *filename*.

`...` An ellipsis indicates that you can repeat the preceding item zero or more times.

`[,...]` You can repeat the preceding item, but you must separate each repetition by a comma.

`screen font` Represents input examples, keywords, filenames, controls and screen output examples.

**bold font** Represents a command name, an option or a complete command line which you can enter.

### For example

`command [option]... filename`

This line could be written in plain English as: execute the command *command* with the optional options *option* and with the file *filename*.

### Illustrations

The following illustrations are used in this manual:



This is a note. It gives you extra information.



This is a warning. Read the information carefully.



This illustration indicates actions you can perform with the mouse.



This illustration indicates keyboard input.



This illustration can be read as “See also”. It contains a reference to another command, option or section.





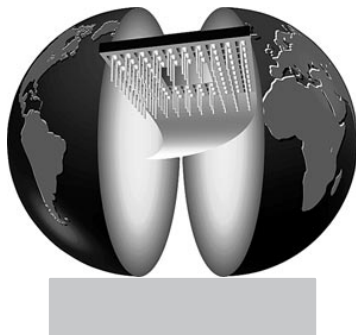
MANUAL STRUCTURE

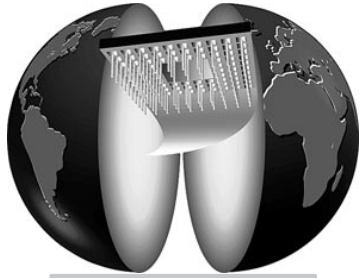
# CHAPTER

# 1

## OVERVIEW

---





---

# 1

# CHAPTER

---

This chapter introduces you to the 80C196 utilities, the 80C196 floating-point library (FPAL96) and to this manual. Intended for the new user, this overview helps you understand the general functions of the utilities and the general functions of the floating-point library.

## 1.1 80C196 UTILITIES AND THE DEVELOPMENT PROCESS

The 80C196 utilities combine all of your object modules and libraries to complete your application program. These utilities include the RL196 linker/locator, LIB196 library manager program, and the OH196 object-to-hexadecimal converter. These utilities perform different roles in the software development process. Figure 1-1 shows where each tool is used in the development process. The following sections give a brief overview of the utilities' role in the development process.

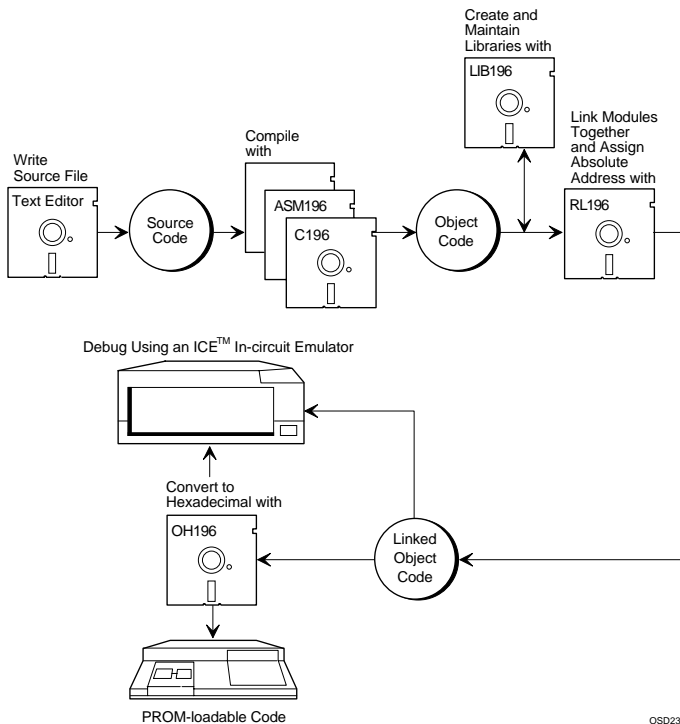


Figure 1-1: The 80C196 application development process

### **1.1.1 RELOCATION AND LINKAGE**

After all of the modules are translated, the RL196 linker/locator processes the object module files. RL196 treats each relocatable segment as an independent unit. The linker allocates all of the relocatable code segments to ROM, the relocatable data and stack segments to RAM, and the relocatable register segments (both overlayable and non-overlayable) to a register area. RL196 also resolves all references between modules, possibly using library files and `publiconly` files. `Publiconly` files are discussed in Chapter 2. The linker produces both an absolute object module file of the complete program and a print file showing the results of the link/locate process, including a segment map, a symbol table, and an inter-module cross-reference listing.

### **1.1.2 USE OF LIBRARIES**

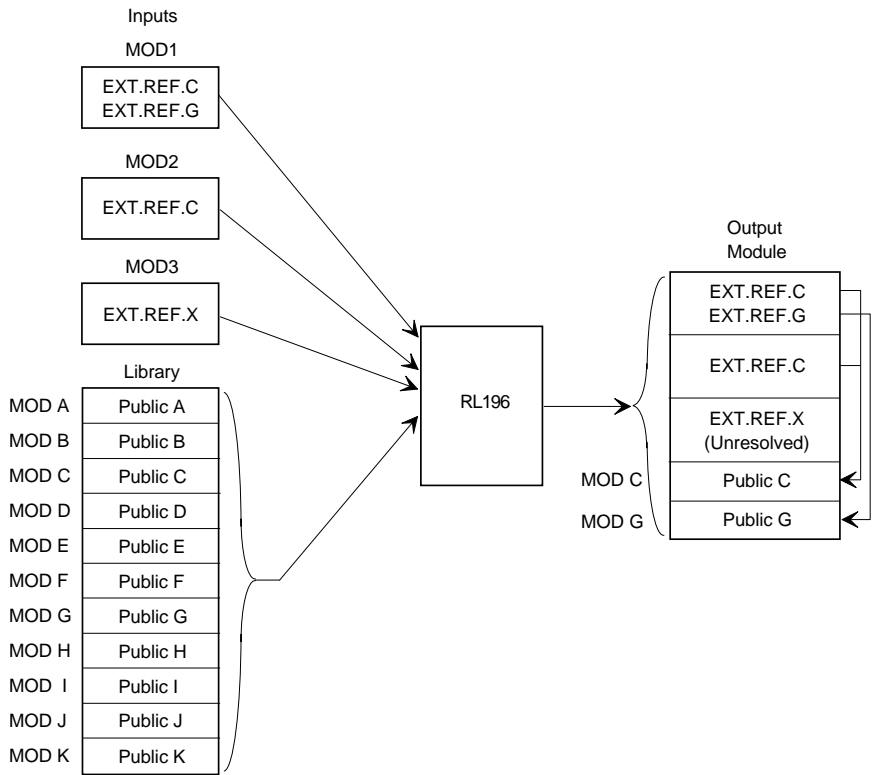
Libraries help build programs. LIB196, the library manager program, creates and maintains files containing object modules by adding, replacing, deleting, and extracting members. LIB196 works both in an interactive and a non-interactive mode.

The RL196 linker treats library files in a special manner. If you specify a library file as input to RL196, the linker searches the library for modules that satisfy unresolved external references in the input modules that the linker has already read. Therefore, you must specify libraries after the input modules that contain external references. If a module included from the library has an external reference, RL196 searches the library again to try to satisfy this reference. This process continues until all external references are satisfied or until no new public symbols that match an unsatisfied external reference are found in the library.

When the linker searches a library, the linker generates an output module that only includes the library modules that satisfy external references. However, RL196 provides the means to unconditionally include a library module even if the library module does not satisfy any external reference. Figure 1-2 shows RL196 handling a library file in the conditional manner.



See Section 2.6.2 for more information on how to select input modules.



QSD986

Figure 1-2: Library Linkage by **rl196**

1.1.3 **FPAL96 FUNCTIONS**

The 80C196 floating-point library (FPAL96) supports the basic floating-point operations for ASM196 and C196 applications. This library allows your application to execute floating point number math using single-precision format, according to the IEEE Standard for Floating-point Arithmetic 754-1985. FPAL96 is an application library. Therefore, you must use the RL196 linker to link the library to your application.



See Chapter 5 for an example of the RL196 invocation.

FPAL96 provides the following functions:

Load and store	These functions perform format conversion between floating point and integer or decimal.
Binary	These functions perform comparisons and arithmetic operations.
Unary	These functions perform sign conversions and square root.
Administrative	These functions allow you to control the behavior of the FPAL96 library.

FPAL96 has one local data structure called the floating-point accumulator (FPACC) and two built-in variables, called the control word and the status word. The FPACC accumulator serves as an implicit operand to all non-administrative functions. The control word contains the exception mask bits and the rounding-mode bits. The status word contains the status of the FPACC and indicates any pending exceptions. You can use these two built-in variables with your exception handler to continue a flagged operation or to analyze results when debugging.



See Chapter 5 for more details.

FPAL96 recognizes standard exception conditions and enables you to respond to the exceptions with your own exception handler or with the default exception handler of the library. Chapter 7 explains how to include your own exception handler.

#### **1.1.4 AUDIENCE DESCRIPTION**

To effectively use the FPAL96 library, you must be familiar with the ASM196 assembler or translator and utilities, and you must have an understanding of floating-point numbers.

### 1.1.5 ROM AND PROM VERSIONS

You can load the absolute object module produced by RL196 into members of the 80C196 family of microcontrollers. For ROM versions of the microcontroller, the program is masked into ROM during manufacturing. For PROM versions and versions with no on-chip code memory, use a PROM programmer to load the absolute module into program memory that is accessible to the microcomputer for execution. Some PROM programmers require the absolute module to be in hexadecimal format. Use the OH196 utility, discussed in Chapter 3, to perform this function. See the *Embedded Microcontrollers and Processors Handbook*, listed in *Related Publications*, for details on the available microcontroller versions.

### 1.1.6 KEEPING TRACK OF FILES

The RL196 linker appends the `.m96` extension to the default map file, to avoid destroying the `.lst` file generated by the assembler or compiler. Executable files, by default, have the `.abs` extension.

We suggest to use the following filename extensions. This naming convention is not required, but it allows utilities (like **mk196**) to execute so-called 'suffix rules'. Note that all names and extensions are in lower case, because on UNIX systems it is case sensitive.

Extension	Description
<code>.c</code> <code>.c96</code>	C file ( <code>.c</code> is preferred, no extension is forced or assumed by the compiler).
<code>.h</code> <code>.h96</code>	Include files for C ( <code>.h</code> is preferred, the compiler does not look for <code>.h96</code> by itself).
<code>.a96</code> <code>.asm</code> <code>.src</code>	Assembly source files ( <b>mk196</b> uses <code>.a96</code> ).
<code>.inc</code>	Include file for assembly.
<code>.cmd</code>	Command file for <b>asm196</b> or <b>c196</b> .
<code>.obj</code>	OMF96 object file produced by <b>c196</b> or <b>asm196</b> .
<code>.lst</code>	LIST files from <b>c196</b> or <b>asm196</b> .
<code>.lnk</code>	Linker command control file.
<code>.out</code>	File containing linked object with unresolved externals.



Extension	Description
.abs	File containing absolute object of application, no remaining unresolved externals (default output file of <b>rl196</b> ).
.m96	Linker MAP file.
.mak	For Makefiles other than 'Makefile' or 'makefile'.
.hex	Hexadecimal output file by <b>oh196</b> .

*Table 1-1: Filename extensions*

Programmers who at present work on MS-DOS but are thinking of future migration to other platforms (UNIX, Windows-NT, etc.) are advised to use lower case characters and forward slashes where possible. This will smoothen the future transition and it will not hurt right now. All the tools are able to find files if forward slashes are used. (Note however that MS-DOS still does not like you to say: `c:/c196/bin/rl196`)

## **1.2 CONVENTIONS**

The colon-arrow ( $\Rightarrow$ ) characters denote a further breakdown of a placeholder.

This manual also uses the conventions listed in the *Conventions Used In This Manual* at the beginning of this manual.

## **1.3 CUSTOMER SUPPORT**

The 80C196 software is under warranty. During the warranty period you are entitled to the following:

- Free replacement of any defective media upon notification in writing of the defect and product information.
- Telephone consultation and bug reporting.
- Our best efforts to replace or repair any software that does not meet the specification described in the 80C196 documentation.

TASKING offers various support contracts that provide benefits as free product updates, reduced rate upgrades, and telephone support. Contact your local TASKING sales representative, for information about support contracts and standard warranties. You will find the addresses and telephone numbers in the "Read This First" Envelop included with this package.

### **1.3.1 IF YOU HAVE A PROBLEM USING THE SOFTWARE**

To help expedite your calls, please have the following information available when you contact us for help.

- The serial number of your software distribution. This number is printed on the label of the tape, cassette, or first floppy of your software distribution. In addition, you may be able to obtain the serial number by running one of the utilities with option **-V**, you may wish to record the serial number here:

**Product:** \_\_\_\_\_

**Serial:**

- The product name, including host, target processor, and release number.
- The exact command line that you used to invoke our tools when you encountered the problem. Please include all switches.
- The exact error message that was printed. A screen dump will often make this easy to record, and can provide very useful information.
- Any additional information that may be useful in helping to define the problem. Examples include:
  - your directive-file for RL196, and invocation line
  - print file of RL196.
  - relevant information about your compilation environment
  - the emulator you are using



# OVERVIEW

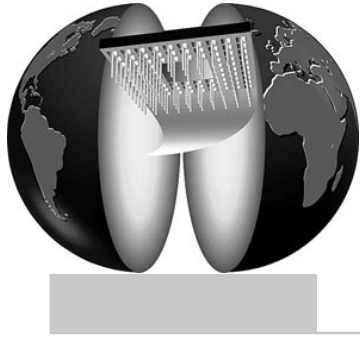
# CHAPTER

## RL196 LINKER

---

# 2





# 2

# CHAPTER

## 2.1 OVERVIEW

This chapter describes the operation of the RL196 program. Most of the process is transparent to the user; however, an understanding of the operation at the level presented here can help you link and locate with RL196 more effectively. Details on some of the subjects presented here are given with the control descriptions in Section 2.8.

The RL196 linker/locator combines all of your translated modules to produce one absolute object file. This utility performs the following major functions:

- Selects modules to be included in the output object file.
- Combines input object modules into a single object file, optionally filtering translator-generated debug information.
- Resolves symbolic intermodule references for the input modules.
- Allocates memory to input segments and binds relocatable addresses to absolute addresses.
- Produces a print file that consists of a link summary, a symbol table listing, and an intermodule cross-reference listing.
- Detects and lists errors found in the invocation command, in the input modules, or during the link-locate process.

To link your modules, RL196 makes two passes. During the first pass, RL196 extracts all of the intermodule reference information and segment definitions from the input modules. By the end of the first pass, the linker allocates memory to both the relocatable and the absolute segments and sets the base addresses of the relocatable segments. RL196 then determines the absolute addresses of the public symbols.

During the second pass, the linker generates the output object module from the input object modules and from the information gathered in the first pass. RL196 then assigns absolute addresses to relocatable symbols and modifies the content records according to fix-ups (i.e., instructions placed in the object file that allow RL196 to fix addresses that are left incomplete by the translator or the linker). The print file is generated during both the first and the second pass.

After normal translation, the translator marks an object module either as a main or non-main module. An application can only have one main module. This object module contains zero or more segments which the translator tagged either as absolute or relocatable. The RL196 linker first locates the absolute segments in the appropriate section of memory starting at the base address specified. Then, the linker locates the relocatable segments in the appropriate section of memory based on their segment type and in accordance with their alignment. RL196 allocates these segments according to the memory allocation process described in Section 2.5. Because the placement procedure takes alignment into account while placing relocatable segments, one, two, or three bytes can be wasted to ensure that segments begin on the proper boundaries.

The linker treats the stack segment in a special manner. An application can have at most one stack. RL196 places the stack, a combination of all of the stack segments from each input module, contiguously in one section of RAM. Since a segment can be located only once, you must defer making the stack segment absolute until all of the information about the stack is present. Use the `noabsstack` (see Section 2.8) control to suppress the stack segment to absolute form. When you specify `noabsstack`, the output of RL196 is termed quasi-absolute, which means all segments are absolutely located except for the stack segment. Quasi-absolute files, like relocatable files, can contain unresolved external references that can be resolved upon subsequent execution of RL196. Quasi-absolute files must be further used as input to an incremental link process.

Use the `stacksize` control to specify a stack size other than the default as long as the input stack segments are still relocatable. The linker issues a warning message if the size of the resultant stack segment is greater than the size you specified in the control.

When you finally specify `absstack`, the linker produces an object module that contains no relocatable content, including its stack segment. This absolute object file can serve as input to loaders, such as in-circuit emulators and PROM programmers.

## **2.2 RESOLVING EXTERNAL REFERENCES**

One module can contain public definition records if public symbols are declared within the module. These records consist of the names and attributes of the symbols declared as public. RL196 determines the absolute address of each public symbol after all of the segments are allocated and their base addresses are established. The linker generates a warning message when two or more of these public symbols have the same name.

One module can also contain external definition records if external symbols are accessed within the module. These records consist of the names and the attributes of externally defined symbols.

The linker resolves external symbol references by finding a public symbol with the same name. The definitions of the public and external symbols with the same name must match. See Section 2.2.1 for details on symbol type matching.

When resolving an external reference, the linker uses the absolute address of the corresponding public symbol to set an absolute address in the output object module, replacing the external reference.

If unresolved external symbols exist in the output object module, RL196 issues the appropriate warning messages, and the names of the symbols appear in the UNRESOLVED EXTERNAL SYMBOLS list. However, you can resolve these external references by subsequent execution of RL196 with some new input modules.

### **2.2.1 TYPE CHECKING**

During the processing of global symbols, public and external, the linker performs type checking. Two symbols with the same name match if they have:

- matching segment types (code, register, data, etc.), and
- matching symbol types (byte, word, long, etc.)

To indicate whether or not to perform type checking by using the control `typecheck/notypecheck`. The default `typecheck` reflects the normal case.



2.2.1.1 SEGMENT TYPE MATCHING

Table 2-1 shows the different combination of segment types that can form a match. An M in the column indicates a match. The segment type `dynamic` is not allowed as the segment type of a global symbol.

Segment Type	Code	High Code	Far Code	Const	Far Const	Data	Far Data	Reg	Over lay	Stack	Null
CODE	M	M									M
HIGH CODE	M	M									M
FAR CODE			M								M
CONST				M	M						M
FAR CONST					M						M
DATA						M	M			M	M
FAR DATA							M				M
REGISTER						M	M	M	M	M	M
OVERLAY						M	M	M	M	M	M
STACK						M	M			M	M
NULL	M	M	M	M	M	M	M	M	M	M	M

Table 2-1: Segment type matching

2.2.1.2 SYMBOL TYPE MATCHING

Two ASM196 global variables with the same name match if they have exactly the same symbol type or one of the global variables has a symbol type of `null`.

Two C196 global variables with the same name match if they have the same symbol type.

For scalars, the symbol type means the scalars must be identical.

For arrays, the symbol type matches when:

- The same type of elements appear in both arrays.
- Each array has the same number of elements, unless you specify one or more of them with an implicit dimension. For example, to declare a constant array called `xsub` with an implicit dimension specifier, do the following:

```
const int xsub[] = { 402, 345, 126 };
```

This declaration defines `xsub` to be an constant array of three, with values of 402, 345, and 126 respectively. See the *80C196 C Compiler User's Guide*, listed in *Related Publications*, for a more detailed explanation on arrays.

For structures, the symbol type matches when:

- The list of members contains the same names and in the same order.
- Respective structure members have the same type.
- Procedures or functions have the same symbol type when:
- The procedures are of the same type (typed/untyped).
- The procedures return the same type values if they are typed procedures.
- The procedures have the same number of parameters, and the respective parameters have the same type.

Table 2-2 describes the symbol type matching between ASM196 global variables and C196 global variables. An M in the column indicates a match.

ASM196		BYTE	WORD	LONG	REAL	ENTRY	NULL
C196	UNSIGNED CHAR	M					M
	UNSIGNED INT		M				M
	UNSIGNED LONG			M			M
	CHAR						M
	INT						M
	LONG						M
	FLOAT						M
	ARRAY						M
	STRUCT						M
	FUNCTION						M

Table 2-2: ASM196 – C196 symbol type matching

ASM196 has no symbol type that matches the symbol types `array` and `structure` of C196 except for `null`. Therefore, to avoid a mismatch warning, you must explicitly or implicitly attach the symbol type `null` to the symbol in the ASM196 module. For example, your C196 module compiled with `registers(200)` contains the following structure declaration:

```
extern struct c_struct { char a[10]; };
```

The compiler places the structure in the data segment. You can then declare the structure with a `null` type in ASM196:

```
dseg
c_struct equ $:null
dsb 10
```

### **2.2.1.3 MISMATCHED TYPES**

The RL196 linker issues a `TYPE MISMATCH` warning for a symbol type mismatch or a segment type mismatch, or both.

Although a type mismatch is only a warning and the execution of the link-locate process continues smoothly, you must be aware of the precise action taken by RL196 in such a situation. The action affects both the output object file and the way in which the other occurrences of the same global symbol are regarded, as explained below.

The following rules determine the output in case of a mismatch:

- When the mismatch occurs between two globals of the same nature, either both `public` or both `external`, the dominant one is the one processed first.
- When the mismatch occurs between two globals with a different nature, that is, one `public` and one `external`, the dominant one is the `public`.

This rule applies to the segment type attribute and to the symbol type attributes in case of type mismatch. Both actions are independent.

## 2.2.2 PERFORMING FIX-UPS

The object modules that serve as input to RL196 can contain portions of object code left undefined by the language translator or by RL196. The portions left undefined can contain a reference to a relocatable symbol that must be bound, or a reference to an external symbol that must be resolved.

The linker performs all binding and resolving operations that are possible at link-locate time. References to unresolved external symbols and, under `noabsstack`, to a relocatable stack segment causes an output of fix-up records to the output object file. Errors in the evaluation of a fix-up expression cause error messages numbered 111, 112, and 113 to be generated.



See Chapter 9 for complete list of RL196 error messages.

Consider the following example of a simple fix-up: at location `n` in an object module, a reference is made to symbol `x`, defined in another module. To get RL196 to store the correct value at location `n`, the translator must include a fix-up that instructs RL196 to determine the absolute address of `x` by computing the sum of the base of the segment that contains `x` and the offset of `x` from that base. This value is then stored at location `n`.

If you are using the vertical windowing feature of the 80C196KC or 80C196KR processors, the linker calculates the fix-up of a variable in an overlay segment located in a window above `0FFH` as the sum of the offset of that variable, with respect to the window base, and the base of the mapped window at the top of the register file from `00H` – `0FFH`. See the *80C196 C Compiler User's Guide*, listed in *Related Publications*, and Section 2.5.6 for more information on vertical windowing.

## 2.3 VARIABLE INITIALIZATION

When you use global variable initialization in your source code, the linker has to locate both the variable space (which is in RAM) and the initial values for these variables (which are in ROM). The library routine `_imain()`, called during startup, then copies the initial values to their corresponding variables. In order to tell the `_imain()` routine which parts of ROM must be copied to RAM, the linker generates an initialization table in ROM. A global symbol `_INIT_TABLE_START_` is generated to point at this initialization table.

The initialization table will not be generated if there is no initialization data. The public symbol `_INIT_TABLE_START_` is needed by the function `_imain()` and is, therefore, always generated. If you do not need global `_imain()`, you can suppress `_INIT_TABLE_START_` from being generated by using the `noinittable` or `noit` control.

## 2.4 COMBINING DIFFERENT OMF96 FORMATS

As of version 5.0, you can specify three different OMF96 formats for the 196 tools. See the `omf` control on how to specify a specific OMF96 format. By default our tools use the OMF96 version 3.2 format. This format contains extra debugging info and support for using initialized global variables. We recommend that you use this default OMF96 format. You can specify the different OMF96 formats with the `omf` control:

```
omf ( 2 )  OMF96 version 3.2
omf ( 1 )  OMF96 version 3.0
omf ( 0 )  OMF96 version 2.0
```

### 2.4.1 GLOBAL INITIALIZATION

It is necessary to use OMF96 version 3.2 if you want to use global initialization. However, it is possible to create an `.abs` file which is OMF 3.0 compatible, but still contains global initialization. This might be necessary for certain third party tools which do not (yet) recognize the new OMF96 format. To do so, you have to use the (default) `omf ( 2 )` for both the compiler and the assembler, and use `omf ( 1 )` for the linker. The resulting `.abs` file has the OMF96 version 3.0 format, but contains all necessary code for global initialization. The same is true for using the libraries. The libraries provided with our tools are compiled with the default `omf ( 2 )` control. If you want to get an OMF96 version 3.0 compatible `.abs` file, just specify `omf ( 1 )` in the linker controls and you can use our default libraries.

A word of caution: if you specify `omf (1)` in your linker controls and if you have any unresolved externals in your application, it is possible that the linker will give a fatal OMF96 error. This is caused by the fact that you have specified OMF96 version 3.0, but the linker needs to write information about the unresolved externals in OMF96 version 3.2 format. You will see a warning about the unresolved externals before you get the fatal `omf` error. So, do not have any unresolved externals when you convert from OMF96 3.2 format to OMF96 3.0 format.

### **2.4.2 OMF96 VERSION 3.0 LIMITATIONS**

OMF96 version 3.0 has the following limitations compared to OMF96 version 3.2:

- Limited support for functions.
- Limited support for structures.
- Limited support for unions.
- Limited support for bit fields.
- No support for vertical windowing.
- Restricted line number information.

## **2.5 MEMORY ALLOCATION**

RL196 enables you to specify the actual memory available for location so that different applications can use different address spaces. To specify memory locations, use the `rom` control for code segments and constant segments or use the `romcode` control for code segments and the `romdata` control for constant segments, and use the `ram` control for data and stack segments. The linker locates register segments (overlayable and non-overlayable) in the internal register memory (1AH-0FFH by default) specified by the `registers` control. Section 2.8 describes the `rom`, `romcode`, `romdata`, `ram` and `registers` controls.

RL196 follows this order of allocation:

1. All absolute segments.
2. If the `regfirst` control is specified relocatable register segments are allocated first, followed by relocatable overlay segments of modules specified by the `regoverlay` control and relocatable overlay segments not yet allocated.

3. If the `regfirst` control is not specified relocatable overlay segments of modules specified by the `regoverlay` control are allocated first, followed by relocatable overlay segments not yet allocated and relocatable register segments.
4. Relocatable code and constant segments of modules specified by the `romcode`, `romdata` and/or `rom` control, and relocatable data segments specified by the `ram` control. Then, the linker allocates the stack if you specify `stack` or `st` with the `ram` control and the input stack segment is relocatable and the `absstack` control is in effect.
5. Relocatable segments not yet allocated.
6. Reserve the remaining free RAM as HEAP space (if specified).

The linker allocates memory to absolute segments at their set base addresses. This process can cause additional fragmentation of the memory available for allocation. The linker reports an error if an absolute segment is placed in an incompatible memory section or if two absolute segments overlap.

RL196 allocates memory to relocatable segments in the appropriate sections that do not overlap absolute segments. Relocatable segment allocation takes into account both the segment size and the segment alignment according to a first fit/decreasing size (FFDS) algorithm. Memory allocation determines the absolute base address for all relocatable segments including the stack segment if `absstack` is in effect. If no room is available for a relocatable segment, that segment appears in the list of `UNALLOCATED SEGMENTS` and the linker issues an appropriate error message.

### 2.5.1 **STACK SEGMENT**

The stack segment is a special segment in which the linker will locate the stack. This stack is used by the compiler, for instance to store temporary variables or to pass parameters to the functions. There are three ways in which the linker can determine the size of the stack:

1. By default the linker calculates the stack size by adding the sizes of all stack segments of all input modules. This method is accurate only if there are no recursive function calls. If there are any recursive function calls, you will have to increase the stack size with either one on the following two methods.

2. It is possible for the user to specify a stack segment in an assembly file by using the user defined stack segment SSEG. You can only have one user defined stack in an application. This user defined stack will overrule the stack size as calculated by the linker. The linker will issue no warnings if the size of the user defined stack is smaller than the stack size as calculated by the linker.

The following example in assembly declares a user defined stack with a size of 256 bytes:

```
SSEG
DSB  0100H
.
.
END
```

3. The final way to specify the stack size is by using the STACKSIZE control. This control overrules both the default calculated stack and the user defined stack. If the specified stack size is smaller than the calculated stack size, the linker will issue an warning, unless a user defined stack was defined. In that case the linker will use the specified stack size without issuing a warning.

Make sure that the stack size as defined in a user defined stack segment or with the STACKSIZE control is large enough. Specifying a stack size which is too small will most likely result in a crash of your application.

### **2.5.2 STACK OVERFLOW**

Some 80C196 models have support to detect stack overflow. This StackOverflow Module (SOM) has 2 SFRs that store the upper and lower SP boundaries. The linker generates two symbols, `_TOP_OF_STACK_` and `_BOTTOM_OF_STACK_`, that represent the upper and lower stack boundaries. It is up to you to load the SFRs with the linker generated symbols in your program. For example:



```

EXTRN  _TOP_OF_STACK_
EXTRN  _BOTTOM_OF_STACK_
.
.
LD      TMPREG0, #_TOP_OF_STACK_
ST      stack_top, TMPREG0
LD      TMPREG0, #_BOTTOM_OF_STACK_
ST      stack_bottom, TMPREG0
.
.

```

The two symbols `_TOP_OF_STACK_` and `_BOTTOM_OF_STACK_` will be set to the boundaries on the stack. If the stack is located at 0300H with a size of 0100H the stack pointer SP will be initialized with 0400H and `_TOP_OF_STACK_` and `_BOTTOM_OF_STACK_` will have the values 0402H and 02FEH respectively. This is conform the specification of the SOM. The upper limit comparator compares for a `SP >= stack_top` condition while the lower limit comparator compares for a `SP <= bottom_stack` condition. If at a later date the behavior of SOM changes, you can easily load other values, for example:

```

LD      TMPREG0, #_TOP_OF_STACK_ - 2
ST      stack_top, TMPREG0
LD      TMPREG0, #_BOTTOM_OF_STACK_ + 2
ST      stack_bottom, TMPREG0

```

### 2.5.3 REGISTER OVERLAYING

Register overlaying is part of the memory allocation process performed by RL196 (see step 2 of the order of allocation in Section 2.5). Each input module has, at most, one relocatable overlay register segment. RL196, by default, regards them as register segments. However, you can specify the control `regoverlay` with an appropriate parameter and request overlaying the overlay segments of the specified modules. For details on how to use this control, see `regoverlay` in Section 2.8.

The memory allocation algorithm for relocatable overlay segments is a combination of the principle of FFDS algorithm and an algorithm dealing with overlaying. As is usual in memory allocation algorithms, the algorithm is not necessarily optimal.

### **2.5.4 PROVIDING MEANS FOR DYNAMIC MEMORY ALLOCATION**

Following the memory allocation process and providing that you specified the `absstack` control, RL196 will generate symbols that are used for dynamic memory allocation. If you specify the `heap` or `he` control, RL196 supplies four public symbols, `_HEAP_START_`, `_HEAP_END_`, `MEMORY` and `?MEMORY_SIZE`. If you omit the `heap` or `he` control, RL196 only supplies `MEMORY` and `?MEMORY_SIZE`. When you use the dynamic memory location routines in the libraries, the symbols `_HEAP_START_` and `_HEAP_END_` are needed. The symbols `MEMORY` and `?MEMORY_SIZE` are provided for backward compatibility.

If the `heap` or `he` control is specified, RL196 finds the largest free RAM section, and assigns its base address to `_HEAP_START_` and its end address to `_HEAP_END_`. This section is referred to as the HEAP space. It is also possible to specify `HEAP` as a module name in the `ram` control. In that case, RL196 finds the largest free RAM section in the specified RAM range and assigns its base address to `_HEAP_START_` and its end address to `_HEAP_END_`.

After locating the HEAP space, or when the `heap` or `he` control is omitted, RL196 finds the largest free RAM section, and assigns it base address to `MEMORY` and its size, in bytes, to `?MEMORY_SIZE`.

When using dynamic memory, be aware of these requirements:

- If no free RAM section is found, all symbols (`_HEAP_START_`, `_HEAP_END_`, `MEMORY` and `?MEMORY_SIZE`) are assigned the value zero.
- Only free sections within those sections defined by the `ram` control or its default are searched for the largest section. If `HEAP` is used as a module with the `ram` control, only this section is searched for the largest section for HEAP space. This limit implies if the `ram` control of the last linkage, during an incremental link, does not include all the sections included in the previous stages, it is possible that the selected free RAM section is not the largest one.
- The public symbols `_HEAP_START_` and `_HEAP_END_` appear in module `_HEAP_` and file `<Dummy>`. The public symbols `MEMORY` and `?MEMORY_SIZE` appear in module `<Dummy>` and file `<Dummy>`. These four symbols are printed in the symbol table listing, in the intermodule cross-reference listing, and in the error messages, like any other public symbol.

- If you specify a public symbol with the name `_HEAP_START_`, `_HEAP_END_`, `MEMORY` or `?MEMORY_SIZE` and `absstack` is in effect, the linker issues the warning `MULTIPLE PUBLIC DEFINITION`. The symbol definition supplied by your module is the dominant one.



The `absstack` control must be in effect during the final linkage. RL196 automatically supplies the `_HEAP_START_`, `_HEAP_END_`, `MEMORY` and `?MEMORY_SIZE` symbols during this time. However, the section indicated by these four symbols are not allocated. If you perform another link after `absstack` is already in effect, the linker issues an error and the output is unusable. You must ensure that `absstack` is in effect only at the last linkage.

### **2.5.5 OVERLAPPING ROM AND RAM MEMORY USING THE INST PIN**

The addressable memory space on the 80C196 family of components consists of 64 kilobytes, mostly available for code or data memory. The instruction (INST) pin present in most 80C196 components, except for the 48-pin 8096 component, was originally reserved for the use of development tools, but can now be used to expand the 64-kilobyte memory limitation.

The INST pin is active high during processor bus cycles that read instructions from memory. Fetching an instruction from memory is also referred to as performing an opcode fetch. The processor then drives the INST pin low during cycles that read or write data to memory locations.

The following discussion presents additional information on the behavior of the INST pin and on how the logical level of the INST signal relates to program activities, such as fetching opcodes and accessing vector tables. This discussion includes an example memory expansion scheme that overlaps code and data. The example provides guidelines for hardware and software development and includes a sample RL196 invocation line.

### **2.5.5.1 INST PIN BEHAVIOR**

Consider the following when developing an 80C196 application that uses the INST pin for overlapping code and data:

- **Fetching Opcodes from Memory.** The INST pin is high during processor cycles that fetch instructions from memory. The first time the INST pin goes high after reset is during the opcode fetch at 2080H.
- **Accessing the Chip Configuration Byte (CCB).** When the 80C196 component is reset, the chip configuration register (CCR) is loaded with the contents of memory location 2018H, the CCB. The CCB is read as data; therefore, the INST pin is low when address 2018H is valid.
- **Accessing the Interrupt Vector Table.** If the component hardware detects an interrupt, the corresponding bit in the interrupt pending register is set. If you enabled the interrupt enable bit (EI) in the PSW and you loaded the interrupt mask with a value that allows execution of that particular interrupt, the processor reads the interrupt vector as data from the interrupt vector table. The interrupt vector table occupies locations 2000H to 2013H for the 8096; and additionally, 2030H to 203FH for the 80C196. The INST pin is low when these addresses are valid.
- **Accessing Program Constants and Variables.** Constants are read as data and therefore, the INST pin is low. The constants should be located in CONST segments. CONST is a new segment type in OM96. The C196 compiler supports this by putting constants in a CONST segment. The INST pin is also low during write operations to variables.
- **Program Access to Vector Tables.** This only applies if C196 'old object' is used. The C196 switch statements sometimes creates, so called, vector tables, to transfer program control. The processor reads the vector value as data thus forcing the INST pin low. These vector tables of data are stored in the ROM memory space, therefore burned into the PROM.

### **2.5.5.2 OVERLAPPING MEMORY SCHEME EXAMPLE**

This example demonstrates how to create an overlapping code and data memory scheme. The accompanying discussion includes software and hardware considerations needed to create a system with 112 kilobytes of memory (56 kilobytes of ROM, 56 kilobytes of RAM).

The memory map of the 112 kilobyte system (shown in Figure 2-1) decodes the INST pin for only part of the memory scheme. The system does not overlap the total memory space using the INST pin for the following reasons:

- The INST pin is not decoded between addresses 2000H and 2080H for two reasons. First, special locations, some of which are read as data and some of which are fetched as opcodes, can reside in adjacent memory without the necessary hardware overhead if the INST pin were decoded. Second, reserved locations reside in this address range.

### **2.5.5.3 GUIDELINES FOR HARDWARE DEVELOPMENT**

Figure 2-1 depicts the example system. This system provides 16 kilobytes of memory (00H to 3FFFH) not decoded by the INST pin (INST is don't care) and 48 kilobytes of overlapping code and data memory (4000H to 0FFFFH). The Boolean expressions describing the memory scheme are shown in the following notations. The overbar indicates active low.

$$\text{RAM1} = \overline{\text{A15}} * \overline{\text{A14}} * \overline{\text{A13}}$$

$$\text{ROM1} = \overline{\text{A15}} * \overline{\text{A14}} * \text{A13}$$

$$\text{RAM3} = (\text{A15} + \text{A14}) * \overline{\text{INST}}$$

$$\text{ROM3} = (\text{A15} + \text{A14}) * \overline{\text{INST}}$$

$$\text{ROM2} = (\text{A15} + \text{A14}) * \text{INST}$$

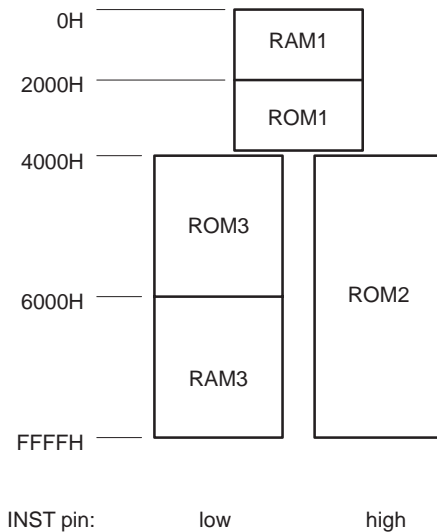


Figure 2-1: 112-Kilobyte overlapping code and data memory map

#### 2.5.5.4 LINKER/LOCATOR EXAMPLE INVOCATION LINE

The following RL196 invocation line can best be put in a Makefile to avoid having to retype it each time RL196 is invoked. See Section 2.7 for more information on how to use the **mk196** utility.

You can use the `romcode` and `romdata` controls instead of the `rom` control. These controls can be very useful when using the 80C196KR or 24-bit processor.

The RL196 invocation line is as follows:

```
rl196 cstart.obj, main.obj, mod1.obj, mod2.obj, mod3.obj,
      c96.lib, fpal96.lib
      to applix.out
      ram(100H-1FFFH, 6000H-0FFFFH)
      rom(2000H-3FFFH(mod2,mod3(const))),
      romcode(4000H-0FFFFH(main,mod1,mod3))
      romdata(4000H-5FFFH(mod1)),
      inst
```

In this example we have decided to put the constants from mod3 and both the code and constants of mod2 in ROM1 memory; the code from mod1 and mod3 are put in ROM2 memory and the constants from mod1 are put in ROM1 memory.

Place the RL196 invocation line in a batch file to avoid having to retype it each time RL196 is invoked. See Section 2.7 for more information on how to create DOS batch files.

### **2.5.5.5 SUMMARY OF RL196 INST USAGE**

Key things to remember during development of overlapping code and data applications are:

- Constants in C196 programs are put in a separate segment of type CONST.
- ASM196 modules can also make use of the CONST segment.

### **2.5.6 SUPPORT FOR VERTICAL WINDOWS**

The 80C196KC and the 80C196KR processors have 256 additional registers from 100H through 1FFH (other processors can have more). Register windowing was implemented so that the additional registers can be accessed using the 8-bit direct addressing mode instead of the 16-bit addressing mode, resulting in faster and tighter code generation. The two types of windows are Horizontal Windows (HWindows) and Vertical Windows (VWindows). This section focuses on Vertical Windows. See the *Embedded Microcontrollers and Processors Handbook*, listed in *Related Publications*, for more information on register windowing.

The 80C196KC, 80C196KR and the 24-bit processors provide vertical windowing so that the additional bytes of RAM can be used as general purpose registers. You can access these registers using the 8-bit direct addressing mode. VWindows differ from HWindows in that you can still access these registers through 16-bit addressing using the indexed or indirect addressing mode since VWindows reside in the address space. You can use VWindows to map sections of the 512 bytes of registers from 00H – 1FFH into 32-, 64-, or 128-byte windows onto the top 32-, 64-, 128-byte block (the upper portion) of the register file. Use the Window Select Register (WSR) to switch between windows.

The C196 compiler uses the additional registers for the block-scope register variables allocated in overlay segments. Block-scope variables are variables declared within non-reentrant functions. For ASM196 modules, RL196 assists in locating the overlay segments in the vertical window, provided you add the WSR management code in all public functions which intend to use the VWindow registers as block-scope variables. Figure 2-2 shows the register allocation scheme that the linker uses to locate register and overlay segments on the 80C196KC processor.

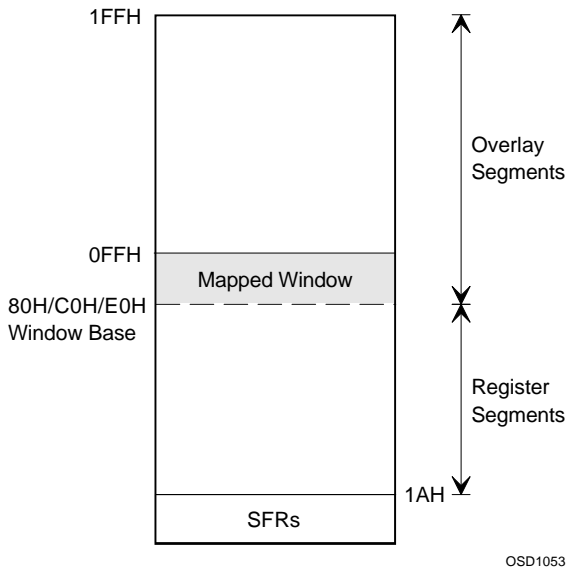


Figure 2-2: 80C196KC register allocation scheme

RL196 first locates the global variables allocated in register segments in the available register space below the window base selected, in the lower 256 registers, during link-time. This scheme allows access to a global variable without needing to swap the WSR. Then, the linker locates the overlay segments after all of the register segments are located. If gaps are between register segments, the linker attempts to fill the gaps with overlay segments of the right size. The linker selects the window size based on the last (highest) address occupied by the last register segment. The last occupied address must fall below 80H (128-byte window), 0C0H (64-byte window), or 0E0H (32-byte window); otherwise, the linker sets WSR to zero, takes no action on the additional registers, and generates a warning stating that there are too many registers. The order of the memory allocation scheme changes when you are performing incremental links.



For incremental links, when `noabsstack` is in effect, RL196 selects the smallest window size (32 bytes) to make sure that the linker has enough memory for all of your application's register segments, unless you have specified a window size with the `window_size` control in your link invocation. The linker takes the specified size into account and locates the register segment of the module. If the register segment fits, the linker locates the overlay segment starting from window base address of your requested window size. If the register segment does not fit under the window base, the linker issues a warning stating that the window size you specified is too large for your application, uses the smallest window size possible (32 bytes), then locates your overlay segment at 0E0H.

When linking modules together, use the RL196's `registers` control to specify the range of the available registers on the target component and the `window_size` control to specify the desired window size. See Section 2.8 for more information on these controls.

## 2.6 INVOKING RL196

The general syntax for the invocation line is:

```
[pathname] rl196 [options] input_list [to output_file] [controls]
```

Where:

- pathname* is the device and/or directory where RL196 resides.
- options* is an optional list of one or more options. See Section 2.6.1 for a detailed description of each option.
- input\_list* is a list of one or more elements separated by commas. An element can be an ordinary object file, an object library file, or a `publiconly` list. See Section 2.6.2 for more details.
- output\_file* is the (optional) file that receives the output object module. See Section 2.6.3 for more details.
- controls* is an optional list of one or more controls. See Section 2.8 for explanation of each control.



When you are using EDE, you can specify linker/locator options and controls in a graphical way in the **EDE | Linker Options...** menu item.

The next several sections give details of the elements of the RL196 invocation command.

### 2.6.1 OPTIONS

The format for a single option is:

*-option\_name* [ {*=* | *:* | *space*} *argument* ]

Where:

- (minus sign) must be prefixed to every option name.
- option\_name* is the name of the option. This name is case sensitive.
- =*, *:* or *space* are used to separate the option name from the argument.
- argument* the argument for an option. This is optional.

Some options can toggle conditions on or off. To turn a condition off, you have to append a minus sign to the option name, as in **-case** versus **-case-** or **-M** versus **-M-**.

Most of the options can also be set using controls. However, some options have no equivalent control. Also some controls have no equivalent option. When equivalent controls and options exist, they default to the same value.

The following table is a list of all options and their equivalent control (if present). For a detailed description of each option that has an equivalent control, refer to the description of the control in Section 2.8. Options that have no equivalent control are described below.

Option	Control	Description
-?		Display invocation syntax
-L	searchlib	Specify alternative search path for input files
-M[-]	[no]print	Specify name of map file
-QW[-]	[no]quietwarns	Prevent warnings from appearing on the screen
-S[-]	[no]sfr	Specify to include model specific SFR file
-V		Display version header only

Option	Control	Description
-W[-]	[no]warning	Specify to exit the linker with a non-zero value when there are one or more warnings
-as[-]	[no]absstack	Specify to locate the stack absolute
-bu[-]	[no]bottomup	Allocate addresses from low to high
-case[-]	[no]case	Linker works case sensitive
-ch[-]	[no]code2high	Convert CODE segments to HIGHCODE
	[no]dataoverlay	Specify data segment to overlay
	farcode	Specify code space configuration for 24-bit models
	farconst	Specify constant space configuration for 24-bit models
	fardata	Specify data space configuration for 24-bit models
-he[-]	[no]heap	Generate HEAP space
-f <i>file</i>		Read options and/or controls from <i>file</i>
-ia[-]	[no]ignoreabs	Ignore warnings about absolute segments
	[no]inittable	Generate initialization table
-in[-]	[no]inst	Allow ROM and RAM addresses to be overlapped
-ix[-]	[no]ixref	Generate symbol cross-reference table in map file
-lb[-]	[no]limit_bitno	Do not allow bit number greater than 7
	[no]list	Specify elements to be included in map file
-md	model	Specify processor model
	name	Specify module name (does not affect the output filename)
	nearcode	Specify code space configuration for 24-bit models
	nearconst	Specify constant space configuration for 24-bit models
	neardata	Specify data space configuration for 24-bit models
	[no]np_rsvup6	Reserve upper six bytes of every page (model(np) and model(nu) only)
-o	to	Specify name of output file

Option	Control	Description
-omf	omf	Specify OMF96 version to generate
	[no]pageprint	Print all code addresses in compatibility mode as 24 bits
-pw	pagewidth	Set map file page width
	[no]purge	Specify elements to be excluded from object file
	ram	Specify designated RAM sections
	registers	Specify register range available to application
-rf[-]	[no]regfirst	Specify to locate register segments first
	[no]regoverlay	Specify register to overlay
	rom	Specify designated ROM (CODE and CONST) sections
	romcode	Specify designated CODE sections
	romdata	Specify designated CONST sections
-ss	stacksize	Specify size of stack segment
-tc[-]	[no]typecheck	Perform type checking
-um[-]	[no]uniquemods	Allow more than one module with the same name
-ws	windowsize	Specify window size for vertical windowing

Table 2-3: RL196 linker options

Below are the detailed descriptions of the options.

- ?** Display an explanation of options on `stdout`.
- V** Display version information on `stdout` and stop.
- f file** Use *file* for command line processing. In this way you can extend the command line. This option can be used more than once, even between the controls.

The control file can contain all of the object module filenames and controls to be processed by the linker. Use this option when your command line becomes too long or as an alternative for using a command file. See Section 2.7.3 for more information on command files.

You can specify other invocation elements, such as filenames and controls, before or after the **-f** option. If you do, make sure that the contents of the control file, when appended to the command line, validly completes the invocation line. See example 2 below for an example of an invalid command line.

The control file has a character limit of 6 kilobytes. The file can contain multiple lines separated by newlines, as shown in the following example:

```
prog1.obj, prog2.obj,
lib1, lib2(mod1,mod2)
to exe_file
stacksize(+100h)
```

### ***Examples***

1. The following example appends a control file called `link.lst`.

```
r1196 -f link.lst
```

The file `link.lst` contains the following:

```
mod1.obj, mod2.obj noas
```

2. For the following example, the control file, `link.lst`, contains the following:

```
, prog2.obj, c96.lib
```

When invoked with the following command line, the linker generates an error:

```
r1196 prog1.obj, cstart.obj, ss(+2h) -f link.lst
```

This is because the resulting invocation string is as follows:

```
r1196 prog1.obj, cstart.obj, ss(+2h), prog2.obj, c96.lib
```

## **2.6.2 INPUT LIST**

The input list tells RL196 the names of the files to be processed. The name of each input file must be unique but it can have the same name as the output object module. However, neither the input nor output module can be called `stack` or `st`.

An element in the input list can be an ordinary object file, a library file, or a `publiconly` object file.

RL196 searches for the file in several places until it is found. The linker tests the path prefixes in the following order:

1. The current directory (no prefix).

For each directory under 2., 3., and 4., RL196 first searches in a model specific subdirectory before the directory itself is searched.

2. The directories specified by the `searchlib` control.
3. The directories in the `C196LIB` environment variable, if defined.
4. The `lib` directory, one directory higher than the directory containing the **rl196** binary. For example, if **rl196** is installed in `/usr/local/c196/bin`, then the directory searched for the object files and library files is `/usr/local/c196/lib`.

You can specify a library file with or without an explicit module list. The following sections describe each of these possibilities.

### **2.6.2.1 ORDINARY OBJECT FILE**

The syntax is:

*[pathname]filename.ext*

Where:

*pathname* is the complete name of the device and/or directory where your object file resides.

*filename.ext* combined with the *pathname*, can be no more than 64 characters long. The extension depends on the output name you assigned during translation. The default is `.obj`.

An ordinary object file is usually either the result of a translation of a source program or is an earlier output of RL196. The file consists of a sequence of object modules included in the output object file, although the file usually contains only one module. LIB196 can also produce an ordinary object file (see Chapter 4), containing one or more modules, which is processed unconditionally, module by module.

The following example specifies an ordinary object file:

```
rl196 main.obj, sort.obj, debug.obj to sort.out
```

Here the output file, `sort.out`, includes the modules contained in `main.obj`, `sort.obj`, and `debug.obj`.

### **2.6.2.2 OBJECT LIBRARY FILE**

The syntax is:

```
lib_filename [(module_name [...])]
```

Where:

*lib\_filename* is the name of the library where *module\_name* resides.

*module\_name* is the name of the module to be included in the link process unconditionally.

The LIB196 librarian creates and maintains an object library file, as explained in Chapter 4. The file consists of a set of object modules. RL196 selects and includes these object modules into the output file if they contain public definitions that resolve at least one external reference.

If you attach a parenthesized list of modules to the library filename, RL196 includes the selection unconditionally, as in the following example:

```
rl196 main.obj, sort.obj, debug.obj,  
      intrpt.lib(int0, int3, int4)
```

Here the output file includes the modules contained in `main.obj`, `sort.obj`, `debug.obj`, and the modules `int0`, `int3`, and `int4` from the library file `intrpt.lib`.

If a module list is not attached to the library file, RL196 includes only those modules containing public definitions that resolve previous external references into the output file. This process is iterative; that is, a selected module can contain unresolved external references, requiring another library file scanning cycle to find other modules that resolve these references, and so on. A library is distinguished from an ordinary object file not by its extension (`.lib`) but by its internal structure.

The following example specifies an object library file:

```
rl196 main.obj, sort.obj, debug.obj,  
      intrpt.lib(int0, int3, int4), c96fp.lib,  
      c96.lib, fpal96.lib
```

In addition to the modules selected in the previous example, the output file includes modules selected from the libraries `c96fp.lib`, `c96.lib` and `fpal96.lib`.

You must consider the order in which you link the library object files. The linker only makes one pass through each library file to resolve any external reference. To ensure all references can be resolved, link all library files last in your input list. See the chapter on library files in the *80C196 C Compiler User's Guide*, listed in *Related Publications*, for more information on linking libraries and object files.

### **2.6.2.3 PUBLICONLY OBJECT FILE**

The syntax is:

```
publiconly(filename [...])
```

Where:

*filename* is the object file that contains the absolute public symbols.

Abbreviation: `po`

A `publiconly` object file is an ordinary object file that is used solely to resolve external references. Thus, RL196 extracts only the absolute public symbol definitions from these files. This file is processed sequentially, module by module.

For example, assume that a piece of code is already located at a certain address space, such as some routines on external ROM, and that a new module refers to that existing code. When you link the new module, specify the object file of the existing code as a `publiconly` file in the input list of the invocation line. The `publiconly` file resolves the references made to that code. If you did not specify the existing object file as a `publiconly` file, an additional copy of that code is included in the new object module.



The following example specifies a `publiconly` object file:

```
rl196 main.obj, sort.obj, debug.obj,  
      intrpt.lib(int0, int3, int4),  
      c96fp.lib, c96.lib, fpal96.lib,  
      publiconly(monitor.abs)
```

The output object file includes the same set of modules as in the previous example. The file, `monitor.abs`, supplies the absolute addresses of its absolute public symbols that are referenced by the other modules.

If the first input file does not contain an extension, a fatal error occurs unless you assign an output filename by using the `to` keyword. For example, the following command is illegal because the output filename defaults to `main`:

```
rl196 main
```

But the same example is legal if you append the `to` keyword with the desired output filename after the input filename:

```
rl196 main to main.96
```

### **2.6.3 OUTPUT FILES**

RL196 produces three outputs: screen messages, a print file, and an object file. Screen messages are discussed in Chapter 9. Use the listing controls to choose the information that appears in the print file.

#### **2.6.3.1 PRINT FILE**

The *filename.m96* (default extension) print file produced by RL196 consists of:

- a sign-on message (as displayed on the screen)
- a link summary
- an optional symbol table
- an optional intermodule cross-reference listing
- a list of error messages
- a sign-off message (as displayed on the screen)

### ***Link Summary***

Figure 2-3 shows a sample link summary. The link summary includes the following information:

- An invocation command summary.
- A list of input modules included in the output object file.
- The segment (or link) map that lists all allocated segments, giving their type, base address, length, alignment, and the module within which the segment was defined. The segment map also shows the segment overlaps and the gaps in the memory space. When an absolute stack segment is created, the linker specifies no module name. If the stack is still relocatable, the corresponding line starts with a **\*\*\*REL\*\*\*** indication, and no base is given.
- A list of all unallocated segments. In such a case, RL196 reports the reason for the unallocated segments by generating a specific error message for each case.
- A list of unresolved externals, when any symbol is left unresolved. RL196 reports each occurrence of an unresolved external symbol in a module with a specific error message.

To suppress the segment map, omit the `segments` option from the `list` control, for example, `list(symbols,lines)`.

```
80C196 relocater/linker vx.y rz SN000000-006 (c) year TASKING, Inc.
(C)1983,1990,1993 Intel Corporation
INPUT FILES: mn.obj, mn1.obj, mn2.obj, cstart.obj, c96.lib
CONTROLS SPECIFIED IN INVOCATION COMMAND:
    li ix ov(mn1,mn2) noas pu(ln,sb,pl) ro(2080H-3fffH) pw(80)
OUTPUT FILE: mn.abs
```

#### INPUT MODULES INCLUDED:

```
mn.obj(mn) 15-Feb-96 16:20:15, C196 v99.9 md(bh)
mn1.obj(mn1) 15-Feb-96 16:21:14, C196 v99.9 md(bh)
mn2.obj(mn2) 15-Feb-96 16:20:18, C196 v99.9 md(bh)
../lib/kb/cstart.obj(STARTUP) 14-Feb-96 14:14:57
../lib/kb/c96.lib(_strlen) 14-Feb-96 14:12:27
../lib/kb/c96.lib(_tmpreg0) 14-Feb-96 14:14:38
../lib/kb/c96.lib(_fram01) 14-Feb-96 14:14:40
../lib/kb/c96.lib(_main) 15-Feb-96 15:17:38, C196 v99.9 md(kb)
../lib/kb/c96.lib(__exit) 14-Feb-96 14:12:55
../lib/kb/c96.lib(_imain) 14-Feb-96 14:14:29, C196 v99.9 md(kb)
```

```

SEGMENT MAP FOR mn.abs(mn):
      TYPE  BASE  LENGTH  ALIGNMENT  MODULE NAME
      ----  -
**RESERVED*
      REG   001AH 0002H  WORD        _fram01
      REG   001CH 0008H  ABSOLUTE    _tmpreg0
      OVRLY 0024H 0008H  DOUBLE WORD mn2
**OVERLAP** OVRLY 0024H 0010H  WORD        mn1
*** GAP ***
      REG   0034H 001CH
      REG   0050H 0002H  ABSOLUTE    mn1
*** GAP ***
      REG   0052H 001EH
      REG   0070H 0002H  ABSOLUTE    mn1
      DATA 0072H 0190H  WORD        mn2
*** GAP ***
      CODE  0202H 1E7EH
      CODE  2080H 0009H  ABSOLUTE    STARTUP
      CONST 2089H 0012H  BYTE        mn2
      CODE  209BH 00EEH  BYTE        _imain
      CODE  2189H 005DH  BYTE        mn1
      CODE  21E6H 003EH  BYTE        mn2
      CODE  2224H 0010H  BYTE        _strlen
      CODE  2234H 0010H  BYTE        _main
      CODE  2244H 000EH  BYTE        mn
      CODE  2252H 0007H  BYTE        __exit
*** GAP ***
      CONST 2259H 0001H
      CONST 225AH 0010H  WORD        _STARTUP_DATA_
*** GAP ***
      226AH DD96H
*** REL *** STACK 0042H WORD

```

UNRESOLVED EXTERNAL SYMBOLS:

```

    fil_initialize
    fil_get_name

```

*Figure 2-3: Sample link summary*

### **Symbol Table**

The optional symbol table displays information on public symbols, local symbols, and source lines, as specified by the `publics`, `symbols`, and `lines` options of the `list` control. In addition, if `symbols`, `lines`, or both are specified with the `list` control, information on source modules and blocks are included.

The symbol table always begins on a new page. Figure 2-4 shows a sample symbol table.

## SYMBOL TABLE FOR EXAMPL(MN):

ATTRIBUTES	VALUE	NAME
-----	-----	-----
PUBLICS:		
CODE	VPL_PROC	2244H main
REG	INTEGER	0050H reg1
REG	INTEGER	0070H reg2
CODE	VPL_PROC	2189H mn1_1
CODE	VPL_PROC	2197H mn1
CONST	ARRAY	2089H copyright
DATA	ARRAY	0072H ptr
CODE	VPL_PROC	21E6H mn2
CODE	ENTRY	2080H cstart
CODE	ENTRY	2087H _exit
CODE	VPL_PROC	2224H strlen
REG	LONG	001CH PLMREG
REG	NULL	001CH TMPREG0
REG	WORD	001AH _FRAME01_
REG	WORD	001AH ?FRAME01
CODE	VPL_PROC	2234H _main
CODE	VPL_PROC	2252H __exit
CODE	VPL_PROC	209BH _imain
CONST	BYTE	0000H _INIT_TABLE_START_
MODULE: mn		
MODULE: mn1		
MODULE: mn2		
LINE#:		
FILE: mn2.c		
	21E6H	6
	21F7H	11
	2204H	13
	2215H	14
CONST	ARRAY	2089H copyright
DATA	ARRAY	0072H ptr
	21E6H	PROC: mn2
OVRLY	INTEGER	0028H a
OVRLY	INTEGER	002AH b
OVRLY	LONGINT	0024H c
DYNAMIC	ARRAY	0002H d
MODULE: STARTUP		
MODULE: _strlen		
MODULE: _tmpreg0		
MODULE: _fram01		
MODULE: _main		
MODULE: __exit		
MODULE: _imain		

Figure 2-4: Sample symbol table

Each entry in the symbol table consists of the following three parts:

ATTRIBUTES Consists of the following three fields:

- The segment type. Indicates the kind of segment to which the symbol belongs. Possible segment types are NULL, CODE, FARCODE, DATA, FARDATA, CONST, FARCONST, STACK, REG, OVERLAY and DYNAMIC.
- The symbol type. For compound data types it gives only an indication of the data type. The symbol types are BYTE, WORD, LONG, ENTRY, REAL, NULL, BIT, ENUM, UNION, SCALAR, SHORTINT, INTEGER, LONGINT, UNSGN\_INT, SGN\_INT, POINTER, PTR, FARPTR, WSR\_PTR, ARRAY, STRUCTURE, LIST, LABEL, WHOLE, PROCEDURE, FPL\_PROC, and VPL\_PROC.

The type ENTRY stands for both labels and procedures/functions in PL/M-96 and C196 as well as the ENTRY of ASM196.

If the linker does not recognize the symbol type, the linker prints the type index of that symbol in the print file and encloses the index in quotation marks: for example, "73". The type index points to a type representation.

- The symbol base. If the symbol is pointed to by another symbol, also called the base, the token BASED appears in this field. In addition, the segment type and the value fields must match the base.

VALUE This field contains the absolute address of the symbol unless the field is one of the following special cases. If the segment type is NULL, this field contains the value of the associated symbol. If the segment type is DYNAMIC, the field contains the symbol offset from the contents of the frame pointer of the procedure where the symbol was defined. If the symbol is still relocatable, the token REL appears in this field.

NAME The name of the module, procedure, do block, public symbol, local symbol, or a line number in decimal. In this field, an indention indicates scope. Public symbols are preceded by the key word PUBLICS on a separate line; line numbers are preceded by the keyword LINE#: and the keyword FILE: on a separate line; a module name, a procedure name, and a block name follow the keywords MODULE:, PROC:, and BLOCK:, respectively.

Not all symbols contain values in all fields. For example, only the name of a module symbol is shown.

A block can be an unnamed block. In this case, the name field remains empty.

The scope rules do not relate to line numbers. RL196 prints line numbers to the symbol table on the fly using the current indentation level. The line numbers that appear in the print file depend on how the translator counts each line in the source file. For example, PL/M-96 counts and assigns line numbers to only non-blank executable lines while C196 counts every line in the source file.

### ***Intermodule Cross-reference Listing***

The optional intermodule cross-reference listing includes an entry for each global symbol. RL196 produces the intermodule cross-reference listing if you specified the `ixref` control during linkage. This section of the listing always begins on a new page. The symbols in the listing are listed alphabetically. Each entry contains the following three fields:

**NAME**            The name of the symbol.

**ATTRIBUTES** Consist of the following two fields:

- The segment type, which indicates the segment type to which the symbol belongs. Possible segment types (NULL, CODE, FARCODE, DATA, FARDATA, CONST, FARCONST, STACK, REG, OVERLAY) appear exactly the same as the segment type field in the symbol table, except that DYNAMIC is not allowed.
- The symbol type. This field is handled exactly the same as the symbol type field in the symbol table.

**MODULES**        The name of all modules in which the symbol is declared either as public or external. The module name in which the symbol was defined as public appears as the first entry. If the symbol was declared as public in multiple modules, the linker then lists all of the module names. The remaining entries are the alphabetically sorted module names in which the symbol was referenced as external. The linker lists only those module names that actually used the symbol. No module name appears when the symbol is unresolved. The linker prints the string **\*\*UNRESOLVED\*\*** instead.

Figure 2-5 shows a sample intermodule cross-reference listing.

```
INTERMODULE CROSS-REFERENCE LISTING:

NAME          ATTRIBUTES          MODULES
----          -
__exit ..... CODE    VPL_PROC    ; __exit _main
_FRAME01_ ..... REG   WORD        ; _fram01
__exit CODE .... ENTRY           ; STARTUP __exit
_ismain ..... CODE    VPL_PROC    ; _ismain _main
_main ..... CODE      VPL_PROC    ; _main STARTUP
?FRAME01 ..... REG    WORD        ; _fram01 _ismain mn2
PLMREG ..... REG      LONG        ; _tmprg0
TMPREG0 ..... REG     NULL        ; _tmprg0 __exit _ismain
                                   _main _strlen
                                   mn mn1 mn2

copyright ..... CONST ARRAY      ; mn2 mn1
cstart ..... CODE      ENTRY      ; STARTUP
fil_get_name ... CODE     VPL_PROC  ; ** UNRESOLVED ** mn2
fil_initialize . CODE     VPL_PROC  ; ** UNRESOLVED ** mn1

main ..... CODE      VPL_PROC    ; mn _main
mn1 ..... CODE      VPL_PROC    ; mn1 mn
mn1_1 ..... CODE     VPL_PROC    ; mn1
mn2 ..... CODE      VPL_PROC    ; mn2 mn
ptr ..... DATA     ARRAY        ; mn2
reg1 ..... REG       INTEGER     ; mn1
reg2 ..... REG       INTEGER     ; mn1
strlen ..... CODE     VPL_PROC    ; _strlen mn1 mn2

WARNING 2: Unresolved external symbol: fil_initialize in
mn1.obj(mn1)
WARNING 2: Unresolved external symbol: fil_get_name in mn2.obj(mn2)
WARNING 4: Reference made to unresolved external: fil_initialize
           in mn1.obj(mn1), at CODE(0034h)
WARNING 4: Reference made to unresolved external: fil_get_name
           in mn2.obj(mn2), at CODE(0026h)

RL196 COMPLETED,    4 WARNING(S),    0 ERROR(S)
```

Figure 2-5: Sample intermodule cross-reference

Error Messages

Error messages appear at the end of the print file. RL196 error messages are categorized as warnings, errors, and fatal errors. Only fatal errors terminate the RL196 operation. An error does not terminate operation but the resulting output module might be unusable. If so, the linker marks it as such. A warning indicates a detected condition that might not be what you wanted.



See Chapter 9 for a complete list of RL196 error messages of all types and their probable causes.

### **2.6.3.2 OUTPUT OBJECT FILE**

RL196 generates an output absolute or quasi-absolute object file that contains:

- The content of the program segments. This information is always present in the RL196 output object file.
- Debug information, if `debug` is in effect during the translation time and `nopurge` is in effect during the link/locate process. This information includes symbols, line numbers, and name-scoping. Name scope determines the context in which user-defined names occur. Your object code must contain debug information if you plan on debugging your code with an emulator.
- Linkage support information, if `nopurge` is in effect during the link/locate process. This information includes segment definitions, a list of (now absolute) public definitions, and a list of unresolved external symbols.



To prevent any inappropriate error messages from the PROM programmer during loading of an 80C196 absolute object file, specify the `purge` control during the final linkage.

To assign a name to the output object file, use the `to` keyword. For example, the line, `rl196 main.obj to main.abs`, assigns `main.abs` as the output object file. If you omit the output filename, RL196 creates a filename for the output file by removing the extension from the first filename in the input list and using only the path and root name.

If the target drive already has a file with the name of the output file, RL196 overwrites the existing file with the new output file.



### ***Creating a Final Absolute Object File***

Use the final absolute object file for programming the 8096 ROM/EPROM or for debugging the application. To create a final absolute object file, set the `absstack` control, either explicitly or as the default. Because the program no longer needs public and segment information at this point, use the `purge(segments, publics)` control to conserve space. If no debugging is required, you can also purge the local symbols and line number information. Because `nopurge` is the default, `purge` must be specified. All external references must be resolved to prevent run-time errors.

### ***Creating a Quasi-absolute Object File***

If the output file is to be reused by RL196, that is the final absolute file is produced after an incremental linkage, you must specify `noabsstack` in the RL196 invocation. Additionally, the segments and public symbols information cannot be purged. In this mode, the output file can still contain unresolved external references that can be resolved by subsequent executions of RL196.

## **2.7 AUTOMATICALLY INVOKING MULTIPLE COMMANDS**

TASKING offers three ways of automatically invoking a series of commands: makefiles, batch files and command files. This section demonstrates ways to use these features with TASKING software development tools. Filenames and directory names appearing in this section are examples.

### **2.7.1 USING MAKE UTILITY MK196**

**mk196** takes a file of dependencies (a 'makefile') and decides what commands have to be executed to bring the files up-to-date. These commands are either executed directly from **mk196** or written to the standard output without executing them.



For a detailed description of this utility, see Chapter 8.

### **2.7.2 USING BATCH FILES**

A batch file contains one or more commands that DOS executes one at a time. A batch file can contain commands valid at the DOS command line prompt and commands valid only within a batch file. All batch files must have the extension `.bat`.

You can pass parameters to a DOS batch file during invocation, so that the batch file can do similar work on a different program or on a set of data each time it executes. In the following example, the batch file `96a.bat` contains a command sequence invoking the ASM196 assembler. Any assembler source filename with the extension `.a96` can be a parameter to `96a.bat`, so `96a.bat` can assemble different source files. DOS replaces the parameter `%1` with the DOS filename of the assembler source file, in this example `prog1`.

1. Create a DOS batch file named `96a.bat` containing the following line:

```
c:\c196\bin\asm196 %1.a96
```

2. Invoke the batch file by typing the name of the batch file, without specifying the `.bat` extension, followed by the name of the source file to be assembled, without specifying the `.a96` extension.

#### **96a prog1**

When `96a.bat` executes, DOS replaces the `%1` with `prog1`, resulting in the command:

```
c:\c196\bin\asm196 prog1.a96
```

Other important characteristics of DOS batch files are as follows:

- In any version of DOS before Version 3.3, batch files cannot be nested. If a batch file invokes another batch file, control passes directly to the other batch file but does not return to the first batch file. Therefore, place any batch file invocation last in a batch file.
- Batch files can contain control flow constructs such as `if` and `goto`. For example, the following command allows the result of program execution from the previously executed batch file to determine which command in the current batch file executes next:

```
if errorlevel n goto label
```

The variable *n* contains the unacceptable error code. If the error code returned by the last batch file executed is the same as or greater than the value of *n*, control transfers to the line immediately after *label1*. The label is any alphanumeric string up to 8 characters. A longer label does not cause an error but only the first 8 characters are significant.

- To process continuation lines in DOS batch files, use redirect input from a file containing the continuation lines. DOS does not support continuation lines in batch files. Although a batch file can contain multiple DOS command lines, each command must fit on a single line.

In the following example, the batch file *96a1.bat* assembles an ASM196 source program, passes the resulting object module to RL196, and invokes OH196 to convert the final object module to hexadecimal format. RL196 uses the existing object files listed in *96a1.ltx*.

1. Create a DOS batch file named *96a1.bat*, containing the following lines:

```
echo off
echo. asml96 assemble and link:
asml96 %1.a96
rl196 %1.obj, -f %0.ltx
if errorlevel 1 goto lfail
oh %1
goto stop
:lfail
echo. failure at link step
:stop
```

Since *96a1.bat* and *96a1.ltx* have identical names except for the extension, *96a1.bat* can refer to *96a1.ltx* as *%0.ltx*. The DOS batch file parameter *%0* is a special parameter DOS always replaces with the name of the batch file, without the *.bat* extension, containing it.

2. Create a text file named *96a1.ltx* containing the following lines:

```
prog0.obj, progxs.lib,
a096l.obj, mylib96l.lib
```

3. Execute the batch file *96a1.bat* by typing the following at the DOS command prompt:

```
96a1 prog1
```

When `96a1.bat` executes, DOS invokes ASM196 to assemble `prog1.a96`, then invokes RL196 to link the resulting object module, `prog1.obj`, to the run-time libraries and object modules specified in `96a1.ltx`. The control flow constructs determine whether the failure message appears on the screen.

### 2.7.3 USING COMMAND FILES

You can invoke the DOS command processor, `command.com`, with input redirected from a file called a command file. A DOS command file contains a sequence of DOS commands and must contain the DOS command `exit` as its final line. See your DOS manual for explanations of the DOS commands `command` and `exit`.

For example, create a command file named `make96.cmd` containing the following:

```
asml96 prog0.a96
asml96 prog1.a96
rl196 -f link_obj.lst
exit
```

The `link_obj.lst` file is a control file used by the `-f` option. See Section 2.6.1 for more details. This file contains the following:

```
progxsl.obj, prog0.obj, prog1.obj, mylib96l.lib,
progxs.lib
```

You can redirect the commands in `make96.cmd` to `command.com` by typing the following at the DOS prompt:

```
command < make96.cmd
```

`Command.com` then invokes all commands listed in the file `make96.cmd`.

The following considerations apply when invoking `command.com` with input redirected from a command file:

- This method of redirecting commands works only for a command file containing a fixed sequence of commands. Parameters cannot be passed to `command.com`.
- If your command line becomes too long, use the `-f` option to append the contents of the control file to the command line. See Section 2.6.1 for more information on the `-f` option.

- `Command.com` does not recognize the DOS batch file commands `if` and `goto`. Flow of control is always sequential, from top to bottom of the command file.
- Command files can be nested. If a command file reinvokes `command.com` with a secondary command file, control returns to the primary command file when the secondary command file exits. `Command.com` can be invoked from the primary command file with a line such as the following:

```
command < comfile2.cmd
```

The secondary command file must contain the command `exit` as its final line. If it does not, control does not return to the primary command file until `exit` is entered at the DOS prompt. When the command `exit` executes, control returns to the point in the primary file immediately following the point from which the secondary file was invoked.

If you redirect the output of a command file to a file, the command line interpreter records the following information in that file:

- All commands from the first line of the command file through the command `exit`.
- All console input and output.

For example, the following command invokes the command file `make96.cmd` and creates a log file named `make96.log`:

```
command < make96.cmd > make96.log
```

Nothing appears on the screen in response to this command except the DOS prompt following execution of the final `exit` command in `make96.cmd`. DOS writes all screen messages, including intermediate DOS prompts, to the log file, `make96.log`.

## 2.8 RL196 CONTROLS

All of the RL196 controls, modify the default operation of the linker. These controls fit into three functional groups:

### *Listing controls*

Listing controls specify what information is to be sent to the print file.

### *Linking controls*

Linking controls specify the name of the output module and determine what debug information is to be placed in the output object file.

### *Locating controls*

Locating controls specify the ROM, RAM, register sections, and the order in which some of the relocatable segments are to be allocated. In addition, you can manipulate the stack segment with these controls.

You can enter more than one control on the RL196 invocation line. Separate controls by spaces, not commas. If you enter the same control more than once, a fatal error results and RL196 aborts.

The RL196 controls are characterized by the following:

- Most of the controls have a parameter.
- Every control name has a two-character abbreviation.
- Most of the controls have a negative form created by placing the prefix `no` before the control name or its abbreviation. Do not attach a parameter to the negative form of a control.
- Every control has a default setting.

Table 2-4 lists the RL196 controls by group.

Group	Control Name	Abbreviation	Default
Listing	[no]ixref	[no]ix	noixref
	[no]list	[no]li	list(all) <sup>1</sup>
	[no]pageprint	[no]pp	pageprint
	pagewidth	pw	pagewidth(120)
	[no]print	[no]pr	print(file.m96) <sup>2</sup>
Linking	[no]case	[no]cs	case

Group	Control Name	Abbreviation	Default
Locating	[no]limit_bitno	[no]lb	nolimit_bitno
	name	na	name(mod_name) <sup>3</sup>
	[no]purge	[no]pu	nopurge
	[no]quietwarns	[no]qw	noquietwarns
	searchlib	sl	n/a
	[no]sfr	[no]sfr	nosfr
	[no]typecheck	[no]tc	typecheck
	[no]warning	wa	nowarning
	[no]absstack	[no]as	absstack
	[no]bottomup	[no]bu	nobottomup
	[no]code2high	[no]ch	code2high (24-bit models)
	[no]dataoverlay	[no]do	nodataoverlay
	[no]ignoreabs	[no]ia	ignoreabs
	[no]inittable	[no]it	inittable for omf(2) noinittable for omf(0) and omf(1)
	[no]inst	[no]in	noinst
	model	md	model(kb)
	nearcode/farcode	nc/fc	nearcode
	nearconst/farconst	nk/fk	nearconst
	neardata/fardata	nd/fd	neardata
	[no]np_rsvup6		np_rsvup6 <sup>4</sup>
	omf	omf	omf(2)
	ram	ra	ram(1AH–1FFFFH (stack)) <sup>5</sup>
	registers	rg	registers(1AH–0FFFH)
	[no]regfirst	[no]rf	noregfirst
	[no]regoverlay	[no]ov	noregoverlay
	rom	ro	rom(2000H–0FFFFH)
	romcode	rc	romcode(2000H–0FFFFH)
	romdata	rd	romdata(2000H–0FFFFH)
	stacksize	ss	stacksize( <i>total</i> ) <sup>6</sup>
	[no]uniquemods	um	nouniquemods

Group	Control Name	Abbreviation	Default
	window size	ws	window size(0) <sup>7</sup>
<b>Notes:</b> <div><div>1</div><div>The <i>all</i> placeholder stands for publics, symbols, lines, and segments.</div></div> <div><div>2</div><div>The <i>file.m96</i> placeholder indicates that the default is the first input filename followed by the extension .m96.</div></div> <div><div>3</div><div>The <i>mod_name</i> placeholder indicates that the default name is the name of the first input module.</div></div> <div><div>4</div><div>The <i>np_rsvup6</i> control can only be used with an NP model or NU model.</div></div> <div><div>5</div><div>The <i>ram</i> control default defines the RAM section and notes that the stack must be allocated as low as possible (i.e., in the register section as much as possible).</div></div> <div><div>6</div><div>The <i>total</i> placeholder indicates that the default stacksize is the total size of all stack segments from input modules. If <i>total</i> is less than 6, the default stacksize is 6.</div></div> <div><div>7</div><div>Indicates that no vertical windowing is used.</div></div>			

Table 2-4: RL196 controls

The remainder of this section explains each control in detail. The controls appear in alphabetical order. Some controls have an equivalent command line option which is also included in the syntax.

Square brackets ([ 1 ]) enclose optional arguments for controls. If you do not specify optional arguments for a particular control, do not use an empty pair of brackets.

Some controls use an optional list of arguments. Separate multiple argument definitions with commas. Brackets surrounding a comma and ellipsis ([ , . . . ]) indicate an optional list.

Curly braces ( { } ) indicate that you must pick one of the options provided. See Conventions Used in this Manual at the beginning of this manual for special meanings of type styles used in this manual.



With controls that can be set from within EDE, you will find a mouse icon that describes the corresponding action.



# absstack

## Function

Determines whether the stack segment of an output module is absolute.

## Syntax



Select the EDE | Linker Options... menu item. Enable or disable the Locate the stack segment as absolute check box in the Locating tab.



<code>absstack</code>		<code>noabsstack</code>
<code>-as</code>		<code>-as-</code>

## Abbreviation

`as` | `noas`

## Class

Locating control

## Default

`absstack`

## Description

Use this control to specify whether you want RL196 to absolutely locate the stack segment in the resultant RL196 output module or to leave the stack segment relocatable. If you specify `noabsstack` and the input stack segments are relocatable, the stack segment remains relocatable and can be expanded. The entire output module is quasi-absolute: all segments but its stack segment are absolute. The stack segment must stay relocatable if the output file is to be relinked.

When the `absstack` control is in effect, the linker supplies two public symbols that allow you to use free memory space for dynamic memory allocation.



See Section 2.5.4 for more information on how to do dynamic memory allocation.

**Example**

The following example produces no absolute stack:

```
r1196 mod1.obj, mod2.obj noas
```

# bottomup

### Function

Allocate low addresses first.

### Syntax



Select the EDE | Linker Options... menu item. Enable or disable the Locate bottom up (24-bit models: low addresses first) check box in the Locating tab.



bottomup		nobottomup
-bu		-bu-

### Abbreviation

bu | nobu

### Class

Locating control

### Default

nobottomup

### Description

Use this control to force allocation of low addresses first. Use this control for 24-bit models and segments FARCODE, CODE, FARDATA and FARCONST. Normally these will be filled high-to-low. With this control they will be filled low-to-high.

This control is especially useful when you have absolute segments in ROM.

### Example

The following example reverses the allocation:

```
r1196 mod1.obj, mod2.obj md(nt) bu
```

# case

## Function

Tells linker to act case sensitive.

## Syntax



Select the EDE | Linker Options... menu item. Enable or disable the Link case sensitive (required for C language) check box in the Linking tab.



case		nocase
-case		-case-

## Abbreviation

cs		nocs
----	--	------

## Class

Linking control

## Default

case

## Description

Use this control to tell the linker to work in a case sensitive manner. However, some general rules regarding case sensitivity must be considered:

1. Options supplied on the command line are always handled case sensitive.
2. Controls supplied on the command line are always handled case insensitive.
3. Keywords are always handled case insensitive.

When you use the nocase control:

4. All module names, public and external symbols are converted to upper case.
5. All filenames are converted to lower case.

When you use the default `case` control (or **-case** option):

6. None of the conventions mentioned in (4) or (5) is performed.

### Example

The following example turns case sensitivity off:

```
r1196 test.obj nocase
```

# code2high

## Function

Convert CODE segments to HIGHCODE.

## Syntax



Select the EDE | Linker Options... menu item. Enable or disable the Convert 16-bit CODE segments to 24-bit HIGHCODE segments check box in the Locating tab.



<code>code2high</code>		<code>nocode2high</code>
<code>-ch</code>		<code>-ch-</code>

## Abbreviation

`ch` | `noch`

## Class

Locating control

## Default

`code2high`

## Description

Use this control to convert CODE segments to HIGHCODE. You can use this control to link old objects (for example compiled for `model(kb)`).

## Example

The following example CODE segments to HIGHCODE:

```
rl196 mod1.obj, mod2.obj ch
```

# dataoverlay

## Function

Overlay data segments.

## Syntax



Select the EDE | Linker Options... menu item. Add the control to the Additional options field in the Misc tab.



```
dataoverlay(overlay_unit [,...]) | nodataoverlay
```

where:

```
overlay_unit ⇒ overlay_element [ {-> | -} overlay_element ]
```

```
overlay_element ⇒ module_name | ( overlay_unit [,...])
```

*module\_name* is a valid name of a module.

```
overlay_factor [ -> | -] overlay_factor]
```

*overlay\_factor* is defined as:

```
module_name | ( overlay_unit [,...])
```

## Abbreviation

do | nodo

## Class

Locating control

## Default

nodataoverlay

## Description

Use this control to specify data overlaying for the specified modules. You can also specify the constraints to be applied in performing data overlaying.

Based on the `dataoverlay` control, the linker builds a calling graph internally, as follows:

- Each module specified is designated as a node.
- Each  $A \rightarrow B$  relationship is designated by an arc from A to B.

This calling graph can be cyclic. The calling graph guides RL196 during memory allocation of overlayable data segments.

In the calls relationship, the right arrow signifies a hyphen followed by a greater than sign ( $\rightarrow$ ) or a hyphen followed by a right square bracket ( $\rightarrow$ ). For some operating systems, the greater than sign ( $>$ ) has a special meaning so it cannot be used to designate the calls relationship. The hyphen followed by a right square bracket ( $\rightarrow$ ) can be used on all operating systems.

A right arrow ( $\rightarrow$ ) designates the calls relationship. *A Calls B* means that module A cannot overlay module B. The calls relationship is transitive, that is,  $A \rightarrow B$  and  $B \rightarrow C$  implies  $A \rightarrow C$ .

In general, the default of overlaying is not to overlay segments. The default is expressed in two ways:

- The control default is `nodataoverlay`.
- Any module not mentioned in the `dataoverlay` control is not overlaid, that is, its relocatable overlayable data segment, if it has one, is regarded as a normal data segment during the memory allocation process.

Regarding modules whose names appear in the `dataoverlay` control, you must specify any relevant calls relationships between two modules. If both module X and module Y are mentioned within the `dataoverlay` control parameter, but neither  $X \rightarrow Y$  or  $Y \rightarrow X$  is specified and neither of the relationships can be deduced (by transitivity), then the linker concludes that X and Y can overlay. By the same token, if you specify a module without using the calls relationship, the linker overlays the module on another specified module.

Try to avoid hidden calls. Sometimes RL196 does not detect dangerous overlaying, such as when the address of a procedure in module A is passed through module C to the calling module, B.



As mentioned earlier, a module name within the control parameter stands for its relocatable overlay segment. Absolute overlay segments are treated like any other absolute segments, so they do not participate in the overlaying. Therefore, overlaying modules during an incremental link-locate is usually less efficient than overlaying them during a single-step link-locate.

If you specify a module in the `dataoverlay` control that does not exist, the following error is issued:

THE SPECIFIED MODULE DOES NOT EXIST

The overlayable data segments of modules A and B are allowed to overlay only if during program execution, procedures in module A do not call any procedure of module B, directly or indirectly.

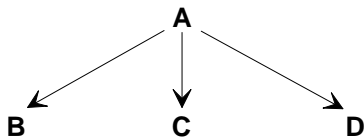
If the linker finds a call or a jump between two overlaid modules, that is, their relocatable overlayable data segments are overlaid, the linker issues the following warning:

A DIRECT CALL BETWEEN TWO OVERLAID MODULES

In some situations, you can disregard this warning. For example, you have two modules, Y and Z. Module Y consists of functions AB and CD. Function AB has overlayable data variables and CD does not. Module Z consists of functions EF and GH. EF has overlayable data variables and GH does not. As long as AB calls GH or CD calls EF, no two overlay segments are active at the same time. You can avoid complex overlaying by keeping one function per module or place all functions with overlayable data in the same module.

## Examples

1. In this example, Module A calls Modules B, C, and D. The calling graph of the application is as follows:

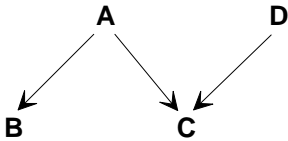


You can ask for the maximum overlaying possible by using either of the control sequences:

```
dataoverlay(A -> (B, C, D))
do(B, C, D)
```

In the first alternative, the control parameter specifies that modules A, B, C and D are to be overlaid under the constraint that A cannot overlay B, C, or D. This call relationship means that A cannot overlay any other specified module. Therefore, A can be eliminated from the control parameter, shown in the second alternative.

2. In this example, Module A calls Modules B and C. Module D also calls Module C. The calling graph of the application is as follows:



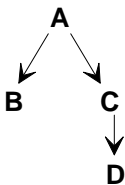
You can specify the maximum overlaying:

```
do(A -> (B,C), D -> C)
```

You can also specify the same structure:

```
do((A,D) -> C, A -> B)
```

3. In this example, Module A calls Modules B and C, and Module C calls Module D. The calling graph of the application is as follows:



You can specify the maximum overlaying:

```
do(A -> (B,C -> D))
```

However, since the call relationship is transitive, the following control line is also sufficient:

```
do(A -> (B,C), C -> D)
```

# heap

## Function

Locates the HEAP space in RAM.

## Syntax



Select the EDE | Linker Options... menu item. Enable or disable the Use largest section of free RAM as HEAP space check box in the Locating tab.



heap		noheap
-he		-he-

## Abbreviation

he		nohe
----	--	------

## Class

Locating control

## Default

npheap

## Description

Use this control to tell RL196 to locate the HEAP space. RL196 supplies the public symbols `_HEAP_START_` and `_HEAP_END_` which define this HEAP space.

## Example

The following example will locate the largest section of free RAM as HEAP space:

```
r1196 mod1.obj, mod2.obj heap
```

The following example will locate the largest section of free RAM in the range of 3000H-3FFFH as HEAP space:

```
r1196 mod1.obj, mod2.obj heap rom(2000H-2FFFFH)
      ram(3000H-3FFFH(mod1, heap), 4000H-7FFFFH)
```



ram

# ignoreabs

## Function

Ignores warnings about absolute segments.

## Syntax



Select the EDE | Linker Options... menu item. Enable or disable the Ignore warnings about absolute segments in range 0-1FFFFH) check box in the Linking tab.



<code>ignoreabs</code>		<code>noignoreabs</code>
<code>-ia</code>		<code>-ia-</code>

## Abbreviation

`ia` | `noia`

## Class

Locating control

## Default

`ignoreabs`

## Description

Use this control to prevent warnings about absolute segments outside the area specified with the `ram` or `registers` control. Only segments in range 0-1FFFFH are ignored. This control is mainly used for having absolute segments for SFR areas.

## Example

The following example ignores warnings about absolute segments:

```
rl196 mod1.obj, mod2.obj ia
```

# inittable

## Function

Generates the initialization table.

## Syntax



Select the EDE | Linker Options... menu item. Enable or disable the Generate ROM table for initialized RAM variables check box in the Locating tab.



`inittable` | `noinittable`

## Abbreviation

`it` | `noit`

## Class

Locating control

## Default

`inittable`

## Description

Use this control to suppress the generation of the initialization table and the public symbol `_INIT_TABLE_START_`. Note that the initialization table is not generated if it is empty, but that the `_INIT_TABLE_START_` is always generated.

## Example

The following example does not generate an initialization table or the public symbol `_INIT_TABLE_START_`:

```
r1196 mod1.obj, mod2.obj noinittable
```

# inst

## Function

Allows ROM and RAM addresses to be overlapped.

## Syntax



Select the EDE | Linker Options... menu item. Enable or disable the Overlap ROM and RAM addresses (using the INST pin) check box in the Locating tab.



```
inst | noinst
-in  | -in-
```

## Abbreviation

in | noin

## Class

Locating control

## Default

noinst

## Description

Use this control to specify ROM-type memory that is independent of RAM-type memory so that the addresses of the two types can overlap. The `inst` control supports applications that use the 80C196 INST signal in memory addressing logic. This control provides a section of ROM-type memory accessible only via instruction fetches.

When you specify the `inst` control, the linker allocates the RAM and ROM sections independently, allowing address overlaps between, but not within, the two sections. Allocation within each section occurs as defined by the `ram` and `rom` controls, with the defaults, steps, and rules within each section. The segment map shows two sections, as follows:

- The RAM-type section with register, stack, and data segments.
- The new INST segment section with all of the code segments.

The linker places all code segments in the INST section, including constants. Take special care when designing the application memory map and addressing logic. You must be able to locate constants and other object modules in non-overlapping memory if necessary.



See Section 2.5.5 for hardware and software development guidelines and a sample RL196 invocation.

# ixref

## Function

Include intermodule cross-reference listing in the print file.

## Syntax



Select the EDE | Linker Options... menu item. Enable or disable the Include a cross-reference listing in the map file check box in the Listing tab.



```
ixref | noixref
-ix   | -ix-
```

## Abbreviation

ix | noix

## Class

Listing control

## Default

noixref

## Description

Use this control to include an intermodule cross-reference listing in the print file. The intermodule cross-reference listing contains the symbol names, the segment type associated with each symbol, the symbol type of each symbol, and the names of all modules in which each symbol is declared as public or external. Section 2.6.3 describes the intermodule cross-reference listing in more detail and includes an example.

## Example

The following example includes an intermodule cross-reference listing in the print file:

```
r1196 mod1.obj, mod2.obj ixref
```



print



# limit\_bitno

## Function

Do not allow bit numbers greater than 7.

## Syntax



Select the EDE | Linker Options... menu item. Enable or disable the Allow bit numbers larger than 7 check box in the Linking tab.



<code>limit_bitno</code>		<code>nolimit_bitno</code>
<code>-lb</code>		<code>-lb-</code>

## Abbreviation

<code>lb</code>		<code>nolb</code>
-----------------	--	-------------------

## Class

Linking control

## Default

`nolimit_bitno`

## Description

When you use the JBS or JBC instruction with an external bit number, the linker will have to fill in the bit number. It is allowed to specify a bit number which is larger than 7. If this is the case, then the bit register will be increased by one and the bit number will be decreased by 8 until the bit number is smaller than 8. If the `limit_bitno` control is used, all external bit number with a value greater than 7 will generate an error.

## Example

The following example does not allow bit numbers larger than 7 in its object files:

```
r1196 mod1.obj, mod2.obj limit_bitno
```

# list

## Function

Specifies elements to be included in the print file.

## Syntax



Select the EDE | Linker Options... menu item. Enable or disable one or more of the Include ... in the map file check boxes in the Listing tab.



```
list[(element[,...])] | nolist
```

where:

*element* is one or a combination of the following: `publics`, `symbols`, `lines` and `segments`.

## Abbreviation

`li` | `noli`

## Class

Listing control

## Default

```
list(publics, symbols, lines, segments)
```

## Description

Use this control to specify elements you want included in the print file. Use the `print` control to assign the name of the print file. By using the `list` control, you can specify the listing of any combination of the following elements:

- public symbols (`publics` or `pl`)
- symbols (`symbols` or `sb`)
- source statement line numbers (`lines` or `ln`)
- segment map (`segment` or `sm`)

The `nolist` control specifies that the print file includes none of the categories. If you do not explicitly enter either `list` or `nolist`, the default setting is `list(publics, symbols, lines, segments)`. When some elements other than the default are required, you must specify all of the required elements. For example, to list everything except line numbers, specify `list(segments, public, symbols)`. Specifying just `list` without any parameter is equivalent to the default.

To select the symbol table content, specify `publics`, `symbols`, and `lines`. To specify that a segment map is required, include `segments` in the control.

Specify `symbols` and/or `lines` to also include information on the input modules and input blocks.

### Example

This example includes `segments` and `publics` only:

```
r1196 def.obj list(segments, publics)
```



```
print
```

# model

## Function

Specifies the processor/memory model.

## Syntax



Choose a cpu from the EDE | CPU Model... menu item. Optionally select one or more of the radio buttons Near Code/Far Code, Near Const/Far Const, Near Data/Far Data.



```
model(processor)  
-md processor
```

where:

*processor* Selects the memory model the RL196 uses in locating code for a specific member of the 80C196 processor family.

## Abbreviation

md

## Class

Locating control

## Default

```
model(kb)
```

## Description

This control allows you to specify which processor/memory model you are using, thus determines the physical memory layout RL196 must follow to locate your data, code, and constants.

Specify the *processor* as one of the following:

61 to select the 8096-61.

90 to select the 8096-90.

196	to select the 80C196KB. This argument to <code>model</code> is available for backward compatibility and is equivalent to specifying <code>kb</code> . For future compatibility, use the <code>model(kb)</code> control specification instead of <code>model(196)</code> .
bh	to select the 8096BH.
ca	to select the 80C196CA. Specifying <code>ca</code> is equivalent to specifying <code>kr</code> .
cb	to select the 80C196CB. This argument can have an extra suffix as described in the note below.
ea	to select the 80C196EA. This argument can have an extra suffix as described in the note below.
ec	to select the 80C196EC. This argument can have an extra suffix as described in the note below.
jq	to select the 80C196JQ. Specifying <code>jq</code> is equivalent to specifying <code>kr</code> .
jr	to select the 80C196JR. Specifying <code>jr</code> is equivalent to specifying <code>kr</code> .
js	to select the 80C196JS. Specifying <code>js</code> is equivalent to specifying <code>kr</code> .
jt	to select the 80C196JT. Specifying <code>jt</code> is equivalent to specifying <code>kr</code> .
jv	to select the 80C196JV. Specifying <code>jv</code> is equivalent to specifying <code>kr</code> .
kb	to select the 80C196KB. Specifying <code>kb</code> is equivalent to specifying <code>196</code> .
kc	to select the 80C196KC.
kd	to select the 80C196KD.
kl	to select the 80C196KL. Specifying <code>kl</code> is equivalent to specifying <code>kr</code> .
kq	to select the 80C196KQ. Specifying <code>kq</code> is equivalent to specifying <code>kr</code> .

kr	to select the 80C196KR.
ks	to select the 80C196KS. Specifying ks is equivalent to specifying kr.
kt	to select the 80C196KT. Specifying kt is equivalent to specifying kr.
lb	to select the 80C196LB.
mc	to select the 80C196MC.
md	to select the 80C196MD.
mh	to select the 80C196MH.
np	to select the 80C196NP. This argument can have an extra suffix as described in the note below.
nt	to select the 80C196NT. This argument can have an extra suffix as described in the note below.
nu	to select the 80C196NU. This argument can have an extra suffix as described in the note below.



The cb, ea, np, nt and nu arguments of the model control can have an additional suffix. Without a suffix, specifying *xx* is the same as specifying *xx-c*, where *xx* is one of cb, ea, ec, np, nt or nu. The following six suffixes are possible:

<i>xx-c</i>	to select the compatible mode and to use near code addressing and near data/near const addressing.
<i>xx-cn</i> f	to select the compatible mode and to use near code addressing and near data/far const addressing.
<i>xx-c</i> f	to select the compatible mode and to use near code addressing and far data/far const addressing.
<i>xx-e</i>	to select the extended mode and to use far code addressing and near data/near const addressing.
<i>xx-en</i> f	to select the extended mode and to use far code addressing and near data/far const addressing.
<i>xx-e</i> f	to select the extended mode and to use far code addressing and far data/far const addressing.

If you specify one of the compatible controls, RL196 assumes the following:

- address space is from 0 to 0FFFFFFH
- default `rom` control is `rom(2000H-0FFFFH, 0FF2000H-0FFFFFFH)`
- Intel reserved area is from 0FF2000H to 0FF207FH
- only type of code segment allowed in input modules is high code
- input modules can contain far data and far constant segments.

If you specify one of the extended controls, RL196 assumes the following:

- address space is from 0 to 0FFFFFFH
- default `rom` control is `rom(2000H-0FFFFH, 0FF2000H-0FFFFFFH)`
- Intel reserved area is from 0FF20000H to 0FF207FH
- only type of code segment allowed in input modules is far code
- input modules can contain far data and far constant segments.

If you specify a control for one of the 16-bit models, RL196 assumes the following:

- you are using the 8096/80196 family of microcontrollers. Therefore, the address space is from 0 to 0FFFFH
- default `rom` control is `rom(2000H-0FFFFH)`
- Intel reserved area is from 2000H to 207FH
- only type of code segment allowed in input modules is near code
- far data and far constant segments are not allowed.



The `model` control cannot be specified with the `inst` control.

# name

## Function

Assigns a module name to the output file.

## Syntax



Select the EDE | Linker Options... menu item. Add the control to the Additional options field in the Misc tab.



```
name(module_name)
```

where:

*module\_name* is a string of characters.

## Abbreviation

na

## Class

Linking control

## Default

```
name(first_input_module_name)
```

## Description

Use this control to assign a module name, specified by *module\_name*, to the output module produced by RL196. If you do not use the name control, the linker uses the name of the first input module as the default output module name without any extension.

The *module\_name* can be 40 characters long. You can use the following characters in any order:

```
? (question mark)
@ (commercial at)
: (colon)
. (period)
_ (underscore)
A, B, C, ..., Z or
0, 1, 2, ...,9
```





The name control does not affect the output filename. Only the module name in the output module's header record is changed.

### Example

In this example, `monitor` is assigned as the output module name produced by `RL196`. The output absolute filename is `program1`. The print file produced is `program1.m96`.

```
r1196 program1.obj name(monitor)
```

# nearcode/farcode

## Function

Specify code space configuration for 24-bit models.

## Syntax



Select the EDE | CPU Model... menu item. Select the Near Code or Far Code radio button.



`nearcode | farcode`

## Abbreviation

`nc | fc`

## Class

Locating control

## Default

`nearcode`

## Description

`nearcode` and `farcode` specify code space configuration for a member of the 24-bit 80C196 family. `nearcode` specifies that the microcontroller is configured in compatible mode. `farcode` specifies that the microcontroller is configured in extended mode. These controls must be preceded by a 24-bit `model` control.

## Example

This example specifies the NT model with far code addressing and near data/near const addressing, both invocations are the same:

```
r1196 mod1.obj, mod2.obj md(nt-e)
```

```
r1196 mod1.obj, mod2.obj md(nt) fc
```



`model control`

# nearconst/farconst

## Function

Specify constant space configuration for 24-bit models.

## Syntax



Select the EDE | CPU Model... menu item. Select the Near Const or Far Const radio button.



`nearconst` | `farconst`

## Abbreviation

`nk` | `fk`

## Class

Locating control

## Default

`nearconst`

## Description

`nearconst` and `farconst` specify the constant space configuration for the 24-bit 80C196 family of microcontrollers. `nearconst` specifies that all data, unless otherwise indicated, reside in the first 64 kilobytes of the address space. `farconst`, on the other hand, means that all data, unless otherwise specified, are located in the 16-megabytes address space. These controls must be preceded by a 24-bit `model` control.

## Example

This example specifies the NT model with near code addressing and near data/far const addressing, both invocations are the same:

```
r1196 mod1.obj, mod2.obj md(nt-cnff)
```

```
r1196 mod1.obj, mod2.obj md(nt) fk
```



`model` control

# neardata/fardata

## Function

Specify data space configuration for 24-bit models.

## Syntax



Select the EDE | CPU Model... menu item. Select the Near Data or Far Data radio button.



neardata | fardata

## Abbreviation

nd | fd

## Class

Locating control

## Default

neardata

## Description

neardata and fardata specify the data space configuration for the 24-bit 80C196 family of microcontrollers. neardata specifies that all data, unless otherwise indicated, reside in the first 64 kilobytes of the address space. fardata, on the other hand, means that all data, unless otherwise specified, are located in the 16-megabytes address space. These controls must be preceded by a 24-bit model control.

## Example

This example specifies the NT model with near code addressing and far data/far const addressing, both invocations are the same:

```
r1196 mod1.obj, mod2.obj md(nt-cf)
```

```
r1196 mod1.obj, mod2.obj md(nt) fd
```



model control

# np\_rsvup6

## Function

Reserve upper six bytes of every page (model(np) and model(nu) only).

## Syntax



Select the EDE | Linker Options... menu item. Enable or disable the Reserve upper 6 bytes of every page (NP and NU onbly) check box in the Locating tab.



np\_rsvup6 | nonp\_rsvup6

## Class

Locating control

## Default

nonp\_rsvup6

## Description

Use this control to prevent the linker from putting code in the last 6 bytes of any page. This control is only valid for an NP or NU model.

## Example

This example reserves the upper six bytes of any page:

```
r1196 mod1.obj, mod2.obj md(np) np_rsvup6
```

# omf

## Function

Specifies OMF96 version.

## Syntax



Select the EDE | Linker Options... menu item. Select an OMF96 Version radio button in the Format tab.



`omf ( n )`  
`-omf : n`

where:

`n` is the number representing the OMF96 version:

- 0 – OMF96 V2.0
- 1 – OMF96 V3.0
- 2 – OMF96 V3.2 (default)

## Abbreviation

`omf`

## Class

Locating control

## Default

`omf ( 2 )`

## Description

Use this control is used to specify the OMF96 version to generate. In a previous version of the linker you could use the control `oldobject` to specify OMF96 V2.0. Also some users were advised to use the internal control `oo1`. The two controls are now combined in one control, `omf`.

## Example

This invocation line tells the linker to use the old OMF96 version V2.0.

```
r1196 mod1.obj, mod2.obj omf(0)
```



Section 2.4

# pageprint

## Function

Prints all code addresses in compatibility mode as 24 bits.

## Syntax

`pageprint` | `nopageprint`

## Abbreviation

`pp` | `nopp`

## Class

Listing control

## Default

`pageprint`

## Description

By default all code addresses in the symbol table are listed as 24 bit addresses. When you specify a model in compatibility mode (`xx-c`, `xx-cnf` or `xx-cf`), the upper 8 bits of the code addresses are always 0xFF (all code in compatibility mode ends up in page 0xFF).

If you specify the `nopageprint` control, the page (i.e. the upper 8 bits of the code address) is not printed. This control only influences the printing of the code addresses in the symbol table and only if you have selected a compatibility mode. Addresses in the segment map are always printed as 24 bit, when using an 24 bit model.

# pagewidth

## Function

Specifies the maximum number of characters per line.

## Syntax



Select the EDE | Linker Options... menu item. Enter the number of *characters* in the Page width (characters per line) field in the Listing tab.



```
pagewidth(number)  
-pw number
```

where:

*number* is a valid number from 72 to 255. This number can also be specified in hexadecimal format.

## Abbreviation

pw

## Class

Listing control

## Default

```
pagewidth(120)
```

## Description

Use this control to specify the maximum number of characters to be printed on a single line. If the number specified is less than 72 or greater than 255, the linker generates an error. Carriage returns and linefeeds are not counted.

## Example

This example specifies that the maximum number of characters to be printed on a single line is 90.

```
r1196 remote.obj ix pw(90)
```



# print

## Function

Directs the listing produced to the specified file.

## Syntax



Select the EDE | Linker Options... menu item. Enable or disable the Generate a linker map file (.m96) check box in the Listing tab.



```
print(output_file) | noprint
-M output_file      | -M-
```

where:

*output\_file* is a string of characters.

## Abbreviation

pr | nopr

## Class

Listing control

## Default

```
print(first_input_file.m96)
```

## Description

Use this control to direct the listing produced by RL196 to the specified file. The specified file cannot have the same name as any input files or the output object file. If you enter the `print` control without naming an output file, the default output file is the first input filename with a `.m96` extension.

If you specify the `noprint` control, no listing is produced. The `noprint` control overrides the `list` and the `ixref` controls if they have been specified.

### Example

This example produces a print file named 96\_appl.m96.

```
r1196 mod1.obj, mod2.obj print(96_appl.m96)
```



```
list  
ixref
```

# purge

## Function

Specifies which elements are removed from the output file.

## Syntax



Select the EDE | Linker Options... menu item. Enable or disable the Include debug information check box in the Linking tab.



```
purge[(element [,...])] | nopurge
```

where:

*element* can be one or a combination of the following: public, symbol, lines or segments.

## Abbreviation

pu | nopu

## Class

Linking control

## Default

nopurge

## Description

Use this control to specify elements to be removed from the output object file. You can select a combination of the following elements for removal by entering the keyword (or its abbreviation):

publics (pl) public symbol definition records

symbols (sb) local symbol debug records

lines (ln) source statement line numbers

segments (sm) segment definition records

Entering the purge control and no element list causes RL196 to purge all of the elements equivalent to `purge(publics, symbols, lines, segments)`.

Information regarding modules and blocks is purged only if you specify both `symbols` and `lines` with this control.

Enter `nopurge` to remove none of the elements from the object module.

If you want to debug the output of the link-locate process, do not purge `symbols` and `lines`.



When performing incremental links, do not purge segments and publics before the last link-locate. Purging these elements destroys the segment and public information needed by the subsequent links.

### Example

The following example produces an absolute object file without any public symbol information:

```
r1196 mod1.obj, mod2.obj purge(publics)
```



```
absstack
```

# quietwarns

## Function

Prevents warnings from displaying.

## Syntax



Select the **EDE | Linker Options...** menu item. Enable or disable the **Display warning in the output window** check box in the **Linking** tab.



<code>quietwarns</code>		<code>noquietwarns</code>
<code>-QW</code>		<code>-QW-</code>

## Abbreviation

`qw` | `noqw`

## Class

Linking control

## Default

`noquietwarns`

## Description

Use this control to prevent the linker from displaying warnings on your screen. Warnings are still put in the map file.

## Example

This example prevents warnings from displaying:

```
r1196 mod1.obj, mod2.obj qw
```

# ram

## Function

Specifies the designated RAM section.

## Syntax



Select the EDE | Linker Options... menu item. Enter a RAM address range in the RAM field in the Memory tab.



```
ram(ram_section [(module_list [ (ram_seg_list) ] )] [,...])
```

where:

*ram\_section* is an address range specifying the starting address and the ending address of available RAM separated by a hyphen.

*module\_list* is a list of valid module names.

*ram\_seg\_list* is a list of data segments found in the module. Possible segments are data, fardata and stack.

## Abbreviation

*ra* for ram  
*dt* for data  
*fd* for fardata  
*st* for stack

## Class

Locating control

## Default

```
ram (1AH-1FFFH(stack))
```

## Description

Use this control to designate the RAM address range. You must specify the RAM sections in ascending order. Follow these rules for each *start\_address* - *end\_address* pair:

- The *start\_address* must be greater than the previous *end\_address*. The minimum *start\_address* is 1AH.

- The *end\_address* must be greater than or equal to its *start\_address*. The maximum *end\_address* is 0FFFFFH or 0FFFFFFFH for far data segments.

When you include a *module\_list* with this control, the linker allocates the relocatable data segments of the specified modules within the address range specified. If a module name is followed by an explicit *ram\_seg\_list*, then only the data segment specified in that list for this module is allocated in the specified range. The memory allocation for these segments is performed from left to right as the modules are encountered in the list. Because of the fragmentation that results from the scattering of absolute segments in the memory section, the segments in physical memory are not necessarily ordered as they are encountered. The keyword *stack*, abbreviated as *st*, can appear as a pseudo-module name in a module list of this control which implies that RL196 must allocate the relocatable stack segment within the associated RAM section.

The linker allocates relocatable segments after all of the absolute segments are allocated. This process has two steps:

1. The data segments for the modules specified in the *ram* control are allocated so that each segment is within its specified RAM section.
2. The rest of the data segments are allocated in the remaining free RAM.

The second step allocates data segments not specified for allocation by the first step.

If you specify *stack* with a RAM section, either explicitly or by default, and a relocatable stack segment is present, the stack segment is allocated in the first step mentioned above. If you do not specify *stack* with the *ram* control, the linker allocates the relocatable stack segment (if any) in the second step.

ROM and RAM sections must not overlap unless you specify the *inst* control; also, they need not exhaust the entire memory range. Unspecified memory locations are treated as gaps.

The default register section of memory consists of the internal register area (i.e., memory addresses 1AH to 0FFH). If you perform incremental links, do not use the register section of memory for RAM sections to allow register segments of subsequent links to be allocated. Do not specify the register space address range in the `ram` control. The linker uses any unused memory in the register section which overlaps a RAM section, specified by `ram`, as RAM after all register segments have been allocated. The ordinary default condition produces an overlapping between the RAM and the register sections from address 1AH to 0FFH.



The address range 2000H-207FH or 0FF2000H-0FF207F for the 24-bit components is reserved for special use by Intel. RL196 does not locate any relocatable segments in this range. The only way to place data in this range is by using absolute segments.

### Example

The RAM sections in this example are 100H to 220H and 1018H to 1FFFFH. The relocatable data segment of module `allocmod` must be allocated in the range 100H to 200H. The space from 201H to 220H contains the relocatable data segment of module `main`.

```
rl196 main.obj, allocate.obj, srt.obj to main.lnk noas  
      ram(100H-200H(allocmod), 201H-220H(main),  
      1018H-1FFFFH)
```

After the register segments and overlay segments of the input modules are allocated in the register space, the unused register memory is not used for the data segment of `main.lnk` because register space and RAM space do not overlap.



# registers

## Function

Specifies the range of registers available to the application.

## Syntax



Select the **EDE | Linker Options...** menu item. Enter a register address range in the **Register space** field in the **Memory** tab.



```
registers(address_range [,...])
```

where:

*address\_range* is an address range specifying the starting and ending address of the available register space separated by a hyphen.

## Abbreviation

rg

## Class

Locating control

## Default

```
registers(1AH-0FFH) or  
registers(26-255)
```

## Description

Use this control to specify the range of registers available to the application. You must specify the register sections in ascending order. Follow these rules for each *start\_address* - *end\_address* pair:

- The *start\_address* must be greater than the previous *end\_address*.
- The *start\_address* must begin at a 128 byte boundary (except for the first 1AH).
- The *end\_address* must be greater than or equal to its *start\_address*.
- The *end\_address* must be at multiples of 128 minus 1.

If you do not specify this control, the linker uses the default, `registers(1AH-0FFH)`, which produces an output object file that matches the one produced without windowing capabilities. If you specify this control with a non-default address range, the linker locates the register overlay segments into the additional register space using the vertical windowing mechanism of the 80C196KC and the 80C196KR microcontrollers. See Section 2.5.6 for more information on how RL196 allocates register variables.

### Example

This example shows that the register space ranges from 1AH to 1FFH and that you are requesting a window size of 64 bytes.

```
rl196 mod1.obj, mod2.obj, mod3.obj, mod4.obj  
      rg(1AH-7FH, 80H-17FH, 180H-1FFH) windowsize(64)
```



windowsize

# regfirst

## Function

Allocate register segments first.

## Syntax



Select the EDE | Linker Options... menu item. Enable or disable the Locate register segments before overlayable segments check box in the Locating tab.



<code>regfirst</code>		<code>noregfirst</code>
<code>-rf</code>		<code>-rf-</code>

## Abbreviation

`rf` | `norf`

## Class

Locating control

## Default

`noregfirst`

## Description

If the `regfirst` control is specified relocatable register segments are allocated first, followed by relocatable overlay segments of modules specified by the `regoverlay` control and relocatable overlay segments not yet allocated.

If the `noregfirst` control is specified (default) relocatable overlay segments of modules specified by the `regoverlay` control are allocated first, followed by relocatable overlay segments not yet allocated and relocatable register segments.

When you use the `window`size control this control has no effect.



See Section 2.5 for more information on memory allocation.

**Example**

The following example allocates register segments first:

```
rl196 mod1.obj, mod2.obj rf
```

# regoverlay

## Function

Overlay register.

## Syntax



Select the EDE | Linker Options... menu item. Enter an *overlay\_unit* in the User specified regoverlay control field in the Locating tab.



```
regoverlay(overlay_unit [...]) | noregoverlay
```

where:

```
overlay_unit ⇒ overlay_element [ {-> | -} overlay_element ]
```

```
overlay_element ⇒ module_name | ( overlay_unit [...])
```

*module\_name* is a valid name of a module.

```
overlay_factor [ -> | -] overlay_factor
```

*overlay\_factor* is defined as:

```
module_name | ( overlay_unit [...])
```

## Abbreviation

ov | noov

## Class

Locating control

## Default

noregoverlay

## Description

Use this control to specify register overlaying for the specified modules. You can also specify the constraints to be applied in performing register overlaying.

Because some registers serve different purposes at different times during program execution, register overlaying enables more variables be allocated in the register section than in on-chip or off-chip RAM. The result is an increased in execution speed and a decreased in memory demand.

In the 80C196 environment, you can only overlay registers in the relocatable overlay segments.; therefore, the term overlaying is used rather than register overlaying. Therefore, the phrase *module A overlays module B* means that the overlay segment of module A overlays the overlay segment of module B.

If you specify overlaying between modules, the linker generates a calling graph which determines what modules can be overlayed. The linker might not overlay the registers completely or perhaps none at all, as determined by the calling graph. However, the linker does not overlay modules specified in a call relationship or if you use `noregoverlay`.

Based on the `regoverlay` control, the linker builds a calling graph internally, as follows:

- Each module specified is designated as a node.
- Each  $A \rightarrow B$  relationship is designated by an arc from A to B.

This calling graph can be cyclic. The calling graph guides RL196 during memory allocation of overlay segments.

In the calls relationship, the right arrow signifies a hyphen followed by a greater than sign ( $\rightarrow$ ) or a hyphen followed by a right square bracket ( $\rightarrow$ ). For some operating systems, the greater than sign ( $\rightarrow$ ) has a special meaning so it cannot be used to designate the calls relationship. The hyphen followed by a right square bracket ( $\rightarrow$ ) can be used on all operating systems.

A right arrow ( $\rightarrow$ ) designates the calls relationship. *A Calls B* means that module A cannot overlay module B. The calls relationship is transitive, that is,  $A \rightarrow B$  and  $B \rightarrow C$  implies  $A \rightarrow C$ .

In general, the default of overlaying is not to overlay segments. The default is expressed in two ways:

- The control default is `noregoverlay`.
- Any module not mentioned in the `regoverlay` control is not overlaid, that is, its relocatable overlay segment, if it has one, is regarded as a register segment during the memory allocation process.

Regarding modules whose names appear in the `regoverlay` control, you must specify any relevant calls relationships between two modules. If both module X and module Y are mentioned within the `regoverlay` control parameter, but neither `X -> Y` or `Y -> X` is specified and neither of the relationships can be deduced (by transitivity), then the linker concludes that X and Y can overlay. By the same token, if you specify a module without using the calls relationship, the linker overlays the module on another specified module.

Try to avoid hidden calls. Sometimes RL196 does not detect dangerous overlaying, such as when the address of a procedure in module A is passed through module C to the calling module, B.

As mentioned earlier, a module name within the control parameter stands for its relocatable overlay segment. Absolute overlay segments are treated like any other absolute segments, so they do not participate in the overlaying. Therefore, overlaying modules during an incremental link-locate is usually less efficient than overlaying them during a single-step link-locate.

If you specify a module in the `regoverlay` control that does not exist, the following error is issued:

```
THE SPECIFIED MODULE DOES NOT EXIST
```

The overlay segments of modules A and B are allowed to overlay only if during program execution, procedures in module A do not call any procedure of module B, directly or indirectly.

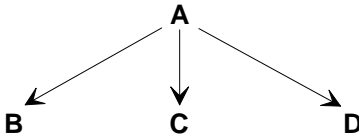
If the linker finds a call or a jump between two overlaid modules, that is, their relocatable overlay segments are overlaid, the linker issues the following warning:

```
A DIRECT CALL BETWEEN TWO OVERLAID MODULES
```

In some situations, you can disregard this warning. For example, you have two modules, Y and Z. Module Y consists of functions AB and CD. Function AB has overlayable register variables and CD does not. Module Z consists of functions EF and GH. EF has overlayable register variables and GH does not. As long as AB calls GH or CD calls EF, no two overlay segments are active at the same time. You can avoid complex overlaying by keeping one function per module or place all functions with overlayable registers in the same module.

## Examples

1. In this example, Module A calls Modules B, C, and D. The calling graph of the application is as follows:



You can ask for the maximum overlaying possible by using either of the control sequences:

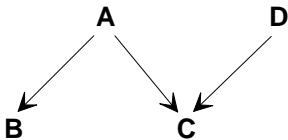
```

regoverlay(A -> (B, C, D))
ov(B, C, D)

```

In the first alternative, the control parameter specifies that modules A, B, C and D are to be overlaid under the constraint that A cannot overlay B, C, or D. This call relationship means that A cannot overlay any other specified module. Therefore, A can be eliminated from the control parameter, shown in the second alternative.

2. In this example, Module A calls Modules B and C. Module D also calls Module C. The calling graph of the application is as follows:



You can specify the maximum overlaying:

```

ov(A -> (B,C), D -> C)

```

You can also specify the same structure:

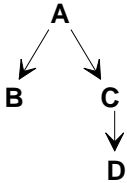
```

ov( (A,D) -> C, A -> B)

```



3. In this example, Module A calls Modules B and C, and Module C calls Module D. The calling graph of the application is as follows:



You can specify the maximum overlaying:

```
ov(A -> (B,C -> D))
```

However, since the call relationship is transitive, the following control line is also sufficient:

```
ov(A -> (B,C), C -> D)
```

# rom

## Function

Specifies designated ROM sections.

## Syntax



Select the EDE | Linker Options... menu item. Enter a ROM address range in the ROM field in the Memory tab.



```
rom(rom_section [(module_list [(rom_seg_list)] )] [,...])
```

where:

*rom\_section* is an address range specifying the starting address and the ending address of available ROM separated by a hyphen.

*module\_list* is an optional list of valid module names.

*rom\_seg\_list* is an optional list of code and constant segments found in the module. Possible segments are code, farcode, const, farconst.

## Abbreviation

ro for rom  
co for code  
fc for farcode  
ko for const  
fk for farconst

## Class

Locating control

## Default

rom(2000H-0FFFFH) for 16-bit models  
rom(0FF2000H-0FFFFFFFH) for 24-bit models

## Description

Use this control to designate the ROM address range. You must specify the ROM sections in ascending order. Follow these rules for each *start\_address* - *end\_address* pair:

- The *start\_address* must be greater than the previous *end\_address*. The minimum *start\_address* is 0H.
- The *end\_address* must be greater than or equal to its *start\_address*. The maximum *end\_address* is 0FFFFH or 0FFFFFFH for far code or high code and far constant segments.

When you specify a *module\_list* with this control, the linker allocates the relocatable code and constant segments of the specified modules within the address range specified. However, if a module name in the *module\_list* is followed by an explicit *rom\_seg\_list*, then only the code and constant segments specified in the *rom\_seg\_list* for this module is allocated in the specified memory range. The linker performs the memory allocation for these segments from left to right as the modules appear in the list. Because of the fragmentation that results from the scattering of absolute segments in the memory section, the actual order of the segments in physical memory does not necessarily match the order in which they were appear in the command line. You cannot specify *stack* in the *rom* control.

The linker allocates the relocatable code and constant segments after all of the absolute segments are allocated. This process has two steps:

1. Code and constant segments for the modules specified in the *rom* control are allocated so that each segment is within its specified ROM section.
2. The rest of the code and constant segments are allocated in the remaining free ROM.

The second step allocates any code and constant segments not specified for allocation by the first step.

ROM and RAM sections must not overlap unless the *inst* control is in effect. Also, they need not exhaust the entire memory range. Unspecified memory locations are treated as gaps.



Once you explicitly specify a particular ROM section with this control, you must specify all of the remaining ROM sections. The address range 2000H–207FH or 0FF2000H–0FF207F for the 80C196NT components is reserved for special use by Intel. RL196 does not locate any relocatable segments in this range. The only way to place code or constants in this range is by using absolute segments. Modules belonging to `publiconly` files cannot be specified in the `module_list`.

You can use the `romcode` and `romdata` controls instead of the `rom` control. These controls can be very useful when using the 80C196KR or a 24-bit processor.

## Examples

1. The ROM sections in this example are 2180H to 2300H and 0E000H to 0FFFFH. The range 2180H to 2300H is specified in two parts to indicate that the relocatable code segment of module `allocmod` must be allocated in the range 2180H to 2200H and that the relocatable code segment of module `main` must be allocated in the range 2201H to 2300H.

```
rl196 main.obj, allocate.obj, srt.obj to main.lnk noas  
rom(2180H-2200H(allocmod), 2201H-2300H(main),  
0E000H-0FFFFH)
```

2. In this example, the following memory sections are considered as ROM-type sections: 2000H–2FFFFH, 3000H–4000H, 5001H–5FFFFH. After all absolute segments are allocated, `mod1`'s code and constants are allocated within 2000H–2FFFFH, `mod7`'s code and constants also have to be allocated within the same range, but only after `mod1`'s allocation. The code for `mod3` and constants are allocated within 5001H–5FFFFH, then the rest of the relocatable code and constant segments are allocated within the free memory sections of the range.

```
rom( 2000H-2FFFFH(mod1,mod7), 3000H-4000H, 5001H-5FFFFH(mod3) )
```

3. In this example, the following memory sections are considered as ROM-type sections: 2000H-2FFFH, 3000H-4000H, 15001H-15FFFFH. After all of the absolute segments have been allocated, mod1's code and constant are allocated within 2000H-2FFFH. After mod1's allocation, mod7's near constants are allocated somewhere within that range. mod3's far code and far constants are allocated within the 15001H-15FFFFH range, as well as mod7's far code and far constants. The rest of the relocatable code and constant segments are then allocated within the free memory sections of the address range.

```
rom( 2000H-2FFFH(mod1,mod7(const)), 3000H-4000H,  
    15001H-15FFFFH(mod3,mod7(code,farconst)) )
```



romcode  
romdata

# romcode

## Function

Specifies designated ROM sections for CODE, HIGHCODE and FARCODE segments.

## Syntax



Select the EDE | Linker Options... menu item. Enter a ROM address range in the ROMCODE field in the Memory tab.



```
romcode(rom_section [(module_list [(rom_seg_list)] )] [,...])
```

where:

*rom\_section* is an address range specifying the starting address and the ending address of available ROM separated by a hyphen.

*module\_list* is an optional list of valid module names.

*rom\_seg\_list* is an optional list of code segments (code, high code or far code) found in the module. Possible segments are code, farcode.

## Abbreviation

rc for romcode

co for code

fc for farcode

## Class

Locating control

## Default

romcode(2000H-0FFFFH) for 16-bit models

romcode(0FF2000H-0FFFFFFFH) for 24-bit models

## Description

Use this control to designate the ROM address range for code, high code and far code segments. You must specify the ROM sections in ascending order. Follow these rules for each *start\_address* - *end\_address* pair:

- The *start\_address* must be greater than the previous *end\_address*. The minimum *start\_address* is 0H.
- The *end\_address* must be greater than or equal to its *start\_address*. The maximum *end\_address* is 0FFFFFH or 0FFFFFFFH for far code or high code segments.

When you specify a *module\_list* with this control, the linker allocates the relocatable code segments of the specified modules within the address range specified. However, if a module name in the *module\_list* is followed by an explicit *rom\_seg\_list*, then only the code segments specified in the *rom\_seg\_list* for this module is allocated in the specified memory range. The linker performs the memory allocation for these segments from left to right as the modules appear in the list. Because of the fragmentation that results from the scattering of absolute segments in the memory section, the actual order of the segments in physical memory does not necessarily match the order in which they were appear in the command line. You cannot specify *stack* in the *romcode* control.

The linker allocates the relocatable code segments after all of the absolute segments are allocated. This process has two steps:

1. Code segments for the modules specified in the *romcode* control and/or *rom* control are allocated so that each segment is within its specified ROM section.
2. The rest of the code segments are allocated in the remaining free ROM.

The second step allocates any code segments not specified for allocation by the first step.

## Examples

1. In this example all CONST segments are located in the range 0E000h-0E7FFh. All CODE segments are located in the range 2000h-3FFFh. All DATA and REGISTER segments are located in the range 1Ah-1FFFh (default range). Note that the linker knows where to find `cstart.obj` and `c96.lib` if they are not in the current directory. The MODEL in this example is KB.

```
r1196 cstart.obj, hello.obj, c96.lib  
romdata(0e000h-0e7ffh) romcode(2000h-3fffh)
```

2. In this example all CONST segments are located in the range 2000h-57FFh. All CODE segments are located in the range 2000h-3FFFh. All DATA and REGISTER segments are located in the range 1Ah-1FFFh (default range). The MODEL in this example is KR.

```
r1196 cstart.obj, hello.obj, c96.lib  
md(kr) romdata(02000h-057ffh)  
romcode(2000h-3fffh) inst
```



```
rom  
romdata
```



# romdata

## Function

Specifies designated ROM sections for CONST and FARCONST segments.

## Syntax



Select the EDE | Linker Options... menu item. Enter a ROM address range in the ROMDATA field in the Memory tab.



```
romdata(rom_section [(module_list [(rom_seg_list)] )] [,...])
```

where:

*rom\_section* is an address range specifying the starting address and the ending address of available ROM separated by a hyphen.

*module\_list* is an optional list of valid module names.

*rom\_seg\_list* is an optional list of constant segments found in the module. Possible segments are *const*, *farconst*.

## Abbreviation

rd for romdata

ko for const

fk for farconst

## Class

Locating control

## Default

romdata(2000H-0FFFFH) for 16-bit models

romdata(0FF2000H-0FFFFFFFH) for 24-bit models

## Description

Use this control to designate the ROM address range for constant and far constant segments. You must specify the ROM sections in ascending order. Follow these rules for each *start\_address* - *end\_address* pair:

- The *start\_address* must be greater than the previous *end\_address*. The minimum *start\_address* is 1AH.

- The *end\_address* must be greater than or equal to its *start\_address*. The maximum *end\_address* is 0FFFFH or 0FFFFFFH for far constant segments.

When you specify a *module\_list* with this control, the linker allocates the relocatable constant segments of the specified modules within the address range specified. However, if a module name in the *module\_list* is followed by an explicit *rom\_seg\_list*, then only the constant segments specified in the *rom\_seg\_list* for this module is allocated in the specified memory range. The linker performs the memory allocation for these segments from left to right as the modules appear in the list. Because of the fragmentation that results from the scattering of absolute segments in the memory section, the actual order of the segments in physical memory does not necessarily match the order in which they were appear in the command line. You cannot specify stack in the romdata control.

The linker allocates the relocatable constant segments after all of the absolute segments are allocated. This process has two steps:

1. Constant segments for the modules specified in the romdata control and/or rom control are allocated so that each segment is within its specified ROM section.
2. The rest of the constant segments are allocated in the remaining free ROM.

The second step allocates any constant segments not specified for allocation by the first step.

## Examples

1. In this example all CONST segments are located in the range 0E000h-0E7FFh. All CODE segments are located in the range 2000h-3FFFh. All DATA and REGISTER segments are located in the range 1Ah-1FFFh (default range). Note that the linker knows where to find cstart.obj and c96.lib if they are not in the current directory. The MODEL in this example is KB.

```
rl196 cstart.obj, hello.obj, c96.lib
romdata(0e000h-0e7ffh) romcode(2000h-3fffh)
```

2. In this example all CONST segments are located in the range 2000h-57FFh. All CODE segments are located in the range 2000h-3FFFh. All DATA and REGISTER segments are located in the range 1Ah-1FFFh (default range). The MODEL in this example is KR.

```
r1196 cstart.obj, hello.obj, c96.lib
      md(kr) romdata(02000h-057ffh)
      romcode(2000h-3ffff) inst
```



rom  
romcode

# searchlib

## Function

Specifies search paths for input files.

## Syntax



Select the EDE | Directories... menu item. Add one or more directory paths to the Library Files Path field.



```
searchlib(pathprefix [...])
-L pathprefix [...]
```

where:

*pathprefix* is a string of characters that RL196 prepends to an input file's filename. This string must include any special characters that the operating system expects in a path prefix.

## Abbreviation

sl

## Class

Linking control

## Description

Use this control to specify a list of possible path prefixes for input files.

Each *pathprefix* argument is a string that, when concatenated to a filename, specifies the relative or absolute path of a file (including a device name and directory name, if necessary). RL196 tries each prefix in the order in which they are specified, until a legal filename is found. If a legal filename is not found, RL196 issues an error.



RL196 searches for input files in a specific order. See Section 2.6.2 for more details.

# sfr

## Function

Specifies to include model specific SFR file.

## Syntax



Select the EDE | Linker Options... menu item. Enable or disable the Link Special Function Register object file check box in the Linking tab.



```
sfr | nosfr
-s | -s-
```

## Abbreviation

```
sfr | nosfr
```

## Class

Linking control

## Default

```
nosfr
```

## Description

Use this control to include the model specific SFR object file to the list of input files. The object file has the name `xx_sfrs.obj`, where `xx` is a model as specified with the `model` control.

## Example

Specifying,

```
r1196 mod1.obj, mod2.obj sfr md(kb)
```

is the same as specifying,

```
r1196 mod1.obj, mod2.obj, kb_sfrs.obj md(kb)
```



```
model
```

# stacksize

## Function

Specifies the size of the stack segment

## Syntax



Select the EDE | Linker Options... menu item. Enter a stack size or stack offset in the Specify or modify stack size field in the Locating tab.



```
stacksize( [ { + | - } ] n )  
-ss [ { + | - } ] n
```

where:

*n* must be an even number in the decimal or hexadecimal format. If *n* is supplied as an absolute number, without a preceding sign, it must be less than or equal to 0FFFEH.

## Abbreviation

ss

## Class

Locating control

## Default

stacksize(*total*) if *total* ≥ 6  
or stacksize(6) if *total* < 6

## Description

Use this control to specify or modify the resultant RL196 output module's stack segment. The linker calculates the default stack size by adding the sizes of all stack segments of the input modules. The default stack size is the sum of the sizes of all stack segments of the input modules or 6 bytes, whichever is greater.

To specify a stack size different from the default, indicate the desired size, in bytes, within the parenthesis. The *n* parameter must be an even number in the decimal or hexadecimal format.

For example, the following control line indicates you want a stack size of 256 bytes, in decimal:

```
stacksize(256)
```

You can indicate the same condition using the hexadecimal format, as follows:

```
stacksize(100H)
```

To modify the stack size, specify a signed parameter. This parameter is added to or subtracted from the default total. If the resultant stack size is less than zero or greater than 0FFF<sub>EH</sub>, the linker issues a warning. If you specify an absolute number with no preceding sign, this number overrides the default value.

If you specify a stack size that is smaller than the default, the linker issues a warning. If you specify a stack size and the stack is already absolute, the control has no effect and a warning is issued.

Use `stacksize(+2)` or more when linking to run on the ICE<sup>™</sup>-196PC, ICE<sup>™</sup>-196KB/HX or ICE<sup>™</sup>-196KC/HX in-circuit emulator. The emulator requires two bytes on the program stack in addition to the stack space required by the program.

If you have reentrant procedures, you must use this control. A translator can calculate the stack requirement of a single entry but cannot determine how many times such a procedure is called recursively. In this case, you must anticipate the program behavior and modify the stack size.

### Example

For this example, the stack segment size is 880.

```
r1196 main.obj stacksize(880)
```

# typecheck

## Function

Specifies if type checking is performed.

## Syntax



Select the EDE | Linker Options... menu item. Enable or disable the Perform type checking check box in the Linking tab.



typecheck		notypecheck
-tc		-tc-

## Abbreviation

tc | notc

## Class

Linking control

## Default

typecheck

## Description

When this control is in effect, the linker performs type checking during publics-externals resolution. In case of a mismatch, the linker issues a SYMBOL ATTRIBUTE MISMATCH warning message. The notypecheck control inhibits type checking during the resolution process. This control does not delete the type definition information from the output object file. See purge control.

RL196 disregards the specified value of the compiler control type/notype. The linker performs the type checking even if you specified the control notype during compilation; however, in that case, the compiler renders the symbol types as null symbols. These null symbols appear in the symbol table listing and in the intermodule cross-reference listing.



**Example**

No type checking is performed in this link example:

```
r1196 mod1.obj, mod2.obj notypecheck
```

# uniquemods

## Function

Allow more than one module with the same name.

## Syntax



Select the EDE | Linker Options... menu item. Enable or disable the Allow more than one module with the same name check box in the Linking tab.



<code>uniquemods</code>		<code>nouniquemods</code>
<code>-um</code>		<code>-um-</code>

## Abbreviation

`um` | `noum`

## Class

Locating control

## Default

`uniquemods`

## Description

When this control is in effect, you can use more than one module with the same name. **rl196** locates all modules with the same name together.

## Example

The following example assumes that the module names of `mod1.obj` and `mod2.obj` are the same; both modules are located together:

```
rl196 mod1.obj, mod2.obj um
```

# warning

## Function

Specifies a non-zero exit value when warnings occur.

## Syntax



Select the EDE | Linker Options... menu item. Enable or disable the Continue building process when warning(s) occur check box in the Linking tab.



warning		nowarning
-W		-W-

## Abbreviation

wa		nowa
----	--	------

## Class

Linking control

## Default

nowarning

## Description

When this control is in effect, the linker exits with a non-zero value when one or more warnings are present. When you are running **r1196** from a makefile, **mk196** will stop execution.

## Example

This example generates a non-zero exit value when warning(s) occur:

```
r1196 mod1.obj, mod2.obj warning
```

# window size

## Function

Specifies the desired window size for vertical windowing.

## Syntax



Select the EDE | Linker Options... menu item. Select a Register vertical window size in the Memory tab.



`window size(n)`

`-ws : n`

where:

*n* is window size desired in bytes: 32, 64, 128.

## Abbreviation

WS

## Class

Locating control

## Description

Use this control to specify, in bytes, the vertical window size you want to use. Since the 80C196KC and the 80C196KR microcontrollers have additional registers, vertical windowing provides access to the additional registers using the 8-bit direct addressing mode. The C196 compiler can then use these additional registers for block scope register variables. The vertical window can be subdivided into 3 different sizes: 32 bytes, 64 bytes, or 128 bytes.

During the link, the linker selects the biggest window size based on the last (highest) address occupied by the last register segment. The last occupied address must fall below 80H (128-byte window) or 0C0H (64-byte window) or 0E0H (32-byte window). Otherwise, the linker sets WSR to 0 and takes no action on the additional registers. If you do not specify this control, the linker uses the biggest window size possible, if more than 256 registers are specified using the `registers` control. See Section 2.5.6 for more information on how the linker allocates register and overlay segments in vertical windows.

The linker considers your window size request when selecting the window size. If you specify a window size that is smaller than the biggest possible window size, the window size you specified is used. If you specify a window size that is larger than the biggest possible size, the linker uses its selected window size and issues a warning.

### Example

This example specifies a register space range from 1AH to 01FFH and a window size of 64 bytes.

```
rl196 mod1.obj, mod2.obj, mod3.obj, mod4.obj  
    registers(1AH - 01FFH) windowsize(64)
```



registers

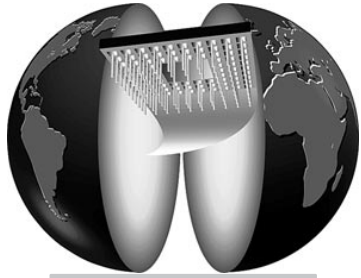
# CHAPTER

## OH196 CONVERTER

---

# 3





# 3

# CHAPTER

The OH196 object-to-hexadecimal converter converts an absolute OMF196 file to a hexadecimal file for use with tools that do not accept this standard Intel object file format.

### 3.1 INVOCATION SYNTAX

The invocation syntax for OH196 is:

```
oh196 [options] abs_objfile [to hexfile]  
oh196 -?  
oh196 -V
```

where:

*abs\_objfile* is an absolute OMF196 file created by RL196.

*hexfile* is the file to contain the hexadecimal output.

**-?** displays the invocation syntax at stdout.

**-V** displays version information at stderr.

*options* is one or more of the following options:

**-o *hexfile*** specify the file to contain the hexadecimal output.

**-p *offset*** add an additional offset to all outputted segments. The maximum value for the offset is +/- 0FFFFFFFH.

**-s *segment*** select the segment which will be outputted into the hex file. The segment can either be *code* (for *CODE* and *FAR\_CODE* segments) or *const* (for *CONST* and *FAR\_CONST* segments). If this option is omitted, all segments will be outputted.

If you do not specify *hexfile*, the output file name takes the name of the root of the object file with a *.hex* extension.

When an error occurs, OH196 generates a fatal error and aborts the processing of the object file. See Chapter 9 for a complete list of error messages and their causes.



3.2 EXAMPLES

- 1. The following example converts the absolute OMF96 file created by RL196, `sort.abs`, to hexadecimal format and places the output into the file `sort.hex`.

```
oh196 sort.abs
```

- 2. The following example converts the absolute OMF96 file `save1.obj` to hexadecimal format and places the output into the file `save.h96`.

```
oh196 save1.obj to save.h96
```

- 3. The following example converts the absolute OMF96 file `tot2.obj` to hexadecimal format and places the output into the file `tot2.hex`.

```
oh196 tot2.obj
```

- 4. The following example converts the absolute OMF96 file created by RL196, `tst.abs`, to hexadecimal format and places only CODE segments into the file `codes.hex`.

```
oh196 -o codes.hex -s code tst.abs
```

- 5. The following example converts the absolute OMF96 file created by RL196, `tst2.abs`, to hexadecimal format and adds an offset of 02000H to all outputted segments into the file `tst2.hex`.

```
oh196 -p 02000H tst2.abs
```

3.3 OUTPUT FILE

In the output file, the record format is:

:	<i>length</i>	<i>offset</i>	<i>type</i>	<i>content</i>	<i>checksum</i>
---	---------------	---------------	-------------	----------------	-----------------

where:

: is the record header.

- length* is the record length. This value occupies one byte (two hexadecimal digits). OH196 outputs records of 16 bytes (32 hexadecimal digits) or less; that is, *length* is never greater than 10H.
- offset* is the absolute address in memory where the data is to be located when loaded by a tool. This field is two bytes long.
- type* is the record type. This value occupies one byte (two hexadecimal digits). The record types are:

Byte Type	Record type
00	Data
01	End of File
02	Extended address
03	Start
04	New (64k) Page Nr

- content* is the information contained in the record. This value occupies up to 16 bytes (32 hexadecimal digits).

If the information in an OMF96 record occupies more than 16 bytes, OH196 divides it into 16-byte hexadecimal records. If any bytes are left over after this operation, OH196 combines those remaining bytes with bytes left over from adjacent records in the OH196 buffer. These adjacent bytes can be part of either a previous or a subsequent record. If no adjacent bytes are available for combination, OH196 puts the remaining bytes in a separate record.

The hexadecimal file always ends with the following end-of-module record:

```
:00000001FF
```

The following is a sample of hexadecimal output:

```
:10208000A1002030B2310F89003030D7F7FE6F0148
:1020900000401CFE6C201CFE4F010040201CFE4C2A
:1020A00020201CFE7F01004020FE7C2220FE5F01DC
:1020B00000402220FE5C22220FE8F0100401CFEF8
:0F20C0008C201CFE9F01004020FE9C222027FE4A
:00000001FF
```

The first record is read as follows:

Field Value	Meaning
10H	This hexadecimal record contains 16 bytes of information.
2080H	The information in this hexadecimal record was at location 2080H in the original OMF96 record.
00H	The record is a data record.
A1...01H	These 16 bytes (32 hexadecimal digits) are the information contained in this record.
48H	This value is the record checksum. The OH196 converter computes the checksum by first adding the binary representation of the previous bytes, starting with 10H to 01H, in this example. OH196 then computes the result of sum modulo 256 and subtracts the remainder subtracted from 256.

The fifth and last hexadecimal record in this example contains one byte (two hexadecimal digits) less than 16 bytes of information. This decrease in size indicates that the length of the original OMF96 record in bytes is one less than a multiple of 16.

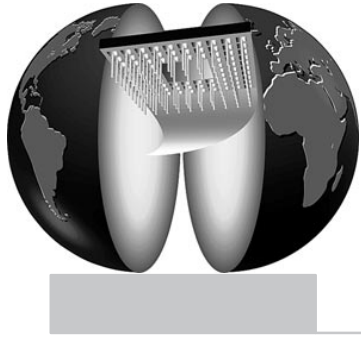
# CHAPTER

**LIB196**  
**LIBRARIAN**

---

# 4





# 4

# CHAPTER

LIB196 allows you to create, modify, and examine library files. This chapter describes the library commands you can use to maintain your library files.

## 4.1 INVOKING LIB196

The following is the syntax for invoking the librarian:

*[pathname] lib196 [options] command*

Where:

*pathname* is the device and/or directory where LIB196 resides.

*options* is an optional list of one or more options. See Section 4.1.1 for a detailed description of each option.

*command* is a command entry discussed in Section 4.2 and Section 4.3.

The librarian responds with the sign-on message then executes the given command. After processing, control returns immediately to the host operating system.

If your command line becomes too long you can use the **-f** option to specify a command file.

### 4.1.1 OPTIONS

The format for a single option is:

*-option\_name* [ {*=* | *:* | *space*} *argument* ]

where:

**-** (minus sign) must be prefixed to every option name.

*option\_name* is the name of the option. This name is case sensitive.

**=**, **:**, or *space* are used to separate the option name from the argument.

*argument* the argument for an option. This is optional.

Below are the detailed descriptions of options.

**-?** Display an explanation of options on `stdout`.



- V** Display version information on `stdout` and stop.
- case** With this option the librarian works in a case sensitive manner.
- f file** Use *file* for command line processing. In this way you can extend the command line. This option can be used more than once.

**4.1.2 CHARACTER SET**

The LIB196 character set consists of the letters A through Z, the digits 0 through 9 and the special characters ?, @, and \_.

**4.2 LIB196 COMMANDS**

Table 4-1 summarizes the LIB196 commands.

Command	Description
<b>a</b> <i>lib</i> { <i>file</i> [ ( <i>module</i> [,...]) ] } [...]	adds modules to a library
<b>c</b> <i>lib</i>	creates a library file
<b>d</b> <i>lib module</i> [,...]	deletes modules from library
<b>x</b> <i>lib module</i> [,...]	extracts modules from libraries
<b>l</b> [ <b>p</b> ] { <i>lib</i> [ ( <i>module</i> [,...]) ]   <i>file</i> [ ( <i>module</i> [,...]) ] } [...]	lists modules contained in libraries or modules, and optionally lists all publics
<b>r</b> <i>lib</i> { <i>file</i> [ ( <i>module</i> [,...]) ] } [...]	replaces modules in a library

Table 4-1: LIB196 commands

### **4.3 COMMAND DESCRIPTIONS**

The remainder of this section explains each LIB196 command in detail. The commands appear in alphabetical order.

Square brackets ([ ]) enclose optional arguments for controls. If you do not specify optional arguments for a particular control, do not use an empty pair of brackets.

Some commands use an optional list of arguments. Separate multiple argument definitions with commas. Brackets surrounding a comma and ellipsis ([ , . . . ]) indicate an optional list.

Curly braces ({ }) indicate that you must pick one of the options provided. See Conventions Used in this Manual at the beginning of the manual for special meanings of type styles used in by this manual.



# a (add)

## Function

Add specified file to the library

## Syntax

**a** *library\_file* { *file* [ ( *module* [...]) ] } [...]

where:

*library\_file* is the name of the library being added to.

*file* is the filename of the module.

*module* is the name of the module being added to the library.

## Description

Use this command to add the specified files to the specified library file. The input filenames can be the names of ordinary object files or object library files. If the input file is an ordinary object file, all modules contained within that file are added to the designated library. The ordinary object file can be produced by a translator, RL196, or the extract command (**x**) of LIB196.

If the input file is a library file, it can be specified with or without a list of module names. If you do not specify any module names, the librarian adds all of the modules contained in the input library to the destination library. If you do specify a list of module names, the librarian adds only those modules specified in the command into the destination library.

If the library does not exist, it is created first.

## Examples

1. This command adds the three files `sin`, `cos`, and `tan` to the destination library `user.lib`.

```
lib196 a user.lib sin, cos, tan
```

2. This command adds the three modules `mod1`, `mod2`, and `mod3` of the library `lib.tmp` to the destination library `proj.tom`.

```
lib196 a proj.tom lib.tmp(mod1, mod2, mod3)
```

# c (create)

## Function

Create library file

## Syntax

**c** *library\_file*

where:

*library\_file* is the name you give to the new library being created.

## Description

Use this command to create an empty library file with the specified name. If the file already exists, an error message is issued and the command terminates. See Chapter 9 for a complete list of error messages.

## Example

This command creates the empty library file `new.lib`.

```
lib196 c new.lib
```

# d (delete)

## Function

Delete specified module from library

## Syntax

**d** *library\_file module\_name* [...]

where:

*library\_file* is the name of the existing library.

*module\_name* is the name of the module being deleted.

## Description

Use this command to remove the specified modules from the designated library file. You can delete only one module at a time from the library. If any of the elements specified for deletion cannot be located, LIB196 issues a warning.

## Example

This command deletes the modules `truth` and `value` from the library `new.lib`.

```
lib196 d new.lib truth,value
```

# x (extract)

## Function

Builds an ordinary object file from the specified files and library members.

## Syntax

```
x library_file module_name [...]
```

where:

*library\_file* is the name of the existing library.

*module\_name* is the name of the module being extracted.

## Description

Use this command to build an ordinary object file from the specified files and library members. The extracted files are not deleted; they are copied to destination object files for each extracted module.

## Example

The modules `worth` and `free` are copied from `new.lib` and placed in `worth.obj` and `free.obj`.

```
lib196 x new.lib worth,free
```

# l / lp (list)

## Function

Print the name of the modules inside the library or the names inside the module.

## Syntax

```
l | lp { library_file [ ( module [...] ) ] |
        file [ ( module [...] ) ] } [...]
```

where:

*library\_file* is the name of the library.

*file* is the filename of the module.

*module* is the name of the module residing in the library.

## Description

Use this command to print the names of the modules, and optionally, the names of the public symbols. The librarian directs the listing to the console output. Use **lp** to list all public symbols contained in those modules with the module names.

## Examples

1. List all module names in the library `user.lib`.

```
lib196 l user.lib
```

2. List all public symbols in the module `temp` in the library `user.lib`.

```
lib196 lp user.lib(temp)
```

# r (replace)

## Function

Replace designated object in library file.

## Syntax

```
r library_file { file [ (module [...]) ] } [...]
```

where:

*library\_file* is the name of the existing library.

*file* is the name of file containing the module.

*module* is the name of the new module.

## Description

Use this command to replace object module or modules in the designated library file with a new version. If the designated module does not exist in the library file, the librarian adds the new version to the library.

## Examples

1. Worth and free are replaced in the library new.lib.

```
lib196 r new.lib worth, free
```

2. The newer version of time in module counter replaces the older version in library user.lib.

```
lib196 r user.lib counter(time)
```



LIBRARIAN

# CHAPTER

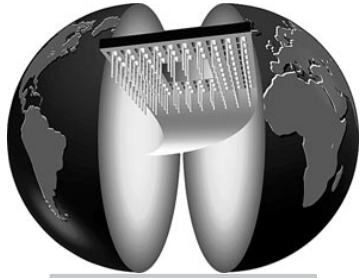
# 5

## USING THE FPAL96 LIBRARY

---







# 5

# CHAPTER

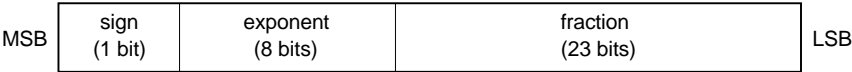
This chapter describes the different external data formats FPAL96 recognizes, the naming and parameter-passing conventions the library follows, and the control variables FPAL96 uses when performing its operations. This chapter also explains how to link the FPAL96 library to your application.

**5.1 DATA FORMATS SUPPORTED**

FPAL96 supports three external data formats: single-precision floating-point numbers, integers, and decimal floating-point numbers. Each format is described in the following sections.

**5.1.1 FLOATING POINT NUMBERS**

A floating point value occupies four contiguous memory bytes that can be viewed as 32 contiguous bits. The bits are divided into fields, as follows:



OSD1181

Where:

- sign** is a 1-bit field that contains the value 0 if the floating point value is positive; 1 if the floating point value is negative.
- exponent** is an 8-bit field that contains a value offset by 127; in other words, the actual exponent can be obtained from the exponent field by subtracting 127. An exponent field of all 0s or all 1s represents special cases that are described in the following section. Otherwise, the floating point is called normalized.
- fraction** is a 23-bit field that contains the fractional part of the floating point value, represented in binary scientific notation.

The following examples illustrate these concepts:

1. The following binary number is equivalent to the decimal value of 10.25:  
1010.01B

The dot (.) in this number is a binary point. In binary scientific notation, the same number can be represented as:

$$1.01001\text{B} \times 2^3$$

The binary point is to the immediate right of the most-significant digit. The digits 01001 represent the fraction, and 3 is the exponent.

The complete 32-bit representation is:

0 10000010 010010000000000000000000

Where:

- The sign bit is 0 because the value is positive.
- The exponent field contains the binary equivalent of  $127 + 3 = 130$ .
- The leftmost digits of the fraction field are 01001, and the remainder of this field is all 0s.

The contents of the four contiguous memory bytes are:

highest address:	01000001
	00100100
	00000000
lowest address:	00000000

2. In binary, the fraction  $1/16$  or 0.0625 is represented as:

0.0001B

In binary scientific notation, the fraction  $1/16$  is represented as

$$1.0000\text{B} \times 2^{-4}$$

The actual exponent,  $-4$ , is represented as 123 (the sum of  $+127$  and  $-4$ ), and the fraction field contains all 0s.

The most-significant digit of the fraction field is not actually represented, because by definition, this digit contains a value of 1 unless the floating point number is 0 or denormalized. Section 5.1.1.1 discusses representation of 0 and denormalized values.

Floating point values can range approximately from  $8.43 \times 10^{-37}$  to  $3.38 \times 10^{38}$ . The greatest finite number is  $2^{104} * (2^{24}-1)$ , which is approximately  $3.38 \times 10^{38}$ . The smallest normalized positive number is  $2^{-126}$ , which is approximately  $8.43 \times 10^{-37}$ . The smallest denormalized positive number is  $2^{-149}$ , which is approximately  $1.4 \times 10^{-45}$  in scientific notation.

ASM196 uses the floating point number format for the real data type, and C196 uses this format for the float, double, and long double data types. You can use the floating point number format in load, store, unary, and binary operations.

5.1.1.1 SPECIAL FLOATING POINT NUMBERS

Special floating point numbers are identified by an exponent field with all 0 or all 1 values. The four kinds of special floating point numbers are Not-a-Number (NaN), denormal, infinity, and zero. The special floating point type is determined by the relationship between the fraction and exponent in the single-precision format. Table 5-1 summarizes this relationship.

Exponent	Fraction	Value	Name
all 0s	zero	$(-1)^s * 0$	zero
all 1s	zero	$(-1)^s * \text{infinity}$	infinity
all 0s	non-zero	$(-1)^s * (0.F) * 2^{-126}$	denormal
all 1s	non-zero	NaN	Not-a-Number

Table 5-1: Relationship between exponent and fraction

Zeros

A zero is a number whose exponent and fraction fields contain all 0s. A sign bit of 0 indicates the number is approaching zero from the positive numbers. You can write this number as +0 or simply 0. A sign bit of 1 indicates the number is approaching zero from the negative numbers. Write this number as -0. In other words, if a small positive number is rounded to 0, the result is +0. If a small negative number is rounded to 0, the result is -0. Table 5-2 summarizes the effect of performing operations on either +0 or -0.

Type	Operation	Result
addition	+0 plus +0	+0
	−0 plus −0	−0
	+0 plus −0, −0 plus +0	see notes
	+X plus −X, −X plus +X	see notes
	#0 plus #X, #X plus #0	#X
subtraction	+0 minus −0	+0
	−0 minus +0	−0
	+0 minus +0, −0 minus −0	see notes
	+X minus +X, −X minus −X	see notes
	#0 minus #X	!X
multiplication	+0 * +0, −0 * −0	+0
	+0 * −0, −0 * +0	−0
	+0 * +X, +X * +0	+0
	+0 * −X, −X * +0	−0
	−0 * +X, +X * −0	−0
	−0 * −X, −X * −0	+0
	+X * +Y, −X * −Y	+0 (when underflow)
	+X * −Y, −X * +Y	−0 (when underflow)
division	#0/#0	invalid operation
	#X/#0	zero divide
	+0/+X, −0/−X	+0
	+0/−X, −0/+X	−0
	−X/−Y, +X/+Y	+0 (when underflow)
	−X/+Y, +X/−Y	−0 (when underflow)
remainder	#0 REM #0	invalid operation
	#X REM #0	invalid operation
	+0 REM #X	+0
	−0 REM #X	−0

Type	Operation	Result
sqrt	−0	−0
	+0	+0
NOTES: The sign of zero is determined by the rounding mode as follows: + for nearest, up or truncate − for down X and Y denote any nonzero operands. # denotes either sign (+ or −). ! denotes the complement of the sign of X.		

Table 5-2: Operations executed with zero operands

### Infinites

Infinity is represented by the exponent field containing all 1s and the fraction field containing all 0s. A sign bit of 0 indicates +infinity, and a sign bit of 1 indicates −infinity. Table 5-3 summarizes the effect of performing operations on either infinity value.

Type	Operation	Result
addition	+infinity plus +infinity	+infinity
	−infinity plus −infinity	−infinity
	+infinity plus −infinity	invalid operation
	−infinity plus +infinity	invalid operation
	#infinity plus #X	\$infinity
	#X plus #infinity	\$infinity
subtraction	+infinity minus −infinity	+infinity
	−infinity minus +infinity	−infinity
	+infinity minus +infinity	invalid operation
	−infinity minus −infinity	invalid operation
	#infinity minus #X	\$infinity
	#X minus #infinity	!infinity
multiplication	#infinity * #infinity	⊕infinity
	#infinity * #X	⊕infinity
	#X * #infinity	⊕infinity
	#0 * #infinity	invalid operation
	#infinity * #0	invalid operation

Type	Operation	Result
division	#infinity / #infinity	invalid operation
	#infinity / #X	⊕infinity
	#X / #infinity	⊕0
remainder	#infinity REM #infinity	invalid operation
	#infinity REM #X	invalid operation
	#X REM #infinity	#X
sqrt		
	+infinity	+infinity
NOTES: X denotes a finite operand. # denotes either sign ( + or -). \$ denotes the sign of the original infinity operand. ! denotes the complement of \$. ⊕ denotes the exclusive OR of the original operand signs.		

Table 5-3: Operations executed with infinity operands

Denormalized Numbers

A normalized number is a number whose most-significant digit is a 1. This digit does not appear in the actual representation. Therefore, the smallest positive normalized number that can be represented using the concept is  $1.0B \cdot 2^{-126}$ . The actual exponent,  $-126$ , is represented as 1 (i.e.,  $127-126$ ), and the fraction field contains all 0s.

By assuming that the most-significant digit is a 0, the denormalized floating point format allows smaller numbers to be represented. You can represent a denormalized number by setting the exponent field to all 0s. The number is assumed to have an exponent of  $-126$ , and its fraction does not have a hidden leading 1. To distinguish it from zero, the fraction field must not be all 0s.

For example, the number  $2^{-130}$  can be represented as  $0.0001B \cdot 2^{-126}$ . Therefore, this value is represented as follows:

- The sign bit is 0, because the value is positive.
- The exponent field contains all 0s to indicate a denormalized number.
- The leftmost digits of the fraction field are 0001, and the remainder of this field is all 0s.



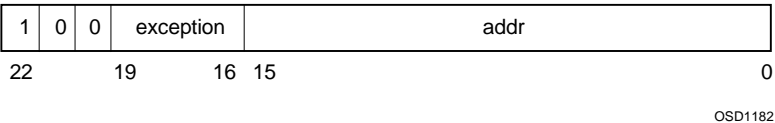
Any access to denormalized operands causes a denormal exception. However, if this exception is masked (set to 1), FPAL96 continues to use the denormalized operand.

***Not-a-Numbers (NaNs)***

A floating point number is called a Not-a-Number (NaN) if its exponent is all 1s and its fraction is nonzero. The two kinds of NaNs are signalling NaNs and quiet NaNs.

The most-significant bit of the fraction field of a signalling NaN is a 0. You can set the remaining bits in the fraction to suit your own purposes. Attempting to load or operate on a signalling NaN raises an invalid-operation exception. Signalling NaNs, however, are useful in detecting operations on uninitialized variables. To do so, you must supply an exception handler to manage the use of an uninitialized variable, then enable the invalid-operation exception. If you set an uninitialized floating point to a signalling NaN, you can determine when an invalid floating point number was used as an operand.

The most-significant bit of the fraction field of a quiet NaN is a 1. FPAL96 sets the remaining bits in the fraction area to indicate the exception type and the address where the exception occurred. The fraction field has the following format:



Where:

*exception* is a 4-bit field that contains the exception number. Table 5-4 contains the list of exception numbers.

*addr* is a 16-bit field that contains the address of the instruction following the FPAL96 function call that caused the exception.





Note that the *addr* field contains 16 bits and uniquely identifies the location of the FPAL96 call in all cases except for an 80C196NT component executing in extended mode. For an 80C196NT component in compatible mode, *addr* gives the low 16 bits of the location, and the high 8 bits of the 24-bit address are assumed to be 0FFH. For an 80C196NT component in extended mode, the high 8 bits of the 24-bit address are not recorded, but in most cases the low 16 bits of the address provide sufficient information for you to deduce the exact location of the call.

Table 5-4 describes the exceptions that cause a quiet NaN and lists the corresponding code that is moved into the FPACC when a quiet NaN occurs.

Code Number	Exception
0000	The operand is a signalling NaN.
0001	A multiplication of zero by infinity and FPACC was zero.
0010	A multiplication of zero by infinity and FPACC was infinity.
0011	A division of zero by zero occurred.
0100	A division of infinity by infinity occurred.
0101	An addition or subtraction leads to subtraction of infinities with the same sign.
0110	The result is the square root from the negative nonzero.
0111	The result is the remainder when FPACC was infinity.
1000	The result is the remainder from a division by zero.
1001	Conversion from floating point to integer or decimal occurred when a true result cannot be obtained.
1010	In decimal-to-floating point conversion with unmasked overflow or underflow, the exponent is too large or too small for conversion.
1011	Comparison with unordered operands occurred.
1100	The FPACC is a signalling NaN.

Table 5-4: Quiet NaN exceptions

If one of the operands is a quiet NaN, the result of the operation is the quiet NaN. If both operands are quiet NaNs, the result of the operation is the quiet NaN that was originally in the FPACC.

### 5.1.2 INTEGERS

Integer values are represented as long (32-bit) integers in two's complement format. Internally, integers are arranged so that the least-significant byte occupies the lowest address, the second least-significant byte occupies the second lowest address, and so on. You can use an integer only as an operand during a load operation or as the result of a store operation to convert between integer and floating point format.

### 5.1.3 DECIMALS

Decimal floating-point numbers are represented as two consecutive binary numbers: a 32-bit integer that represents the mantissa and an 8-bit integer that represents the exponent. To use decimals in your program, use the declarations shown below.

For ASM196, declare the following symbols in the data segment:

```
dseg
decimal_type:
    mantissa:    dsl 1
    exponent:    dsb 1
```

For C196, declare the following structure:

```
struct decimal_type {
    long int mantissa; /* signed */
    short char exp;    /* signed */
}
```

The value represented by a decimal floating-point number is the result of the operation  $M \cdot 10^E$ . You can use a decimal number only as an operand in a load operation or as the result of a store operation to convert between decimal and floating point format.

## 5.2 CONVENTIONS

This section describes the naming and parameter-passing conventions used by the FPAL96 library.

### 5.2.1 NAMING CONVENTIONS

FPAL96 uses several naming conventions to help you remember the names of the routines you need:

- All FPAL96 procedures start with the prefix `fp`.
- All load operations start with `fpld`.
- All store operations start with `fpst`.
- Operations using floating point operands have no suffix, for example, `fpadd`.
- Operations using integer operands have the suffix `int`, for example, `fpldint`.
- Operations using decimal operands have the suffix `dec`, for example, `fpstdec`.

FPAL96 also internally uses some public procedures and variables that start with `fp`. To avoid duplicating FPAL96 names, do not define public symbols beginning with the letters `fp`.

### 5.2.2 PARAMETER PASSING

The floating-point library uses the following parameter-passing conventions:

- FPAL96 pushes parameters onto the stack in left-to-right order. A `byte` parameter (8 bits) is pushed onto the stack as the low-order byte of a word. A `word` parameter (16 bits) is pushed as a word. A `parameter longint` or `real` (floating point) parameter (32 bits) is pushed as two words; the high-order word is pushed first.
- FPAL96 returns function results to a global double-word register called `PLMREG`. An ASM196 program that uses FPAL96 services must define this register as external as shown below:

```
rseg
extrn PLMREG:dword
```

If the register contains a word value, the low-order word is used. Otherwise, the full register is used.

- FPAL96 passes basic parameters, such as byte, word, integers, long integers, floating point numbers, etc., by value.
- FPAL96 passes structure parameters, such as decimal numbers and save areas, by address, that is, as a near or far pointer.

### **5.3 FPAL96 CONTROL VARIABLES**

This section describes three of the variables FPAL96 uses when performing floating-point operations.

#### **5.3.1 FLOATING-POINT ACCUMULATOR**

All of the floating-point operations use a data structure called the floating-point accumulator (FPACC) as one of the operands, or as the place to store the result, or both. The FPACC value represents the result of the last FPAL96 operation in floating point format, except for store operations. Store operations do not change the FPACC but rather convert and store it in an external format.

#### **5.3.2 BUILT-IN VARIABLES**

FPAL96 has two built-in variables that you can use in processing exceptions: the control word and the status word.

You can set the control word to determine the response to various exception conditions and to establish the rounding you desire. Section 5.3.2.1 discusses the format of the control word.

The status word is divided into two bytes. The first byte, called the error byte, indicates any pending exceptions. The second byte shows the status of the FPACC after an operation is executed. This second byte also holds the result of an `fpcomps` or `fpcompq` operation, described in Chapter 6. You can use the control word and status word with your exception handler to continue a flagged operation or to analyze results when debugging.

The control word consists of 1

TM is the invalid-operation mask

DM is the denormal mask

7M is the zero-divide mas

OM is the overflow mask

ITEM is the underflow mask

RM is the precision mask

BC is the round control

The settings of bits 0 through 5 de

Bits 10 and 11 determine how rounding is done. The combination codes and their meanings are as follows:

01                    round down

10                  round up

11 truncate

Bounding modes are

The other bits in the control word are reserved for future implementations. FPAL96 initializes the control word to 003FH (i.e., round-to-nearest mode and all exceptions masked). You can change this value with the `fpldcw` function described in Chapter 6.

***Rounding Modes***

FPAL96 performs all of its operations with infinite precision. When an infinitely precise result cannot be represented by the given format, FPAL96 performs rounding. FPAL96 supports four different rounding modes: round to the nearest, round down, round up, and truncate. You choose the rounding mode that best suits your application by setting the control word (bits 10 and 11) appropriately. If  $r$  is the infinitely precise result, and  $r\_up$  and  $r\_down$  are exactly representable numbers that lie closest to  $r$  (i.e.,  $r\_down < r < r\_up$ ), then  $r$  is rounded as described in Table 5-5.

Rounding mode	Bit setting in Control Word	Rounded result
round to nearest	00	FPAL96 delivers the result closer to $r$ of $r\_down$ or $r\_up$ . If the numbers are equally close, FPAL96 delivers the number with zero as its least-significant bit.
round down	01	FPAL96 delivers the $r\_down$ (toward -infinity) result.
round up	10	FPAL96 delivers the $r\_up$ (toward +infinity) result.
truncate	11	FPAL96 delivers the smaller number of $r\_down$ (round toward 0) or $r\_up$ .

*Table 5-5: FPAL96 rounding modes*

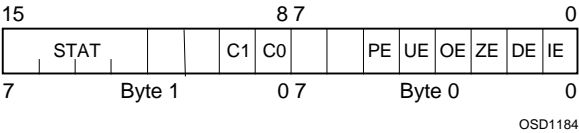
Rounding introduces an exception in the result. The exception is less than one unit in the last place to which the result is rounded. Round to nearest is the default mode and is suitable for most applications. This mode provides the most accurate and statistically unbiased estimate of the true result. Truncate helps control conversions to integers that follow conventions embedded in programming languages such as FORTRAN and C.

Round up and round down are directed rounding, which you can use to implement interval arithmetic. Interval arithmetic generates a certifiable result independent of the occurrence of rounding and other exceptions. You can compute the upper and lower bounds of an interval by executing the algorithm twice, rounding up in one pass and rounding down in the other.

Section 7.4 describes the case when  $x$  is infinity or approaches infinity.

5.3.2.2 STATUS WORD

The status word also contains 16 bits and is structured in the following format:



The bits stand for the following functions:

IE is the invalid operation.

DE is the denormal.

ZE is the zero divide.

OE is the overflow.

UE is the underflow.

PE is the precision.

C0 and C1 are set by `fpcomps` or `fpcompq` (see Chapter 6).

STAT is a 4-bit field that indicates the FPACC status as defined in Table 5-5-6.

STAT Value	FPACC Value
0000	+0
0001	+infinity
0010	+denormal
0011	signalling NaN
0100	+normal
0101	(reserved)
0110	(reserved)
0111	quiet NaN
1000	-0
1001	-infinity
1010	-denormal
1011	signalling NaN
1100	-normal
1101	(reserved)
1110	(reserved)
1111	quiet NaN

*Table 5-6: STAT field in the status word*

### **Error Byte**

Byte 0 of the status word is called the error byte. When an exception occurs after a floating-point operation, FPAL96 sets the corresponding exception flag bit in the error byte to 1, then it checks to see if the corresponding bit in the control word is masked (1) or unmasked (0) and calls the default or your exception handler respectively.

### **Processing Exceptions Using the Status Word**

Your exception handler normally reads the status word using `fpstsw` when using ASM196. The C196 compiler implicitly call `fpstsw` when testing the status word to determine what caused the exception. To determine the exception, do the following code:

```
You can determine the cause by ANDING the error byte with
NOT( control-word ).
```



```
control_word = fpstcw();
exception_bit = struct_name.Status_Word & (~control_word);
/* You can find the structure declaration in the fpal96.h file
*/
```

The lower byte of *exception\_bit* variable contains the value of the error byte. For example, if the last operation generated an invalid-operation exception, *exception\_bit* contains the value 0000000000000001. Process the exception accordingly, then reset the exception bit to 0 in the error byte.

The exception bits 0 through 5 are sticky bits. That is, once an exception has occurred, the corresponding bit remains set until you explicitly reset it using the *fpclb*, *fprstor*, or *fpinit* function. See Chapter 6 for an explanation of each function. See Chapter 7 for instructions on how to write your own exception handler.

## 5.4 DECLARATION AND LINKAGE

The following sections show you how to declare the floating-point functions you want to use and how to link the FPAL96 library to your application program. See Chapter 6 for complete examples for each function.

### 5.4.1 DECLARING FLOATING-POINT FUNCTIONS

To use the floating-point libraries, you must declare the functions as externals inside your program. The following sections show you how to declare them as externals.

#### 5.4.1.1 IN AN ASM196 PROGRAM

Since FPAL96 uses the PL/M-96 calling convention, you must use the following rules when calling FPAL96 functions from ASM196 routines:

- Use the PL/M-96 calling sequence, given for each procedure, as a basis.
- Declare FPAL96 functions as externals using the *extrn* directive.
- For an 80C196NT component operand in extended (far code) mode, use the extended call (*ecall*) instruction. For all other cases, use the *call* instruction.

- Parameters are pushed onto the stack in a left-to-right order.
- A parameter of type `byte` or `shortint` is pushed as a word and is contained in the low-order byte of this word.
- A parameter of type `dword`, `longint`, or `real` is pushed as two words: first the high-order word then the low-order word. Thus, the low-order word has the lower address.
- Any address (pointer) parameter is pushed as a word. NT near pointers are pushed as words, far pointers are pushed as dwords.
- Typed FPAL96 function return their value in `PLMREG`. `PLMREG` must be defined as external using the `extrn` directive.
- You can specify operands using any valid addressing modes, except for destination operands which cannot use immediate.



See the *80C196 Assembler User's Guide*, listed in *Related Publications*, for more information on ASM196 conventions, instructions, and directives.

#### **5.4.1.2 IN AN C196 PROGRAM**

The C196 compiler automatically translates expressions containing floating point variables or constants to the correct sequence of FPAL96 function calls for binary operations. These operations include addition, subtraction, multiplication, division, modulo, and comparison.

To explicitly use any of the FPAL96 functions, you need only to include the `fpa196.h` file inside your program using the `#include` directive. *C: A Reference Manual*, listed in *Related Publications*, tells you how to use the `#include` directive.

#### **5.4.2 SELECTING THE CORRECT LIBRARY**

The FPAL96 library comes in multiple versions. One library for each microcontroller family.

Based on the `model` control given to the linker, the linker searches for the correct `fpa196.lib` library in the library search path. See Section 2.6.2 for more information on the the library search path.

### 5.4.3 INITIALIZING THE FLOATING-POINT LIBRARY

Before you can use floating point routines in your C or assembly source, the FPAL96 library must be initialized.

In assembly, you can do this by a call to `fpinit`:

```
cseg
extrnfpinit
call fpinit      ; Initialize FPAL96
...
```

In C, it depends on the OMF version what you should do. In the default situation, OMF V3.2, you do not have to put anything special in your C source. The initialization is done in the `cstart.obj` module. So, with OMF V3.2 you only have to link `cstart.obj` with your object modules and libraries.

In older OMF versions (< V3.2), you have to call the function `fpinit()`, which is normally done in the C function `_main()`. See the file `_main.c` delivered with the product for more details.

### 5.4.4 LINKING THE FLOATING-POINT LIBRARY

Because FPAL96 is an 80C196 application library, you must use RL196 to link it with your application. You must specify the name of the library at the end of the input file list to the RL196 linker. The linker scans your program and links only those procedures you need. Procedures you do not use, even if they are declared as externals in your program or in one of your include files, are not linked to your application. The following example shows the RL196 invocation line:

For 16-bit components:

```
rl196 input_list, fpal96.lib [to output_file]
model(kc) [controls]
```

For a 24-bit component in compatible code mode:

```
rl196 input_list, fpal96.lib [to output_file]
model(nt-c) [controls]
```

For a 24-bit component in extended code mode:

```
rl196 input_list, fpal96.lib [to output_file]
model(nt-e) [controls]
```

Where:

*input\_list* is a list of object files or library files. If you are working with C196, the list must include `c96.lib`.

*output\_file* is the optional file that receives the output module.

*controls* is an optional list of RL196 controls.

## 5.5 EXAMPLES USING FPAL96 ROUTINES

For the examples the parameters are defined as follows:

*anyVar* A variable of type *any*, where *any* can be real, long (integer), integer, short (integer), word, byte, or decimal.

*anyOpr* An operand of type *any*.

*.anyOpr* The near or far address of an operand. That is, the full 24-bit address of an operand for an 80C196NT component operated in `compatible` or `extended` mode; otherwise, it contains the 16-bit address of an operand.

*CRef* The near or far address of a code entry point. That is, it is equivalent to the "@" operator producing a 24-bit address for an 80C196NT component in `extended` mode, and to the "." operator producing a 16-bit address for any other case.

*CODEPTR* A near or far pointer to a code entry point. That is, it is equivalent to "pointer" which holds a 24-bit address for an 80C196NT in `extended` mode, and to "address" which holds a 16-bit address for any other case.

This terminology is also used in the sections delineating usage of the various FPAL96 procedures unless otherwise noted.

### Examples

The following examples of FPAL96 function invocation use the conventions described in Section 5.2. You can find additional examples under the description of each function in Chapter 6.

1. The following ASM196 example for a non-80C196NT program invokes the `fpldddec` function:

```

dseg
shortopr: dsb 1
decopr:    dsl 1          ; Mantissa.
          dsb 1          ; Exponent.

cseg
extrn fpinit, fpldddec
call fpinit          ; Initialize FPAL96.
push shortopr        ; ShortOpr and following byte
                    ; are pushed onto the stack.
push #decopr         ; Address of DecOpr is pushed
                    ; onto the stack.
call fpldddec        ; Convert to floating point
                    ; in FPACC.

```

The same function is written in C196 as follows:

```

#include <fpal96.h> /* Include header file. */
short shortopr      /* Number of digits to the
                    right of the decimal
                    point. */
DecimalType decopr; /* DecimalType is defined in
                    fpal96.h */

main()
{
    shortopr = 3; /* 3 digits after the decimal
                  point. */
    fpldddec(shortopr, &decopr);
}

```

ASM196 calling sequence for an 80C196NT program with far code and arguments in a near data segment:

```

dseg          near
ShortOpr:     dsb 1
DecOpr:       dsl 1 ;Mantissa
              dsb 1 ;Exponent

cseg          far
extrn         fpldddec:entry

push ShortOpr    ;ShortOpr and following byte are pushed
push 0           ;High order bits of address of DecOpr
                ;are zero.
push #DecOpr     ;Low order bits of address of DecOpr
                ;are pushed.
ecall fpldddec   ;Convert to floating point in FPACC.
```

ASM196 calling sequence for an 80C196NT program with far code and arguments in a far data segment:

```

rseg
reg:          dsb 1

dseg          far
ShortOpr:     dsb 1
DecOpr:       dsl 1 ;Mantissa
              dsb 1 ;Exponent

cseg          far
extrn         fpldddec:entry

elddb reg, ShortOpr
push reg        ;ShortOpr and following byte are pushed
push #msw(DecOpr) ;High order bits of address of DecOpr
                ;are pushed.
push #lsb(DecOpr) ;Low order bits of address of DecOpr
                ;are pushed.
ecall fpldddec   ;Convert to floating point in FPACC.
```

2. The following ASM196 example for non-80C196NT program, or for an 80C196NT program with near code and variables in a near data segment invokes the `fpst` function:

```

rseg
extrn      plmreg:word

dseg
realvar:   dsr 1

cseg
extrn      fpinit, fpst

call fpinit          ; Initialize FPAL96.
call fpst            ; Store FPACC in PLMREG.
st plmreg, realvar    ; Move bits 0-15.
st plmreg+5, realvar+5 ; Move bits 16-31.

```

The same function is written in C196 as follows:

```

#include <fpal96.h> /* Include FPAL96 header file. */
float realvar;

main()
{
    realvar = 3.1;      /* Initialize realvar */
    fpld(realvar);      /* Load realvar to FPACC */
}

```

ASM196 calling sequence for an 80C196NT with near code and variables in a far data segment:

```

rseg
extrn      plmreg:word

dseg      far
RealVar:   dsr 1

cseg      near
extrn      fpst:entry

call fpst          ;Store FPACC in PLMREG
est plmreg, RealVar ;Move bits 0-15
est plmreg+5, RealVar+5 ;Move bits 16-31

```

3. The following ASM196 calling sequence for a non-80C196NT program, or for an 80C196NT program with near code and arguments in a near data segment invokes the `fpadd` function:

```
dseg
realvar:  dsr 1

cseg
extrn      fpinit, fpadd

call fpinit      ; Initialize FPAL96.
push realvar+5    ; Push high-order word (sign
                  ; + exp + 7 bits of the
                  ; fraction) onto the stack.
push realvar      ; Push low-order word (16
                  ; least-significant bits of
                  ; the fraction) onto the
                  ; stack.
call fpadd        ; Add to FPACC.
```

The same function is written in C196 as follows:

```
#include <fpal96.h> /* Include FPAL96 header file */

float realvar;      /* Define floating point variable */

main()
{
    fpld(1.5);       /* Store 1.5 in FPACC */
    fpadd(5.3);      /* Add 5.3 to FPACC */
    realvar = fpst(); /* Store result in realvar */
}
```

ASM196 calling sequence for 80C196NT program with far code and arguments in a near data segment:

```
dseg      near
RealVar:  dsr 1

cseg
extrn      fpadd:entry

push RealVar+5    ; High-order word (s+exp+7 ms bits of
                  ; frac.)
push RealVar      ; Low-order word (16 ls bits of frac.)
ecall fpadd       ; add to FPACC
```



ASM196 calling sequence for an 80C196NT program with far code and arguments in a far data segment:

```
rseg
reg:      dsw 1

dseg      far
RealVar:  dsr 1

cseg      far
extrn     fpadd:entry

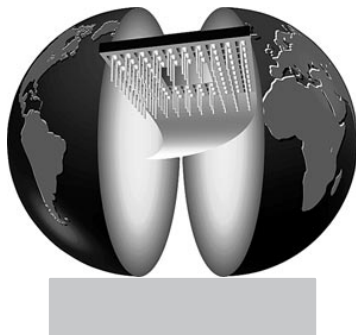
eld reg, RealVar+5 ; High-order word (s+exp+7 ms bits of
                  ; frac.)
push reg
eld reg, RealVar   ; Low-order word (16 ls bits of frac.)
push reg
ecall fpadd        ; Add to FPACC
```

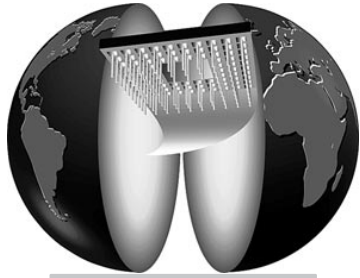
# CHAPTER

# 6

## **FPAL96 FUNCTIONS REFERENCE**

---





# 6

# CHAPTER

## 6.1 INTRODUCTION

FPAL96 operations can be divided in three groups: administrative, load and store, and unary and binary operations. This chapter describes each group and the functions associated within the group.



See Section 6.7 for a description of each function. The functions are arranged in alphabetical order. Chapter 7 describes the cause or causes of each exception.

## 6.2 ADMINISTRATIVE OPERATIONS

The following administrative functions control the execution environment in which FPAL96 works:

<code>fpinit</code>	Initializes FPAL96.
<code>fpldcw</code>	Loads the control word.
<code>fpstcw</code>	Stores the control word.
<code>fpstsw</code>	Stores the status word.
<code>fpclcb</code>	Clears the error byte.
<code>fpssave</code>	Saves the state of FPAL96.
<code>fprestore</code>	Restores the state of FPAL96.
<code>fpseteh</code>	Sets the exception handler.

## 6.3 LOAD OPERATIONS

Load functions load various operand types (common constants, floating point numbers, integers, and decimals) into the floating-point accumulator (FPACC). FPAL96 converts non-floating point operands into floating point operands before performing these load operations.

Loading operations include:

<code>fpldz</code>	Loads the FPACC with +0.0.
<code>fpld1</code>	Loads the FPACC with +1.0.

<code>fpld</code>	Loads the FPACC with a floating point operand.
<code>fpldint</code>	Converts an integer operand to floating point and loads it into the FPACC.
<code>fplduint</code>	Converts an unsigned integer operand to floating point and loads it into the FPACC.
<code>fplddec</code>	Converts a decimal operand to floating point and loads it into the FPACC.

## 6.4 STORE OPERATIONS

Store functions first convert the value of the FPACC into the desired format then store the converted value into the destination operand.

Store operations include:

<code>fpst</code>	Stores the value in the FPACC into a floating point variable.
<code>fpstint</code>	Stores the value in the FPACC into an integer variable after converting it from floating point format to integer format.
<code>fpstuint</code>	Stores the value in the FPACC into an unsigned integer variable after converting it from floating point format to integer format.
<code>fpstdec</code>	Stores the value in the FPACC into a decimal variable after converting it from floating point format to decimal format.

## 6.5 UNARY OPERATIONS

Unary functions operate on the FPACC value, then place the result in the FPACC. This group consists of the following functions:

<code>fpneg</code>	Sets the sign of the FPACC value to negative.
<code>fpabs</code>	Sets the sign of the FPACC value to positive.
<code>fpsqrt</code>	Takes the square root of the number in the FPACC.
<code>fprndint</code>	Rounds the FPACC value to an integer value.

6.6 BINARY OPERATIONS

Binary functions accept an operand in floating point format, then perform the specified operation on that operand and the FPACC accumulator, as shown in the following process:

FPACC ... FPACC *operation* *y*

Where:

*operation* is the binary function to be performed.


*y* is the operand in floating point format.

The *operation* function operates on the initial FPACC value and the value indicated by *y*, then places the result in the FPACC accumulator, as indicated by the left arrow (...).

For example, `fpadd(RealOpr)` adds the FPACC value and the value of `RealOpr`. The result is then placed in the FPACC. The `fpcomps` and `fpcompq` functions, however, do not follow this pattern.

Binary functions include:

- `fpadd` Adds a floating point number to the number in the FPACC.
- `fpsub` Subtracts a floating point number from the number in the FPACC.
- `fpmul` Multiplies a floating point number by the number in the FPACC.
- `fpdiv` Divides the number in the FPACC by a floating point operand.
- `fprem` Finds the remainder of a division operation.
- `fpcomps` Compares a floating point number with the value in the FPACC for numerical order.
- `fpcompq` Does the same comparison as `fpcomps`, but does not raise an invalid operation exception even if one of the operands is a NaN.

 See Section 5.1.1.1 for the results of operations that use zeroes and infinities.

**6.7 FUNCTIONS LIST**

The following pages explain each function in detail. The functions appear in alphabetical order.

# fpabs

## Function

Sets the sign of the FPACC value to positive.

## Syntax

fpabs

## Description

Use `fpabs` to set the sign of the FPACC value to positive. This operation generates an invalid-operation exception if the FPACC is a signalling NaN. See Chapter 7 for more information on exceptions.

## Example

The following examples show how to call the `fpabs` function in ASM196, and C196. The examples contain the minimum lines of code needed to obtain a successful translation.

1. The following example illustrates the code needed for ASM196:

```
test module main

rseg
    realopr: dsr 1
extrn PLMREG
sp equ 18h

cseg at 2080H
extrn fpinit, fpabs, fpld, fpst

start: ld sp, #stack
       call fpinit           ; Initialize FPAL96.
       push #0BFC0H         ; Push -1.5 onto the stack.
       push #0
       call fpld            ; Load FPACC with -1.5
       call fpabs           ; Convert to positive.
       call fpst            ; Load PLMREG with FPACC.
       st PLMREG+2, realopr+2 ; Load realopr with
                               ; PLMREG
       st PLMREG,realopr     ; value.
end
```



2. The following example illustrates the code needed for C196:

```
#include <fpal96.h>      /* Include FPAL96 and */
#include <math.h> file.  /* math.h header files. */

float realopr, rv;

main()
{
    rv = -1.5;
    realopr = fabs(rv); /* fabs is a run-time
                        library function
                        declared in the
                        math.h header file.
                        */
}
```



invalid-operation exception  
Not-a-Number

# fpadd

## Function

Adds a floating point number to the FPACC value.

## Syntax

`fpadd(real_var)`

where:

*real\_var* is a variable in floating point format.

## Description

Use `fpadd` to add the value of the floating point operand to the FPACC value. FPAL96 places the sum back in the FPACC. Table 6-1 shows the FPACC content when adding zero or infinity numbers.

Operation	Result
+0 plus +0	+0
−0 plus −0	−0
+0 plus −0, −0 plus +0	see the notes
+X plus −X, −X plus +X	see the notes
#0 plus #X, #X plus #0	#X
+infinity plus +infinity	+infinity
−infinity plus −infinity	−infinity
+infinity plus −infinity	invalid operation
−infinity plus +infinity	invalid operation
#infinity plus #Z	\$infinity
#Z plus #infinity	\$infinity
<b>NOTES:</b> The sign of zero is determined by the rounding mode as follows: + for nearest, up or truncate − for down X and Y denote any nonzero operands. # denotes either sign (+ or −).	

Table 6-1: Addition with zero and infinity operands

The `fpadd` function can generate an invalid operation, denormal, overflow, underflow, or precision exception. See Chapter 7 for more information on these exceptions.

### Example

The following example shows how to call the `fpadd` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

The following example illustrates the code needed for ASM196:

```

        test module main
        rseg
        realopr: dsr 1           ; Define floating point
                                ; operands.

        result: dsr 1

        extrn PLMREG
        sp equ 18h

        cseg at 2080H
        extrn fpinit, fpadd, fpst, fpld

start:  ld sp, #stack           ; Load stack pointer.
        call fpinit             ; Initialize FPAL96.
ldacc:  push #404DH             ; Load 3.21 value to FPACC.
        push #70A4H
        call fpld
ldopr:  ld realopr+2, #3FC0H    ; Load realopr with 1.5
                                ; to realopr.
        ld realopr, #0
        push realopr+2         ; Push the high-order
                                ; part
        push realopr           ; Push low-order part of
                                ; operand onto the
                                ; stack.
        call fpadd             ; Add 1.5 to 3.21.
sum:    call fpst
        st PLMREG+2, result+2  ; Put sum into result.
        st PLMREG,result
        end

```



You need not call `fpadd` explicitly in C196. The compiler calls the function implicitly when performing the operations shown below. Make sure `fpinit` is called before performing any floating-point operation.

```
#include <fpal96.h>
float a,b;

a = 1.1;
b = a + 2.5;
```



denormal exception  
infinity  
invalid-operation exception  
overflow exception

precision exception  
underflow exception  
zero

# fpcleb

## Function

Clears the error byte

## Syntax

```
fpcleb(byte_val)
```

where:

*byte\_val* is a byte value.

## Description

Use `fpcleb` to clear the error byte after an exception occurs. This function sets the error byte to zero if the value of the byte variable is greater than 7. Otherwise, only the bit designated by the variable in the error byte is cleared. For example, if you include the line, `fpcleb(0)`, FPAL96 only clears bit 0 of the error byte.

## Examples

The following examples show how to call the `fpcleb` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

1. The following example illustrates the code needed for ASM196:

```
test module main
rseg
clrvar: dsw 1                ; Defined as a word to
                             ; guarantee word
                             ; alignment.

cseg at 2080H
extrn fpinit, fpcleb        ; Declare fpinit and
                             ; fpcleb as externals.

br start
excpt_hndlr:
    ldbze clrvar, #08H      ; Load the value of
                             ; eight
                             ; in bytevar and sign
                             ; extend.This clears all
                             ; the bits in the error
                             ; byte.
```

```

        push clrvar                ; Push bytevar onto the
                                   ; stack.
        call fpcleb                ; Call fpcleb.
        ret
start:
        call fpinit                ; Initialize FPAL96
        push #excpt_hndlr          ; Push exception handler
                                   ; address onto the
                                   ; stack.
        call fpseteh               ; Set exception handler.
        end

```

2. The following example illustrates the code needed for C196:

```

#include <fpal96.h>                /* Include FPAL96 header
                                   file. */

#define CLRVAR 8

void alien_err_hndlr(Info *info, Result *result)
                                   /* Info and Result are */
                                   /* defined in fpal96.h. */
{
    fpcleb(CLRVAR);                /* Clears the error byte. */
}

main()
{
    fpseteh(err_hndlr); /* Set the exception
                          handler. */
}

```



fpseteh  
status word

# fpcomps / fpcompq

## Function

Compare FPACC with a floating point number

## Syntax

```
fpcomps(real_var) | fpcompq(real_var)
```

Where:

*real\_var* is a variable in floating point format.

## Description

Use `fpcomps` or `fpcompq` to compare the number in the FPACC with a floating point number for numerical order. The functions set the C1 and C0 comparison bits of the status word depending on the result, as shown in Table 6-2.

Order	C1	C0
FPACC > <i>realopr</i>	0	1
FPACC < <i>realopr</i>	1	0
FPACC = <i>realopr</i>	0	0
unordered	1	1

Table 6-2: Status word (C1 and C0) settings

The unordered case occurs when at least one of the operands is Not-a-Number (NaN).

The `fpcompq` and `fpcomps` functions can generate a denormal exception. However, only the `fpcomps` function generates an invalid-operation exception if one of its operands is a NaN. See Chapter 7 for more information on these exceptions.

## Example

The following example shows how to call the `fpcomps` and `fpcompq` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

The following example illustrates the code needed for ASM196:

```

test module main
rseg
    realopr: dsr 1
sp equ 18H

cseg at 2080H
extrn fpinit, fpld          ; Declare as
extrn fpcomps, fpcompq      ; externals.
ld sp, #stack               ; Load stack pointer.
call fpinit                 ; Initialize FPAL96.
push #404DH                 ; Load 3.21 value to
push #70A4H                 ; FPACC.
call fpld
ld realopr+2, #3FC0H        ; Load realopr with
ld realopr, #0              ; 1.5.
push realopr+2              ; Push high-order part
                             ; of operand onto the
                             ; stack.
push realopr                ; Push low-order part
                             ; of operand onto the
                             ; stack.
call fpcomps                ; Compare realopr with
                             ; FPACC (signalling
                             ; comparison)
call fpcompq                ; Compare realopr with
                             ; FPACC
                             ; (quiet comparison).
end;

```



You need not call `fpcomps` and `fpcompq` explicitly in C196. The compilers call the functions implicitly when performing the operation shown below. Make sure `fpinit` is called before performing any floating-point operation.

```

float a,b;

if (a > b)
    printf(" a is greater than b");

```



denormal exception  
Not-a-Number  
status word



# fpdiv

## Function

Divides the FPACC value by a floating point number.

## Syntax

```
fpdiv(real_var)
```

where:

*real\_var* is a variable in floating point format.

## Description

Use `fpdiv` to divide the FPACC value by a floating point operand. FPAL96 places the result back in the FPACC. Table 6-3 shows the FPACC content when dividing zero or infinity numbers.

Operation	Result
#0/#0	invalid operation
#X/#0	zero divide
+0/+X, -0/-X	+0
+0/-X, -0/+X	-0
-X/-Y, +X/+Y	+0 (when underflow)
-X/+Y, +X/-Y	-0 (when underflow)
#infinity / #infinity	invalid operation
#infinity / #Z	⊕infinity
#Z / #infinity	⊕0
NOTES: X and Y denote any nonzero operands. # denotes either sign (+ or -). Z denotes a finite operand. ⊕ denotes the exclusive OR of the original operand signs.	

Table 6-3: Division with zero and infinity operands

The `fpdiv` function can generate an overflow, zero divide, underflow, precision, invalid operation, or denormal exception. Table 6-6-3 shows some cases when you can get the invalid operation and the zero-divide exceptions. See Chapter 7 for more information on these exceptions.

### Example

The following example shows how to call the `fpdiv` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

The following example illustrates the code needed for ASM196:

```

test module main
rseg
    realopr: dsr 1          ; Define floating point
                           ; operands.
    result: dsr 1

extrn PLMREG
sp equ 18h

cseg at 2080H
extrn fpinit, fpdiv, fpst, fpld

start: ld sp, #stack      ; Load stack pointer.
      call fpinit         ; Initialize FPAL96.
ldacc: push #40C7H        ; Load 6.23 value to FPACC.
      push #5C29H
      call fpld
ldopr: ld realopr+2, #404DH ; Load realopr with 3.21
      ld realopr, #70A4H   ; to realopr.
      push realopr+2       ; Push realopr value onto
      push realopr         ; the stack.
      call fpdiv           ; Divide FPACC by realopr.
      call fpst
      st PLMREG+2, result+2 ; Put quotient into result
      st PLMREG, result
end

```



You need not call `fpdiv` explicitly in C196. The compilers call the function implicitly when performing the operations shown below. Make sure `fpinit` is called before performing any floating-point operation.

```
float a,b;
```

```
a = 6.25;
```

```
b = a / 2.5;
```



denormal exception  
infinity  
invalid-operation exception  
overflow exception

precision exception  
underflow exception  
zero  
zero-divide

# fpinit

## Function

Initializes the FPAL96 control variables with default values.

## Syntax

```
fpinit
```

## Description

Use `fpinit` to initialize the FPAL96 environment with default values. This function performs the following tasks:

- Sets the control word to its default value of 003FH, that is, round to nearest and all exceptions masked.
- Sets the status word to 3000H which indicates that the FPACC is a signalling NaN and the error byte is zero. The fraction part of the signalling NaN stored in the FPACC is 000001H, that is, 1 in the least significant bit of the fraction.
- Attaches the default exception handler.

## Example

The following example shows how to call the `fpinit` function in ASM196. The example contains the minimum lines of code for a successful translation. For C196, with OMF V3.2 you only have to link `cstart.obj` with your modules and libraries. With older OMF versions the function `fpinit` is usually called in the function `_main()`. See the file `_main.c` for details.

The following example illustrates the code needed for ASM196:

```
test module main
sp equ 18H

cseg at 2080H
extrn fpinit

ld sp, #stack      ; Load stack pointer.
call fpinit        ; Initialize FPAL96.
end
```

# fpld

## Function

Loads a floating point value into the FPACC.

## Syntax

```
fpld(real_var)
```

where:

*real\_var* is a variable in floating point format.

## Description

Use `fpld` to load a single-precision floating point number into the FPACC. This function can generate an invalid operation or denormal exception. See Chapter 7 for more information on these exceptions.

## Example

The following example shows how to call the `fpld` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

The following example illustrates the code needed for ASM196:

```
test module main
rseg
realopr: dsr 1

cseg at 2080H
extrn fpinit, fpld

call fpinit           ; Initialize FPAL96.
ld realopr+2, #3FC0H  ; Load the 1.5 to realopr.
ld realopr, #0
push realopr+2        ; Push high-order part of
                     ; operand onto the stack.
push realopr          ; Push low-order part of
                     ; operand onto the stack.
call fpld             ; Loads realopr value to
                     ; FPACC.
end
```



The compiler will notice any explicit floating point operation on a variable and will call `fp1d` implicitly. See the example below. Make sure `fpinit` is called before performing any floating-point operation.

```
float a, b;
```

```
a = 1.5;      /* Initialize a and call fp1d */  
b = 2 * a;
```



denormal exception  
invalid-operation exception

# fpldcw

## Function

Sets the control word to the specified value.

## Syntax

```
fpldcw(word_val)
```

where:

*word\_val* is a word variable.

## Discussion

Use `fpldcw` to change the default setting of the control word after calling `fpinit`. FPAL96 loads the value, specified by *word\_val*, into the control word. The library takes no additional action even if this function unmask a previously masked exception.

## Examples

The following examples show how to call the `fpldcw` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

1. The following example illustrates the code needed for ASM196:

```
test module main
rseg
wordvar: dsw 1

cseg at 2080H
extrn fpinit, fpldcw

call fpinit           ; Initialize FPAL96.
ld wordvar, #083FH    ; Selects round-up mode.
push wordvar          ; Push the value onto
                      ; the stack.

call fpldcw
end
```

2. The following example illustrates the code needed for C196:

```
#include <fpal96.h>    /* Include the FPAL96 header
                        file. */

unsigned int wordvar;

main()
{
    wordvar = 0x083F; /* Selects round-up mode */
    fpldcw(wordvar);
}
```



control word  
fpinit



# fplddec

## Function

Converts a decimal to floating point and places the result in the FPACC.

## Syntax

```
fplddec(shortopr,decopr)
```

where:

*shortopr* is a 16-bit signed variable.

*decopr* is a structure containing the mantissa and exponent of a decimal number.

## Description

Use `fplddec` to convert a decimal value into floating point format. FPAL96 places the result back in the FPACC.

The value of the converted decimal number is:

$$\text{FLOAT} = \text{decopr.mantissa} * 10^{\text{decopr.exponent} - \text{shortopr}}$$

Here the period (.) is used as the C196 membership operator and not as a decimal point. See Chapter 5 for further details on the representation of decimal operands in FPAL96.

This function can generate a precision, overflow, or underflow exception. See Chapter 7 for more information on these exceptions.

## Examples

The following examples show how to call the `fplddec` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

1. The following example illustrates the code needed for

```
test module main

extrn PLMREG
rseg
    realopr:  dsr 1
    shortopr: dsw 1           ; The number of digits to the
                                ; right of decimal point.

    decopr:   dsl 1           ; Mantissa.
    exp:      dsb 1           ; Exponent.
sp equ 18H

cseg at 2080H
extrn fpinit, fpldddec
extrn fpst

ld sp, #stack           ; Load stack pointer.
call fpinit             ; Initialize FPAL96.
ld shortopr, #3         ; 3 digits after the decimal
                        ; point.
ld decopr, #4D2H        ; Load decopr with 1234.
ld decopr+2, #0
ldb exp, #0              ; Exponent = 0.
push shortopr            ; Make sure shortopr is
                        ; word-aligned.
push #decopr             ; Push address of decopr onto
                        ; stack.
call fpldddec            ; Convert decimal to floating
                        ; point.

call fpst
st PLMREG+2, realopr+2
st PLMREG, realopr       ; Load floating point number
                        ; to realopr.

end
```

2. The following example illustrates the code needed for C196:

```
#include <fpal96.h>      /* Include header file. */
short shortopr;          /* Number of digits to the
                          /* right of the decimal pt.*/
DecimalType decopr;      /* DecimalType structure is
                          /* defined in fpal96.h */

float result;

main()
{
    shortopr = 3;         /* 3 digits after the decimal
                          /* point. */

    decopr.mantissa = 1234;
    decopr.exponent = 0;
    fplddec(shortopr, &decopr);
    result = fpst();      /* result = 1.234 */
}
```



decimal floating-point number  
 overflow exception  
 precision exception  
 underflow exception

# fpldint / fplduint

## Function

Converts a long integer or a long unsigned integer to floating point and places the result in the FPACC.

## Syntax

`fpldint(long_var) | fplduint(ulong_var)`

where:

*long\_var* is a signed 32-bit variable.

*ulong\_var* is an unsigned 32-bit variable.

## Description

Use `fpldint` to convert a long integer operand into floating point format. Use `fplduint` to convert a long unsigned integer operand into floating point format. FPAL96 places the result back in the FPACC. This function can generate a precision exception when the operation sets the overflow or underflow exception bit. See Chapter 7 for more information on this exception.

## Examples

The following examples show how to call the `fpldint` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

1. The following example illustrates the code needed for ASM196:

```
test module main
extrn PLMREG
rseg
    longopr:  dsl 1
    realopr:  dsr 1
sp equ 18H

cseg at 2080H
extrn fpinit, fpldint, fpst
```

```

ld sp, #stack           ; Load stack pointer.
call fpinit             ; Initialize FPAL96.
ld longopr+2, #0H       ; Load longopr with 47018.
ld longopr, #0B7AAH
push longopr+2          ; Push longopr onto stack.
push longopr
call fpldint            ; Convert long to floating
                        ; point

call fpst
st PLMREG+2, realopr+2   ; Load FPACC to realopr.
st PLMREG, realopr       ; realopr = 47018.0
end

```

2. The following example illustrates the code needed for C196:

```

#include <fpal96.h>      /* Include the FPAL96 header
                        file. */

long longopr;
float result;

main()
{
    longopr = 47018;
    fpldint(longopr);    /* Convert long to float. */
    result = fpst();      /* Place FPACC value to
                        result. */
}

```



overflow exception  
precision exception  
underflow exception

# fpldz / fpld1

## Function

Facilitates the use of the constants 0 and 1.

## Syntax

fpldz | fpld1

## Description

Use fpldz and fpld1 to facilitate the use of the commonly applied constants, 0 and 1. These functions load +0.0 or +1.0 to the FPACC respectively.

## Example

This example shows how to call fpldz and fpld1 in ASM196:

```
test module main
extrn PLMREG
sp equ 18H
rseg
    a: dsr 1
    b: dsr 1

cseg at 2080H
extrn fpinit, fpldz, fpld1, fpst
ld sp, #stack           ; Load stack pointer.
call fpinit             ; Initialize FPAL96.
call fpld1              ; Load FPACC with 1.
call fpst
ld a+2H,PLMREG+2        ; Load a with 1.0
ld a,PLMREG
call fpldz              ; Load FPACC with 0.0
call fpst
ld b+2H,PLMREG+2        ; Load b with 0.0
ld b,PLMREG
end
```



The compiler will notice any explicit floating point operation on a variable and will call `fpldz` and `fpldl` implicitly. See the example below. Make sure `fpinit` is called before performing any floating-point operation.

```
float a, b, c, d;  
  
a = 1.0;      /* Initialize a and call fpldl */  
b = a + 2.5;  
  
c = 0.0;      /* Initialize c and call fpldz */  
d = c + 1.5;
```

# fpmul

## Function

Multiplies a floating point number by the FPACC value.

## Syntax

```
fpmul(real_var)
```

where:

*real\_var* is a variable in floating point format.

## Description

Use `fpmul` to multiply a floating point number by the FPACC value. FPAL96 places the result in the FPACC. Table 6-4 shows the FPACC content when multiplying zero or infinity numbers.

Operation	Result
+0 * +0, -0 * -0	+0
+0 * -0, -0 * +0	-0
+0 * +X, +X * +0	+0
+0 * -X, -X * +0	-0
-0 * +X, +X * -0	-0
-0 * -X, -X * -0	+0
+X * +Y, -X * -Y	+0 (when underflow)
+X * -Y, -X * +Y	-0 (when underflow)
#infinity * #infinity	⊕infinity
#infinity * #Z	⊕infinity
#Z * #infinity	⊕infinity
#0 * #infinity	invalid operation
#infinity * #0	invalid operation
NOTES: X and Y denote any nonzero operands. Z denotes a finite operand. # denotes either sign ( + or -). ⊕ denotes the exclusive OR of the original operand signs.	

Table 6-4: Multiplication with Zero or Infinity Operands



The `fpmul` function can generate an invalid operation, denormal, overflow, underflow, or precision exception. Table 6-6-4 shows some cases when an invalid-operation exception is generated. See Chapter 7 for more information on these exceptions.

### Example

The following example shows how to call the `fpmul` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

The following example illustrates the code needed for ASM196:

```
test module main
rseg
    realopr: dsr 1          ; Define floating point
                ; operands.
    result: dsr 1
    extrn PLMREG
    sp equ 18h

    cseg at 2080H
    extrn fpinit, fpmul, fpst, fpld

start: ld sp, #stack      ; Load stack pointer.
      call fpinit        ; Initialize FPAL96.
ldacc: push #40C7H       ; Load 6.23 to FPACC.
      push #5C29H
      call fpld
      ld realopr+2, #404DH ; Load realopr with 3.21
      ld realopr, #70A4H
      push realopr+2      ; Push realopr value onto
      push realopr        ; the stack.
      call fpmul          ; Multiply FPACC by realopr.
      call fpst
      st PLMREG+2, result+2 ; Put product into result.
      st PLMREG,result
end
```



You need not call `fpmul` explicitly in C196. The compilers call the function implicitly when performing the operations shown below. Make sure `fpinit` is called before performing any floating-point operation.

```
float a,b;
```

```
a = 6.25;
```

```
b = a * 2.5;
```



denormal exception  
infinity  
invalid-operation exception  
overflow exception

precision exception  
underflow exception  
zero

# fpneg

## Function

Changes the FPACC value to a negative number.

## Syntax

fpneg

## Description

Use fpneg to change the FPACC value to a negative number. This function can generate an invalid-operation exception when the FPACC is a signalling NaN. See Chapter 7 for more information on this exception.

## Example

This example shows how to use fpneg in ASM196:

```
test module main
rseg
    realopr: dsr 1

extrn PLMREG
sp equ 18h

cseg at 2080H
extrn fpinit, fpneg, fpld, fpst

ld sp, #stack
call fpinit           ; Initialize FPAL96.
push #3FC0H           ; Push -1.5 onto the
                        ; stack.

push #0
call fpld             ; Load FPACC with 1.5
call fpneg            ; Convert to negative.
call fpst             ; Load PLMREG with
                        ; FPACC.

st PLMREG+2, realopr+2 ; Load realopr with
st PLMREG, realopr     ; PLMREG value.
end
```



You need not explicitly call `fpneg` when coding in C196. The compiler implicitly calls the function when performing the following operation:

```
float a, b;
```

```
a = -b;      /* a equals negative b */
```



invalid-operation exception

# fprem

## Function

Computes the remainder of FPACC divided by a floating point number.

## Syntax

```
fprem(real_var)
```

where:

*real\_var* is a variable in floating point format.

## Description

Use `fprem` to compute the remainder of FPACC value divided by a floating point operand. The remainder is defined as  $FPACC - Q * realopr$ , where  $Q$  is the integer nearest to the exact value of  $FPACC / realopr$ . The remainder is always greater or equal to  $-realopr / 2$ , and less than or equal to  $+realopr / 2$ . FPAL96 places the remainder back in the FPACC. Table 6-5 shows the FPACC content when operating with zero and infinity numbers.

Operation	Result
#0 REM #0	invalid operation
#X REM #0	invalid operation
+0 REM #X	+0
-0 REM #X	-0
#infinity REM #infinity	invalid operation
#infinity REM #Z	invalid operation
#Z REM #infinity	#X
NOTES: X denotes any nonzero operands. # denotes either sign (+ or -). Z denotes a finite operand.	

Table 6-5: Remainder With Zero and Infinity Operands

The `fprem` function can generate an invalid operation, denormal, or underflow exception. Table 6-6-5 shows some cases when an invalid-operation exception is generated. See Chapter 7 for more information on these exceptions.

## Examples

The following examples show how to call the `fprem` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

1. The following example illustrates the code needed for ASM196:

```

test module main
rseg
    realopr: dsr 1      ; Define floating point
                      ; operands.
    rem: dsr 1

extrn PLMREG
sp equ 18h

cseg at 2080H
extrn fpinit, fprem, fpst, fpld

start: ld sp, #stack      ; Load stack pointer.
      call fpinit        ; Initialize FPAL96.
ldacc: push #40C7H        ; Load 6.23 value to FPACC
      push #5C29H
      call fpld
ldopr: ld realopr+2, #404DH ; Load realopr with 3.21
      ld realopr, #70A4H   ; to realopr.
      push realopr+2      ; Push realopr value onto
      push realopr        ; the stack.
      call fprem          ; Calculate remainder of
                      ; FPACC/realopr.

      call fpst
      st PLMREG+2, result+2 ; result = remainder
      st PLMREG, result
end

```

2. The following example illustrates the code needed for C196:

```
#include <fpal96.h>          /* Include FPAL96 header
                               file. */

float realopr;

main()
{
    realopr = 6.23;
    fpld(realopr);
    fprem(3.21);              /* Calculate remainder. */
    result = fpst();          /* result = remainder. */
}
```



denormal exception  
infinity  
invalid-operation exception

precision exception  
underflow exception  
zero

# fprndint

## Function

Round the FPACC value to the nearest integer.

## Syntax

`fprndint`

## Description

Use `fprndint` to round the number stored in the FPACC to the nearest integer value. This function can generate an invalid operation or precision exception. See Chapter 7 for more information on these exceptions.

## Examples

The following examples show how to call the `fprndint` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

1. The following example illustrates the code needed for ASM196:

```
test module main
extrn PLMREG
rseg
    result:  ds1 1
    realopr:  dsr 1
    sp equ 18H

cseg at 2080H
extrn fpinit, fprndint, fpstint, fpld
ld sp, #stack           ; Load stack pointer.
call fpinit             ; Initialize FPAL96.
ld realopr+2, #404DH    ; Load realopr with 3.21456
ld realopr, #0BB5AH
push realopr+2
push realopr
call fpld               ; Load FPACC with realopr.
call fprndint           ; Round to nearest integer.
call fpstint
st PLMREG+2, result+2   ; Load integer to result.
st PLMREG, result
end
```



2. The following example illustrates the code needed for C196:

```
#include <fpal96.h>          /* Include FPAL96 header
                               file. */

int a;

main()
{
    fpld(3.21456);           /* Load FPACC with 3.21456 */
    fprndint();
    a = fpstint();            /* a now contains 3. */
}
```



invalid-operation exception

# fprstor

## Function

Restores the previous values of the FPAL96 variables.

## Syntax

```
fprstor(struc)
```

where:

*struc* is a structure containing the values of the status word, the address of the exception handler, the control word, and local data.

## Description

Use `fprstor` to restore FPAL96 to its previous state by loading the value of the storage structure (*struc*), previously saved using `fpsave`. FPAL96 takes no additional action even if a previously masked exception becomes unmasked when this function is invoked. You can use this function in the epilog of interrupt procedures with `fpsave` to allow reentrancy within FPAL96.

## Examples

The following examples show how to call the `fprstor` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

1. The following example illustrates the code needed for ASM196:

```
test module main
sp equ 18H
dseg
savearea:                ; Savearea structure
    status_word:    dsw 1
    err_handler:    dsp 1
    control_word:   dsw 1
    local_data:     dsb 60
```

```

cseg at 2080H
extrn fpinit, fprstor, fpsave
ld sp, #stack
call fpinit           ; Initialize FPAL96.
push #savearea        ; Push the address of
                      ; savearea onto the stack.
call fpsave           ; Save the FPAL96 status.
.
.
push #savearea
call fprstor          ; Restore the previous
                      ; status.
end

```

2. The following example illustrates the code needed for C196:

```

#include <fpal96.h>      /* Include FPAL96 header
                        file. */
SaveArea savearea;      /* SaveArea is predefined
                        in fpal96.h */

main()
{
    fpsave(&savearea)    /* Save the FPAL96 status. */
    .
    .
    .
    fprstor(&savearea);  /* Restore the previous
                        status. */
}

```



Creating your own exception handler

fpsave

Program status word

# fpsave

## Function

Saves the complete state of FPAL96

## Syntax

```
fpsave(struc)
```

where:

*struc* is a structure containing the value of the status word, as well as the address of the exception handler, the control word, and local data.

## Description

Use `fpsave` to save the complete state of FPAL96 (status word, exception handler address, control word, and local data including the FPACC and the program status word) to the `savearea` and initialize the state of FPAL96 as if `fpinit` is executed. You can use this function in the prolog of interrupt functions to allow reentrancy in FPAL96 or to save data for diagnostic purposes.



This function calls the `pushf` instruction to save the program status word (PSW) onto the stack. The `pushf` instruction also clears the interrupt masks. Make sure you reload the interrupt masks after calling `fpsave` if you want interrupts to be enabled.

## Examples

The following examples show how to call the `fpsave` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

1. The following example illustrates the code needed for ASM196:

```
test module main
sp equ 18H
dseg
savearea:                ; Savearea structure
    status_word:    dsw 1
    err_handler:    dsp 1
    control_word:    dsw 1
    local_data:     dsb 60
```

```

cseg at 2080H
extrn fpinit, fpsave
ld sp, #stack           ; Load stack pointer.
call fpinit             ; Initialize FPAL96.
push #savearea          ; Save the address of
                        ; savearea onto the stack.
call fpsave             ; Save FPAL96 status. */
end

```

2. The following example illustrates the code needed for C196

```

#include <fpal96.h>      /* Include FPAL96 header
                        file. */

SaveArea savearea;      /* SaveArea is defined in
                        fpal96.h */

main()
{
    fpsave(&savearea);  /* Save FPAL96 status. */
}

```



Creating your own exception handler

```

fprstor
Program status word
pushf

```

# fpseteh

## Function

Sets the exception handler

## Syntax

```
fpseteh(handler_name)
```

where:

*handler\_name* is the name of your exception handler routine.

## Description

Use `fpseteh` to tell FPAL96 to invoke your exception handler upon detection of any unmasked exceptions. This function overrides the default exception handler FPAL96 attaches after initialization (`fpinit`). See Section 7.8 for details on how to build your own exception handler.

## Examples

The following examples show how to call the `fpseteh` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

1. The following example illustrates the code needed for ASM196:

```
test module main
sp equ 18H

cseg at 2080H
extrn fpinit, fpseteh
ld sp, #stack
br START
errhdlr: .                ; Define error handler
.                ; procedure.
.
.
ret
```

```

start: call fpinit          ; Initialize FPAL96.
      push #errhdlr        ; Push the exception handler
                                ; address onto the stack.
      call fpseteh
      end

```

2. The following example illustrates the code needed for C196:

```

#include <fpal96.h>          /* Include header file.      */
void alien_err_hdlr (Info *info, Result *result)
/* Info and Result are      */
/* predefined in fpal96.h,  */
{                             /* Define the Handler    */
}                             /* routine.         */

main()
{
    fpseteh(err_hdlr);      /* Set handler.      */
}

```



control word  
exception handler  
fpldcw

# fpsqrt

## Function

Sets the FPACC value to its square root.

## Syntax

`fpsqrt`

## Description

Use `fpsqrt` to set the FPACC value to its square root. Table 6-6-6 shows the FPACC content when operating with zero or infinity numbers.

Operation	Result
-0	-0
+0	+0
+infinity	+infinity

Table 6-6: Square root with zero and infinity operands

The `fpsqrt` function can generate an invalid operation, denormal, or precision exception. See Chapter 7 for more information on these exceptions.

## Examples

The following examples show how to call the `fpsqrt` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

1. The following example illustrates the code needed for ASM196:

```
test module main
extrn PLMREG
sp equ 18H

rseg
  b:  dsr 1

cseg at 2080H
extrn fpinit, fpsqrt, fpst, fpld
```



```

ld sp, #stack           ; Load stack pointer.
call fpinit             ; Initialize FPAL96.
push #4101H             ; Push 8.1234 onto the stack.
push #0F972H
call fpld               ; Load FPACC with 8.1234
call fpsqrt             ; Calculate the square root.
call fpst
ld b+2H, PLMREG+2       ; Load b with square root.
ld b, PLMREG
end

```

2. The following example illustrates the code needed for C196:

```

#include <fpal96.h>      /* Load header file. */
float result;

main()
{
    fpld(8.1234);
    fpsqrt();
    result = fpst();     /* result = square root of
                          8.1234 */
}

```



denormal exception  
infinity  
invalid-operation exception

precision exception  
zero

# fpst

## Function

Returns the FPACC value in floating point format.

## Syntax

```
real_var = fpst
```

where:

*real\_var* is a variable in floating point format.

## Description

Use `fpst` to return the FPACC value in floating point format. This function generates an invalid operation if the operand is a signalling NaN. See Chapter 7 for more details on these exceptions.

## Examples

The following examples show how to call the `fpst` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

1. The following example illustrates the code needed for ASM196:

```
test module main
extrn PLMREG
sp equ 18H

rseg
realvar: dsr 1

cseg at 2080H
extrn fpinit, fpst
ld sp,#stack           ; Load stack pointer.
call fpinit            ; Initialize FPAL96.
call fpst              ; Load FPACC value in PLMREG.
st PLMREG+2, realvar+2 ; Store high-order part of
                       ; the result to realvar.
st PLMREG,realvar      ; Store low-order part of
                       ; result to realvar.
end
```

2. The following example illustrates the code needed for C196:

```
#include <fpal96.h>      /* Include FPAL96 header
                           file. */

float realvar;

main()
{
    realvar = fpst();      /* Store FPACC value in
                           realvar. */
}
```



invalid-operation exception  
signalling NaN

# fpstcw

## Function

Stores the content of the control word into a word variable.

## Syntax

```
word_var = fpstcw
```

where:

*word\_var* is a unsigned 16-bit variable.

## Description

Use `fpstcw` to store the content of the control word into the word variable, indicated by *word\_var*.

## Examples

The following examples show how to call the `fpstcw` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

1. The following example illustrates the code needed for ASM196:

```
test module main
extrn PLMREG
sp equ 18H

dseg
    wordvar: dsw 1

cseg at 2080H
extrn fpinit, fpstcw
ld sp, #stack           ; Load stack pointer.
call fpinit             ; Initialize FPAL96.
call fpstcw             ; Load the control word
                        ; value in PLMREG.
st PLMREG, wordvar      ; Store the control word
                        ; value in wordvar.
end
```

2. The following example illustrates the code needed for C196:

```
#include<fpal96.h>          /* Include header file. */

unsigned int wordvar;

main()
{
    wordvar = fpstcw();
                                /* Store the control word
                                value in wordvar. */
}
```



control word

# fpstdec

## Function

Stores the FPACC value in a decimal floating-point number.

## Syntax

```
fpstdec(shortopr, .dec_var)
```

where:

*shortopr* is a signed 16-bit variable.

*dec\_var* is a structure containing the mantissa and exponent of a decimal number.

## Description

Use `fpstdec` to convert the value in the FPACC into decimal format. FPAL96 stores the converted value in a decimal variable.

The value of the converted decimal number is:

$$\text{FLOAT} = \text{decopr.mantissa} * 10^{(\text{decopr.exponent} - \text{shortopr} + 1)}$$

The dot (.) here acts as the C196 membership operator and not as a decimal point. See Chapter 5 for further details on the representation of decimal operands in FPAL96.

This function can generate an invalid operation or a precision exception. See Chapter 7 for more information on these exceptions.

## Examples

The following examples show how to call the `fpstdec` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

1. The following example illustrates the code needed for ASM196:

```

test module main
extrn PLMREG
rseg
    realopr: dsr 1
    shortopr: dsw 1           ; The number of digits to the
                               ; right of the decimal point.

    decopr:   dsl 1           ; Mantissa.
    exp:      dsb 1           ; Exponent.
    decopr2:  dsl 1           ; Mantissa.
    exp2:     dsb 1           ; Exponent.
sp equ 18H

cseg at 2080H
extrn fpinit, fpldddec
extrn fpst, fpstdec
ld sp, #stack                ; Load stack pointer.
call fpinit                  ; Initialize FPAL96.
ld shortopr, #3              ; 3 digits to the right of
                               ; the decimal point.

ld decopr, #4D2H             ; Load 1234 to decopr.
ld decopr+2, #0
ldb exp, #0                  ; Exponent = 0.
push shortopr                ; Make sure shortopr is
                               ; word-aligned.

push #decopr                 ; Push address of decopr onto
                               ; stack.

call fpldddec                ; Convert to decimal to
                               ; floating point.

call fpst
st PLMREG+2, realopr+2        ; Move floating point value
                               ; to realopr.
st PLMREG, realopr           ; realopr = 1.234
ld shortopr, #4              ; 4 digits to the right of
                               ; the decimal point

push shortopr
push #decopr2
call fpstdec                 ; Convert floating point
                               ; to decimal.
                               ; decopr2 = 1234

end

```

2. The following example illustrates the code needed for C196:

```
#include <fpal96.h>          /* Include header file.   */
short shortopr;
DecimalType decopr;          /* DecimalType is predefined
                              in fpal96.h.            */

main()
{
    fpstdec(shortopr,&decopr); /* Convert FPACC value */
                              /* to decimal.          */
}
```



decimal floating-point number  
invalid-operation exception  
precision exception



# fpstint and fpstuint

## Function

Stores the FPACC value in a long integer variable or in a long unsigned integer variable.

## Syntax

```
long_var = fpstint | ulong_var = fpstuint
```

where:

*long\_var* is a signed 32-bit variable.

*ulong\_var* is an unsigned signed 32-bit variable.

## Description

Use `fpstint` to store the FPACC value in a long integer format. Use `fpstuint` to store the FPACC value in a long unsigned integer format. Before the value is stored in the long integer variable, FPAL96 converts the value from the internal floating point format to the external long integer format. The conversion does not affect the FPACC.

This function can generate an invalid operation or precision operation. See Chapter 7 for more details on these exceptions.

## Examples

The following examples show how to call the `fpstint` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

1. The following example illustrates the code needed for ASM196:

```
test module main
extrn PLMREG
rseg
    result: dsl 1
    realopr: dsr 1
sp equ 18H
```

```

cseg at 2080H
extrn fpinit, fprndint, fpstint, fpld
ld sp, #stack           ; Load stack pointer.
call fpinit             ; Initialize FPAL96.
ld realopr+2, #404DH    ; Load realopr with 3.21456
ld realopr, #0BB5AH
push realopr+2
push realopr
call fpld               ; Load FPACC with realopr.
call fprndint          ; Round to nearest integer.
call fpstint
st PLMREG+2, result+2   ; Load integer to result.
st PLMREG, result
end

```

2. The following example illustrates the code needed for C196:

```

#include <fpal96.h>      /* Include FPAL96 header
                        file. */

long longvar;

main()
{
    fpld(3.21456);      /* Load FPACC with 3.21456 */
    fprndint();         /* Round to nearest int. */
    longvar = fpstint(); /* Store FPACC value in
                        longvar. */
}

```



invalid-operation exception  
precision exception

# fpstsw

## Function

Stores the content of the status word in a word variable.

## Syntax

```
word_var = fpstsw
```

where:

*word\_var* is a unsigned 16-bit variable.

## Description

Use `fpstsw` to store the content of the status word into a variable.

## Examples

The following examples show how to call the `fpstsw` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

1. The following example illustrates the code needed for ASM196:

```
test module

extrn PLMREG
sp equ 18H
dseg
wordvar: dsw 1

cseg at 2080H
extrn fpinit, fpstsw

ld sp, #stack           ; Load stack pointer.
call fpinit             ; Initialize FPAL96
call fpstsw             ; Load status word value in
                        ; PLMREG.
st PLMREG, wordvar      ; Store PLMREG value in
                        ; wordvar.

end;
```

2. The following example illustrates the code needed for C196:

```
#include <fpal96.h>          /* Include FPAL96 header
                               file. */

unsigned int wordvar;

main()
{
    wordvar = fpstsw();      /* Store status word value
                              in wordvar. */
}
```



status word

# fpsub

## Function

Subtract a floating point number from the FPACC value.

## Syntax

```
fpsub(real_var)
```

where:

*real\_var* is a variable in floating point format.

## Description

Use `fpsub` to subtract the value of a floating point number from the FPACC value. FPAL96 places the result back in the FPACC. Table 6-7 shows the FPACC content when subtracting zero or infinity numbers.

Operation	Result
+0 minus -0	+0
-0 minus +0	-0
+0 minus +0, -0 minus -0	see Note 1
+X minus +X, -X minus -X	see Note 1
#0 minus #X	!X
+infinity minus -infinity	+infinity
-infinity minus +infinity	-infinity
+infinity minus +infinity	invalid operation
-infinity minus -infinity	invalid operation
#infinity minus #Z	\$infinity
#Z minus #infinity	!infinity
NOTES: The sign of zero is determined by the rounding mode as follows: + for nearest, up or truncate - for down X denotes any nonzero operands. # denotes either sign (+ or -). ! denotes the complement of the sign of X. Z denotes a finite operand. \$ denotes the sign of the original infinity operand.	

Table 6-7: Subtraction with zero and infinity operands

The `fpsub` function can generate an invalid operation, denormal, overflow, underflow, or a precision exception. Table 6-6-7 shows some cases when an invalid-operation exception is generated. See Chapter 7 for more information on each exception.

### Example

The following example shows how to call the `fpsub` function in ASM196, and C196. The examples contain the minimum lines of code for a successful translation.

The following example illustrates the code needed for ASM196:

```

test module main
rseg
    realopr: dsr 1      ; Define floating point
                      ; operands.
    result: dsr 1

extrn PLMREG
sp equ 18h

cseg at 2080H
extrn fpinit, fpsub, fpst, fpld

start: ld sp, #stack      ; Load stack pointer.
      call fpinit        ; Initialize FPAL96.
ldacc: push #40C7H        ; Load 6.23 value to FPACC
      push #5C29H
      call fpld
ldopr: ld realopr+2, #404DH ; Load realopr with 3.21
      ld realopr, #70A4H   ; to realopr.
      push realopr+2      ; Push realopr value onto
      push realopr        ; the stack.
      call fpsub          ; Divide FPACC by realopr.
      call fpst
      st PLMREG+2, result+2 ; Move value into result.
      st PLMREG, result
end

```



You need not call `fpsub` explicitly in C196. The compilers call the function implicitly when performing the operations shown below. Make sure `fpinit` is called before performing any floating-point operation.

```
float a, b;
```

```
a = 6.25;
```

```
b = a - 2.6;
```



denormal exception  
infinity  
invalid-operation exception  
overflow exception

precision exception  
underflow exception  
zero

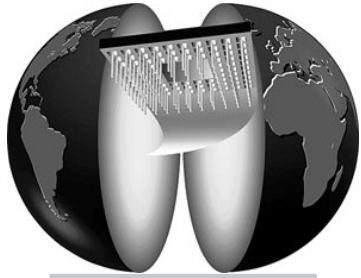
# CHAPTER

# 7

## EXCEPTIONS AND EXCEPTION HANDLING







# 7

# CHAPTER

7.1 INTRODUCTION

This chapter explains the different exceptions that FPAL96 can generate during a floating-point operation. FPAL96 recognizes six exceptions: invalid operation, zero divide, overflow, underflow, precision, and denormal. Table 7-1 gives a list of possible exceptions for each operation. The operation code is the number associated with the corresponding operation.

Operation	Operation Code	Possible Exceptions*
fpabs	8	I
fpadd	11	I, D, O, U, P
fpcompq	17	D
fpcomps	16	I, D
fpdiv	14	I, D, O, U, P, Z
fpneg	7	I
fpld	1	I, D
fplddec	3	P, O, U
fpldint	2	P
fplduint	18	P
fpmul	13	I, D, O, U, P
fprem	15	I, D, U
fprndint	10	I, P
fpsqrt	9	I, D, P
fpst	4	I
fpstdec	6	I, P
fpstint	5	I, P
fpstiunt	19	I, P
fpsub	12	I, D, O, U, P
* Abbreviations for exceptions are: I invalid operation exception. U underflow exception. D denormal exception. P precision exception. O overflow exception. Z zero divide exception.		

Table 7-1: Possible exceptions for each procedure

You can use the operation codes found in Table 7-7-1 to determine the last operation FPAL96 performed before the exception occurred. For example, the exception handler checks to see whether the operation that caused an overflow was a division operation:

```
#include <fpal96.h>

#define OVERFLOW 0x08      /* Overflow mask.          */
                          /* Unmask bit.          */
#define CLRB(var,bit_mask) var &= (~bit_mask)
                          /* Info, Result, DIV_OP
                          and OPERATION are
                          predefined in the
                          fpal96.h file.          */
void alien errhndlr( Info *info, Result *result)
{
    if (info->OPERATION == DIV_OP)
        /* Check if the operation
           is a division.          */
        printf("Division Error\n\r");
        fpcleb(3);          /* Clear OV mask.          */
}

main()
{
    float f1;
    register unsigned int temp;

    fpinit();              /* Initialize FPAL96.      */
    fpseteh(errhndlr);     /* Set exception handler.  */
    temp = fpstcw;         /* Get the control word.
                           value.          */
    CLRB(temp, OVERFLOW);  /* Enable overflow mask.  */
    fpldcw(temp);         /* Load new control word. */
    f1 = 3.3e38 / 2.0e-20; /* Causes an overflow.    */
}
```

When an exception occurs, FPAL96 sets the appropriate flag in the status word to 1, and checks the corresponding exception mask in the control word for a response. If the exception mask bit is 0, the response is unmasked, FPAL96 calls your supplied exception handler. If the mask bit is 1, FPAL96 processes the exception using the default exception handler. To direct FPAL96 to use your own handler, clear the appropriate bit in the control word and use the `fpseteh` function to assign the handler name. See Section 7.8 for instructions on how to create your own exception handler. See Chapter 5 for more details on the bits of the status word and control word.

## **7.2 INVALID-OPERATION EXCEPTION**

An invalid-operation exception occurs if an operand is invalid for the specified operation. The exception occurs in any of the following cases:

- An operand is a signalling NaN.
- Zero is multiplied by infinity.
- Infinity is divided by infinity.
- Infinities of opposite signs are added together, or infinities with the same sign are subtracted from one another; for example,  $(+\text{infinity}) - (+\text{infinity})$ .
- The square root of a negative nonzero number is attempted.
- The remainder of an infinity is divided by any number.
- Any number is divided by zero.
- A floating point number is converted to integer or decimal and the operand cannot be represented in the resulting format, for example, if the floating point number is a NaN or infinity.
- A NaN operand is used with the `fpcomps` comparison.
- FPACC is a signalling NaN.

If the invalid operation exception mask bit is unmasked, that is, the bit has a 0 value, FPAL96 calls your exception handler in each of the above cases. Otherwise, FPAL96 returns a quiet NaN as the result of the operation. For more information on NaNs, See Chapter 5.

7.3 ZERO-DIVIDE EXCEPTION

A zero-divide exception occurs when a finite nonzero number is divided by 0. When the zero-divide exception mask bit is unmasked, FPAL96 calls your exception handler. Otherwise, FPAL96 returns either +infinity or -infinity as the result. The result is positive when both operands have the same sign, negative otherwise.

7.4 OVERFLOW EXCEPTION

An overflow exception occurs when the exponent of the rounded result from an operation, assuming an unbounded exponent range, is greater than its upper limit (127).

If the overflow exception bit is unmasked, meaning it has a 0 value, FPAL96 divides the infinitely precise result, assuming an unbounded exponent, by  $2^{192}$  then rounds the quotient according to the rounding mode selected. FPAL96 passes the rounded result to your exception handler. If the operation is a conversion from decimal, the exponent of the rounded result can still be greater than 127. If so, FPAL96 sets the rounded result to a quiet NaN before passing it to your exception handler.

If the overflow exception is masked, FPAL96 returns a value determined by the sign of the result of the operation and the current rounding mode, as shown by Table 7-2.

Sign of Result	Rounding Mode	Result Returned by FPAL96
positive	nearest	+infinity
positive	down	largest finite positive number
positive	up	+infinity
positive	truncate	largest finite positive number
negative	nearest	-infinity
negative	down	-infinity
negative	up	argest finite negative number
negative	chop	largest finite negative number

Table 7-2: FPAL96 Rounding result with overflow exception masked

## 7.5 UNDERFLOW EXCEPTION

FPAL96 raises underflow exceptions under various conditions, depending on whether the underflow exception bit is masked.

If the underflow exception bit is unmasked, an underflow exception occurs when the infinitely precise result, assuming an unbounded exponent, lies in the range  $-2^{-126} < x < 2^{-126}$ , where  $x$  is not 0. When the result falls within this range, FPAL96 multiplies this number by  $2^{192}$  and then rounds the quotient according to the specified rounding mode. FPAL96 then passes the rounded result to your exception handler. If the operation is a conversion from decimal, the exponent of this rounded result can still be too low. If so, FPAL96 sets the rounded result to a quiet NaN before passing it to your exception handler.

When the exception bit is masked, FPAL96 denormalizes a result that lies in the range  $-2^{-126} < x < 2^{-126}$ , where  $x$  is not 0. The denormalization is done by shifting the fraction right while incrementing the value of the exponent until it reaches  $-126$ . FPAL96 then rounds this value according to the selected rounding mode. FPAL96 sets the underflow bit in the error byte if any bits were lost during the shift which means a loss of precision.

## 7.6 PRECISION EXCEPTION

A precision exception occurs when the rounded result is not exact or when the rounded result overflows and the overflow exception bit is masked. FPAL96 uses the rounding mode in the control word to round the result properly. For example, if the numbers 1.0000001 and 1.0000000 are rounded to the nearest thousandth, both results contain the number 1.000. However, the first number generates a precision exception, while the other does not.

If the precision exception bit is unmasked, FPAL96 passes the rounded result to your exception handler. Otherwise, FPAL96 returns the rounded result as the result of the operation.

## **7.7 DENORMALIZED-NUMBER EXCEPTION**

Denormalized-number exceptions occur when at least one of the floating-point operands is a denormal number, as specified in Chapter 2. If the denormal exception bit is unmasked, FPAL96 passes the denormalized operand to your exception handler without performing the operation. Make the necessary correction, such as normalization, if you do not want to operate on a denormal operand. FPAL96 uses the result returned by your exception handler to proceed with the operation. Otherwise, when the exception bit is masked, FPAL96 continues to use the denormalized operand in the operation. Your program is more likely to generate an overflow or underflow exception when using denormal numbers.

## **7.8 CREATING YOUR OWN EXCEPTION HANDLER**

As part of the FPAL96 initialization, you can attach your own exception handler. The handler can perform the following tasks:

- handle uninitialized variables
- execute nonstandard data by using signalling NaNs for operands
- store diagnostic information for debugging purposes
- generate your own responses to the exceptions raised

FPAL96 calls your exception handler and supplies to it the relevant information when an exception occurs. The exception handler can use information stored in the built-in variables, the status word and the control word, to continue a flagged operation or to analyze results when debugging.

The following code is an example of an exception handler written in C. The code checks for a divide-by-zero exception or an overflow then calls the exception handler.

```

#include <fpal96.h>                /* FPAL96 include file */
#define MASK 0x0C                  /* Select OV and divide by
                                   0 bits of control word*/
#define TSTBIT(var, bit_mask) ((var) &= (~(bit_mask)))
                                   /* Unmask bits OV and
                                   divide-by-0 bits */
#pragma fixedparams(errhndlr) /* Floating-point uses
                                   fixedparam convention */
#define CLR_ALL 8

                                   /* Info and Result are
                                   predefined in the
                                   fpal96.h file */

void errhndlr(Info *info, Result *result)
{
    if (info->Status_Word & 0x04)
        /* Check if divide bit */
        printf("Divide by 0 occurred!\n\r");
        /* is set */
    else
        if (info->Status_Word & 0x08)
            /* Check if OV bit is set*/
            printf("Overflow occurred!\n\r");
        fpcleb(CLR_ALL);          /* Clear error byte */
}

float f1, f2;

main()
{
    register unsigned int temp;
    init_putchar();              /* Prime TI bit */
    fpseteh(errhndlr);          /* Assigns the exception
                                   handler */
    temp = fpstcw();             /* Get the control word
                                   value */
    TSTBIT(temp, MASK);          /* Enable Overflow and
                                   Divide-by-0 exceptions*/
    fpldcw(temp);               /* Load new control word */
    f2 = 3.3e38;
    f1 = f2 / 2.0e-30;          /* Causes an overflow */
    f1 = f2 / 0;                /* Causes a divide-by-zero
                                   error */
}

```



After a normal return from the exception handler, FPAL96 uses the returned result as the actual result for the current operand or operation. If, for example, the exception occurred during an `fpadd` operation, FPAL96 converts the returned result to the internal representation for floating point values then stores the value in the FPACC. However, if you do not want to do a normal return from the exception handler, you can use a `goto` statement rather than a `return` statement when your exception handler finishes.

You must specify your exception handler using the `fpseteh` procedure described in Chapter 6. A default exception handler is attached upon initialization of FPAL96, using `fpinit`. This exception handler manages the exception by returning the preliminary result to the area pointed to by the `Result_Ptr` pointer variable.



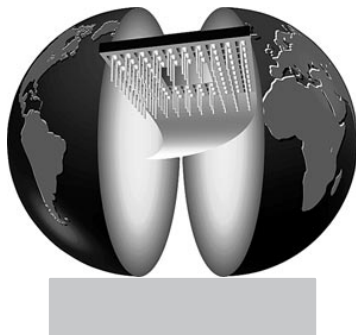
If your exception handler uses floating-point functions during its execution, you must save the current state of FPAL96, using `fpsave`, before invoking the floating-point functions from your exception handler. You must also restore the FPAL96 state, using `fprestore`, before returning to the next instruction following your exception handler. Your exception handler must be reentrant.

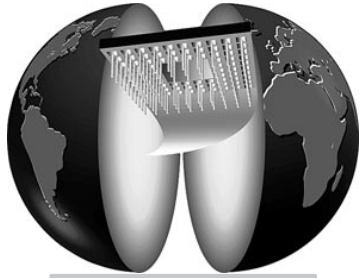
# CHAPTER

# 8

## **MK196 MAKE UTILITY**

---





# 8

# CHAPTER

This chapter describes the operation of the MK196 program. MK196 allows you to maintain, update, and reconstruct groups of programs.

## 8.1 INVOCATION SYNTAX

```
mk196  [option ...] [target ...] [macro=value ...]
mk196  -V
mk196  -? ( UNIX C-shell: "-?" or -\? )
```

## 8.2 DESCRIPTION

**mk196** takes a file of dependencies (a 'makefile') and decides what commands have to be executed to bring the files up-to-date. These commands are either executed directly from **mk196** or written to the standard output without executing them.

If no target is specified on the command line, **mk196** uses the first target defined in the first makefile.



Long filenames are supported when they are surrounded by double quotes (""). It is also allowed to use spaces in directory names and file names.

### Options

- ?            Show invocation syntax.
- D            Display the text of the makefiles as read in.
- DD          Display the text of the makefiles and 'mk196.mk'.
- G *dirname*    Change to the directory specified with *dirname* before reading a makefile. This makes it possible to build an application in another directory than the current working directory.
- K            Do not remove temporary files.
- S            Undo the effect of the -k option. Stop processing when a non-zero exit status is returned by a command.
- V            Display version information at stderr.

- W *target*** Execute as if this target has a modification time of "right now". This is the "What If" option.
- d** Display the reasons why **mk196** chooses to rebuild a target. All dependencies which are newer are displayed.
- dd** Display the dependency checks in more detail. Dependencies which are older are displayed as well as newer.
- e** Let environment variables override macro definitions from makefiles. Normally, makefile macros override environment variables. Command line macro definitions always override both environment variables and makefile macros definitions.
- f *file*** Use the specified file instead of 'makefile'. A - as the makefile argument denotes the standard input.
- i** Ignore error codes returned by commands. This is equivalent to the special target .IGNORE:.
- k** When a nonzero error status is returned by a command, abandon work on the current target, but continue with other branches that do not depend on this target.
- m *file*** Read command line information from *file*. If *file* is a '-', the information is read from standard input.
- n** Perform a dry run. Print commands, but do not execute them. Even lines beginning with an @ are printed. However, if a command line is an invocation of **mk196**, that line is always executed.
- q** Question mode. **mk196** returns a zero or non-zero status code, depending on whether or not the target file is up to date.
- r** Do not read in the default file 'mk196.mk'.
- s** Silent mode. Do not print command lines before executing them. This is equivalent to the special target .SILENT:.
- t** Touch the target files, bringing them up to date, rather than performing the rules to reconstruct them.

**-w** Redirect warnings and errors to standard output. Without, **mk196** and the commands it executes use standard error for this purpose.

*macro=value*

Macro definition. This definition remains fixed for the **mk196** invocation. It overrides any regular definitions for the specified macro within the makefiles and from the environment. It is inherited by subordinate **mk196**'s but act as an environment variable for these. That is, depending on the **-e** setting, it may be overridden by a makefile definition.

## 8.3 USAGE

### *Makefiles*

The first makefile read is 'mk196.mk', which is looked for at the following places (in this order):

- in the current working directory
- in the directory pointed to by the HOME environment variable
- in the etc directory relative to the directory where **mk196** is located

Example (PC):

when **mk196** is installed in \C196\BIN the directory \C196\ETC is searched for makefiles.

Example (UNIX):

when **mk196** is installed in /usr/local/c196/bin the directory /usr/local/c196/etc is searched for makefiles.

It typically contains predefined macros and implicit rules.

The default name of the makefile is 'makefile' in the current directory. If this file is not found on a UNIX system, the file 'Makefile' is then used as the default. Alternate makefiles can be specified using one or more **-f** options on the command line. Multiple **-f** options act as if all the makefiles were concatenated in a left-to-right order.

The makefile(s) may contain a mixture of comment lines, macro definitions, include lines, and target lines. Lines may be continued across input lines by escaping the NEWLINE with a backslash (\). If a line must end with a backslash then an empty macro should be appended. Anything after a "#" is considered to be a comment, and is stripped from the line, including spaces immediately before the "#". If the "#" is inside a quoted string, it is not treated as a comment. Completely blank lines are ignored.

An *include* line is used to include the text of another makefile. It consists of the word "include" left justified, followed by spaces, and followed by the name of the file that is to be included at this line. Macros in the name of the included file are expanded before the file is included. Include files may be nested.

An *export* line is used for exporting a macro definition to the environment of any command executed by **mk196**. Such a line starts with the word "export", followed by one or more spaces and the name of the macro to be exported. Macros are exported at the moment an export line is read. This implies that references to forward macro definitions are equivalent to undefined macros.

### **Conditional Processing**

Lines containing *ifdef*, *ifndef*, *else* or *endif* are used for conditional processing of the makefile. They are used in the following way:

```
ifdef macroname
if-lines
else
else-lines
endif
```

The *if-lines* and *else-lines* may contain any number of lines or text of any kind, even other *ifdef*, *ifndef*, *else* and *endif* lines, or no lines at all. The *else* line may be omitted, along with the *else-lines* following it.

First the *macroname* after the *if* command is checked for definition. If the macro is defined then the *if-lines* are interpreted and the *else-lines* are discarded (if present). Otherwise the *if-lines* are discarded; and if there is an *else* line, the *else-lines* are interpreted; but if there is no *else* line, then no lines are interpreted.

When using the *ifndef* line instead of *ifdef*, the macro is tested for not being defined. These conditional lines can be nested up to 6 levels deep.

## Macros

Macros have the form 'WORD = text and more text'. The WORD need not be uppercase, but this is an accepted standard. Spaces around the equal sign are not significant. Later lines which contain \$(WORD) or \${WORD} will have this replaced by 'text and more text'. If the macro name is a single character, the parentheses are optional. Note that the expansion is done recursively, so the body of a macro may contain other macro invocations. The right side of a macro definition is expanded when the macro is actually used, not at the point of definition.

Example:

```
FOOD = $(EAT) and $(DRINK)
EAT = meat and/or vegetables
DRINK = water
export FOOD
```

'\$(FOOD)' becomes 'meat and/or vegetables and water' and the environment variable FOOD is set accordingly by the export line. However, when a macro definition contains a direct reference to the macro being defined then those instances are expanded at the point of definition. This is the only case when the right side of a macro definition is (partially) expanded. For example, the line

```
DRINK = $(DRINK) or beer
```

after the export line affects '\$(FOOD)' just as the line

```
DRINK = water or beer
```

would do. However, the environment variable FOOD will only be updated when it is exported again.



You are advised not to use the double quotes (") for long filename support in macros, otherwise this might result in a concatenation of two macros with double quotes (") in between.

## Special Macros

**MAKE** This normally has the value **mk196**. Any line which invokes MAKE temporarily overrides the **-n** option, just for the duration of the one line. This allows nested invocations of MAKE to be tested with the **-n** option.



**MAKEFLAGS**

This macro has the set of options provided to **mk196** as its value. If this is set as an environment variable, the set of options is processed before any command line options. This macro may be explicitly passed to nested **mk196**'s, but it is also available to these invocations as an environment variable. The **-f** and **-d** flags are not recorded in this macro.

**PRODDIR** This macro expands the name of the directory where **mk196** is installed without the last path component. The resulting directory name will be the root directory of the installed 80196 package, unless **mk196** is installed somewhere else. This macro can be used to refer to files belonging to the product, for example a library source file.

Example:

```
DOPRINT = $(PRODDIR)/lib/src/_doprint.c
```

When **mk196** is installed in the directory `/c196/bin` this line expands to:

```
DOPRINT = /c196/lib/src/_doprint.c
```

**SHELLCMD**

This contains the default list of commands which are local to the SHELL. If a rule is an invocation of one of these commands, a SHELL is automatically spawned to handle it.

**\$** This macro translates to a dollar sign. Thus you can use "\$\$" in the makefile to represent a single "\$".

There are several dynamically maintained macros that are useful as abbreviations within rules. It is best not to define them explicitly.

**\$\*** The basename of the current target.

**\$<** The name of the current dependency file.

**\$@** The name of the current target.

**\$?** The names of dependents which are younger than the target.

**\$!** The names of all dependents.

The `$<` and `$*` macros are normally used for implicit rules. They may be unreliable when used within explicit target command lines. All macros may be suffixed with `F` to specify the Filename components (e.g. `${*F}`, `${@F}`). Likewise, the macros `$*`, `$<` and `$@` may be suffixed by `D` to specify the directory component.



The result of the `$*` macro is always without double quotes (`"`), regardless of the original target having double quotes (`"`) around it or not. The result of using the suffix `F` (Filename component) or `D` (Directory component) is also always without double quotes (`"`), regardless of the original contents having double quotes (`"`) around it or not.

### Functions

A function not only expands but also performs a certain operation. Functions syntactically look like macros but have embedded spaces in the macro name, e.g. `$(match arg1 arg2 arg3)`. All functions are built-in and currently there are five of them: `match`, `separate`, `protect`, `exist` and `nexist`.

The `match` function yields all arguments which match a certain suffix:

```
$(match .obj prog.obj sub.obj mylib.lib)
```

will yield

```
prog.obj sub.obj
```

The `separate` function concatenates its arguments using the first argument as the separator. If the first argument is enclosed in double quotes then `'\n'` is interpreted as a newline character, `'\t'` is interpreted as a tab, `'\ooo'` is interpreted as an octal value (where, `ooo` is one to three octal digits), and spaces are taken literally. For example:

```
$(separate ", &\n" prog.obj sub.obj)
```

will result in

```
prog.obj, &
sub.obj
```

Function arguments may be macros or functions themselves. So,

```
$(separate ", &\n" $(match .obj $!))
```

will yield all object files the current target depends on, separated by a comma – ampersand – newline string.

The `protect` function adds one level of quoting. This function has one argument which can contain white space. If the argument contains any white space, single quotes, double quotes, or backslashes, it is enclosed in double quotes. In addition, any double quote or backslash is escaped with a backslash.

Example:

```
echo $(protect I'll show you the "protect" function)
```

will yield

```
echo "I'll show you the \"protect\" function"
```

The `exist` function expands to its second argument if the first argument is an existing file or directory.

Example:

```
$(exist test.c c196 test.c)
```

When the file `test.c` exists it will yield:

```
c196 test.c
```

When the file `test.c` does not exist nothing is expanded.

The `nexist` function is the opposite of the `exist` function. It expands to its second argument if the first argument is not an existing file or directory.

Example:

```
$(nexist test.obj c196 test.c)
```

## ***Targets***

A target entry in the makefile has the following format:

```
target ... : [dependency ...] [; rule]
           [rule]
           ...
```

Any line which does not have leading white space (other than macro definitions) is a 'target' line. Target lines consist of one or more filenames (or macros which expand into same) called targets, followed by a colon (:). The ':' is followed by a list of dependent files. The dependency list may be terminated with a semicolon (;) which may be followed by a rule or shell command.

Special allowance is made on MS-DOS for the colons which are needed to specify files on other drives, so for example, the following will work as intended:

```
c:foo.obj : a:foo.c
```

If a target is named in more than one target line, the dependencies are added to form the target's complete dependency list.

The dependents are the ones from which a target is constructed. They in turn may be targets of other dependents. In general, for a particular target file, each of its dependent files is 'made', to make sure that each is up to date with respect to its dependents.

The modification time of the target is compared to the modification times of each dependent file. If the target is older, one or more of the dependents have changed, so the target must be constructed. Of course, this checking is done recursively, so that all dependents of dependents of dependents of ... are up-to-date.

To reconstruct a target, **mk196** expands macros and functions, strips off initial white space, and either executes the rules directly, or passes each to a shell or COMMAND.COM for execution.

For target lines, macros and functions are expanded on input. All other lines have expansion delayed until absolutely required (i.e., macros and functions in rules are dynamic).

### ***Special Targets***

**.DEFAULT:**

The rule for this target is used to process a target when there is no other entry for it, and no implicit rule for building it. **mk196** ignores all dependencies for this target.

**.DONE:**

This target and its dependencies are processed after all other targets are built.

**.IGNORE:**

Non-zero error codes returned from commands are ignored. Encountering this in a makefile is the same as specifying **-i** on the command line.

**.INIT:**

This target and its dependencies are processed before any other targets are processed.

- .SILENT:** Commands are not echoed before executing them. Encountering this in a makefile is the same as specifying **-s** on the command line.
- .SUFFIXES:** The suffixes list for selecting implicit rules. Specifying this target with dependents adds these to the end of the suffixes list. Specifying it with no dependents clears the list.
- .PRECIOUS:** Dependency files mentioned for this target are not removed. Normally, **mk196** removes a target file if a command in its construction rule returned an error or when target construction is interrupted.

## Rules

A line in a makefile that starts with a TAB or SPACE is a shell line or rule. This line is associated with the most recently preceding dependency line. A sequence of these may be associated with a single dependency line. When a target is out of date with respect to a dependent, the sequence of commands is executed. Shell lines may have any combination of the following characters to the left of the command:

- @ will not echo the command line, except if **-n** is used.
- **mk196** will ignore the exit code of the command, i.e., the ERRORLEVEL of MS-DOS. Without this, **mk196** terminates when a non-zero exit code is returned.
- + **mk196** will use a shell or COMMAND.COM to execute the command.

If the '+' is not attached to a shell line, but the command is a DOS command or if redirection is used (<, |, >), the shell line is passed to COMMAND.COM anyway. For UNIX, redirection, backquote (`) parentheses and variables force the use of a shell.

You can force **mk196** to execute multiple command lines in one shell environment. This is accomplished with the token combination `;\'.

Example:

```
cd c:\c196\bin ;\
c196 -V
```



The ';' must always directly be followed by the '\' token. Whitespace is not removed when it is at the end of the previous command line or when it is in front of the next command line. The use of the ';' as an operator for a command (like a semicolon ';' separated list with each item on one line) and the '\' as a layout tool is not supported, unless they are separated with whitespace.

**mk196** can generate inline temporary files. If a line contains '<<WORD' then all subsequent lines up to a line starting with WORD, are placed in a temporary file. Next, '<<WORD' is replaced by the name of the temporary file.



No whitespace is allowed between '<<' and 'WORD'.

Example:

```
rl196 & < <<EOF
$(separate " , &\n" $(CSTART) $(match .obj $!) $(match .lib $!) $(LIBS)) &
to $@ &
$(separate " &\n" $(LDFLAGS))
EOF
```

The three lines between the tags (EOF) are written to a temporary file (e.g., "tmp/mk2"), and the command line is rewritten as "rl196 & < tmp/mk2".

### ***Implicit Rules***

Implicit rules are intimately tied to the .SUFFIXES: special target. Each entry in the .SUFFIXES: list defines an extension to a filename which may be used to build another file. The implicit rules then define how to actually build one file from another. These files are related, in that they must share a common basename, but have different extensions.

If a file that is being made does not have an explicit target line, an implicit rule is looked for. Each entry in the .SUFFIXES: list is combined with the extension of the target, to get the name of an implicit target. If this target exists, it gives the rules used to transform a file with the dependent extension to the target file. Any dependents of the implicit target are ignored.

If a file that is being made has an explicit target, but no rules, a similar search is made for implicit rules. Each entry in the `.SUFFIXES:` list is combined with the extension of the target, to get the name of an implicit target. If such a target exists, then the list of dependents is searched for a file with the correct extension, and the implicit rules are invoked to create the target.

## 8.4 EXAMPLE

This makefile says that `serialk.out` depends on two files `serialk.obj` and `cstart.obj`, and in turn they depend on their corresponding source files (`serialk.c` and `cstart.a96`).

The makefile uses the implicit rules (from `mk196.mk`) to perform compilation, assembly, linking, and converting to HEX.

```
#
# Makefile for the serialk example.
#

MODEL =      md(kd)
CCFLAGS    = $(MODEL) type debug code dn(0)
CSTART     = $(C196LIB)/cstart.obj
LIBS       = $(C196LIB)/c96.lib
LDFLAGS    =      ss(+20) ra(1ah-1ffh,5000h-7fffh) ro(2000h-4fffh)
ixref

all:        all_hex

all_hex:    all_out \
            serialk.hex

all_out:    serialk.out

# Serialk is an example for 196KD
# Explicit rules are still needed to enforce suffix-rules.

serialk.hex: serialk.out
serialk.out: cstart.obj serialk.obj
serialk.obj: serialk.c
cstart.obj:  cstart.a96
```

See the `examples` directory for a more detailed example of a makefile.

## **8.5 FILES**

makefile	Description of dependencies and rules.
Makefile	Alternative to makefile, for UNIX.
mk196.mk	Default dependencies and rules.

## **8.6 DIAGNOSTICS**

**mk196** returns an exit status of 1 when it halts as a result of an error. Otherwise it returns an exit status of 0.



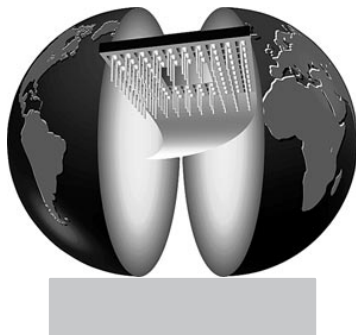


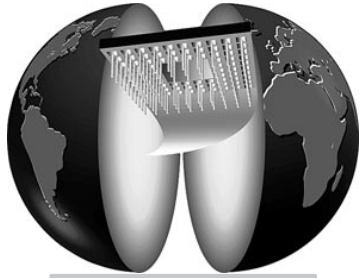
# CHAPTER

# 9

## MESSAGES AND ERROR RECOVERY

---





# 9

# CHAPTER

This chapter provides a list of all the console, warning, error, and fatal error messages produced by the RL196 linker, OH196 converter, and LIB196 librarian.

The text of each message is in uppercase; placeholders in the message are shown in lowercase italics. A brief explanation of the probable cause for the error condition accompanies each message.

## 9.1 RL196 MESSAGES

The RL196 linker generates three different error messages: warnings, errors, and fatal errors. A warning reports a suspicious condition that you might want to change. A warning does not terminate the link-locate operation. Neither does an error but the resulting output module might be unusable. A fatal error, on the other hand, terminates the link-locate operation immediately.

If an error occurs in a segment, RL196 displays the names of the file and module containing the segment and the segment's type classification.

RL196 supplies the public symbols and when `absstack` is in effect. The associated module and file names are `<Dummy>` when `MEMORY` and `?MEMORY_SIZE` appear in the symbol table, the intermodule cross-reference listing, or the error messages.

If the *offset* parameter appears in some of the messages, the *offset* is simply the offset from the segment base to be used if the associated segment is relocatable. If the associated segment is absolute (i.e., located), that offset displayed is actually an absolute address.

### 9.1.1 CONSOLE MESSAGES

RL196 sends a sign-on message and a sign-off message to the console. The sign-on message appears when you invoke the linker.

The sign-on message appears in the following format:

```
80C196 relocater/linker vx.y rz   SN00000-005 (c)year TASKING, Inc.
```

where:

`vx.y` identifies the version of the assembler.

*rz* identifies the revision of the assembler.

*year* identifies the copyright year.

When the linker completes its processing, the following sign-off message is sent to the console:

```
RL196 COMPLETED, nnn WARNING(S), mmm ERROR(S)
```

where:

*nnn* is the number of warnings issued by RL196.

*mmm* is the number of errors issued by RL196.

If *mmm* is not zero, the output object file is marked as erroneous. If a fatal error occurs, an error message to that effect replaces the sign-off message.

### **9.1.2 FATAL ERRORS**

Upon detecting a fatal error within the system hardware or on the invocation line, RL196 prints a message on the screen, terminates the linking/locating processing, and returns control to the host system.

Fatal error messages can be caused by the following:

- invocation-line errors
- memory errors
- I/O errors

#### **9.1.2.1 RL196 ERROR MESSAGES**

The linker displays fatal RL196 errors in the following form:

```
FATAL RL196 ERROR num: message
```

where:

*num* is an error number.

*message* is a message describing the cause of the error.

The following list of error messages provides their decimal codes and their meanings.

**201: *Invalid command line syntax: token***

A syntax error was detected in the command line near the specified token.

**202: *Invalid command line, token too long*****203: *Expected item missing***

An expected item in the command line (e.g., an input filename or a filename following `to`) is missing. A filename longer than 128 characters also causes these errors.

**204: *Invalid keyword***

An invalid keyword was found in the command line.

**206: *Invalid constant: name***

An illegally constructed constant was found in the command line. For example, RL196 found a hexadecimal number that begins with a letter. These hexadecimal numbers must be preceded with a zero (0).

**207: *Invalid module name: name***

The specified *name* on the command line is an illegal module name. See name control in Chapter 2.

**208: *Invalid file name***

The specified file on the command line is an illegal filename. See Chapter 2 for more information on filenames.

**209: *File used in conflicting contexts: filename***

The specified file was used in more than one context, for example, for both input and output. This error can be due to the default rules regarding the output object filename and the print filename. For example, if the first input filename is `[directory / device] ABCD` and no output filename is specified by a `to` keyword, the output object filename, by default, also becomes `[directory device] ABCD`.

**210: *I/O error, input file***

The linker detected an I/O error while accessing the specified input file.

**211: *I/O error, output file***

The linker detected an I/O error while accessing the specified output file.

**212: I/O error, print file**

The linker detected an I/O error while accessing the specified print file.

**213: Duplicate keyword: keyword**

The specified keyword appears in the command line more than once.

**215: Checksum error**

The linker detected a bad checksum in the specified input module. This message indicates a bad input module.

**217: No module to be processed**

After scanning all the input files, RL196 selected no module to process. This message usually indicates an empty input file. This error also appears if you link an empty a library or a publiconly file.

**218: Invalid input object file: filename**

The specified file is not a valid object file. Possible causes are incorrect record order, incorrect record type, illegal field, illegal relation between fields, or a required record is missing. This error can be the result of a translator error, a librarian error, or a disk error.

**219: Not an 8096 object module**

The translator\_id field in the module header record indicates that the specified module is not an OMF96 module. Another possible cause is the RL196 version is not compatible with the translator (or tool) that has produced the object file. Check the RL196 version.

**220: No object file to be processed**

RL196 expects at least one object file to process.

**221: Addresses not in ascending order: addr1-addr2**

Addresses specified in a locating control are not in ascending order. For example, the following control line generates this error:

```
rom( 3000H-4000H(mod1) , 3000H-4000H(mod2) )
```

**222: Address out of range: address**

An address specified in a locating control is out of the permissible range. Either the address in the rom control is below 100H or the address in the ram control is below 1AH.

**223: *Overlap between ROM and RAM ranges***

The ROM and the RAM sections, as specified by the `rom` and `ram` controls, overlapped. ROM and RAM sections are not allowed to overlap unless the `inst` control is in effect. Recheck the address range you specified in the `ram` and `rom` controls.

**224: *The STACK may not be specified there as a module***

`Stack` (or `st`) has been specified where a module name is expected. For example, you specified `stack` with the `rom` control or with the `regoverlay` control.

**225: *Internal processing error***

RL196 has made a processing error. This error message indicates a problem within RL196. Report such errors to TASKING by calling your local TASKING sales representative.

**226: *"PURGE(SEGMENTS)" and "NOABSSTACK" not allowed simultaneously***

A relocatable stack and a segment definition purging are incompatible because the result of the purging is no stack segment at all.

**227: *Module used in conflicting contexts: filename(module\_name)***

The specified module name has already been specified explicitly at the command line in another conflicting context. For example, the same module name appears more than once in association with the same locating control (`ram` or `rom`), or the same module name is specified for more than one input file.

**228: *Parameter is not allowed in that context: name***

The negative form of the control (with the prefix `no`) cannot have parameters.

**229: *The prefix "NO" is not allowed for this control***

The control has no negative form.

**230: *Invocation line too long***

The invocation line is too long, that is, it contains too many characters.



**231: An ordinary file may not be specified with a module list: filename**

RL196 process an ordinary file as a whole; modules can only be selected out of a library file.

**232: A library file may not be specified with "PUBLICSONLY"**

Only an ordinary file can be specified as `publicsonly`. See Chapter 2 for definitions of ordinary files and `publicsonly` files.

**233: The specified module does not exist in the specified file:**

*filename(module\_name)*

The module specified in module list of the library file does not exist. Use the LIB196 `list` command to display all objects modules in a library.

**234: I/O error, ixref file**

During the preparation of the `ixref` listing, RL196 uses an auxiliary temporary file. The above error indicates that an I/O error was detected during file access.

**235: The specified module does not exist: module\_name**

A module specified as a parameter of either the `ram`, `rom` or `regoverlay` control does not exist.

**236: PARAMETER OUT OF RANGE: name**

The specified parameter does not lie in its legal range. For example, the parameter specified as `pagewidth` must be in the inclusive range 72 to 132.

**237: REGOVERLAY parameter too complex**

The specified parameter is too complex (e.g., too long) to be processed by RL196. Simplify your parameter.

**242: Invalid register range: addr1-addr2**

The address range specified in the `registers` control does not conform to the component's register space. See the *Embedded Microcontrollers and Processors Handbook*, listed in *Related Publications*, for memory space information.

**243: Window size specified for default register space**

A window size other than 0 was specified for the default register range of 1AH to 0FFH (i.e., a component without vertical windowing feature).

**244: *Invalid window size specified for register space***

A window size of 0 was specified for a register space larger than 256 bytes. Do not specify the `registers` control if you are not using vertical windows.

**245: *Illegal segment for current model***

The `farconst` or `fardata` was specified in the `rom` or `ram` control and the `model` control was not specified.

**246: *Segment cannot be placed in address range***

An attempt was made to place a code, constant or data segment above 0FFFFH, or a high code segment below 0FF0000H.

**247: *Segment cannot be placed in ROM***

Data and far data segments cannot be placed in ROM.

**248: *Segment cannot be placed in RAM***

Code, near constant, and far constant segments cannot be placed in RAM.

**249: *Invalid segment name***

An unknown segment name was specified in a `rom` or `ram` control.

**250: *Conflict between MODEL and INST controls***

The `model` and `inst` controls cannot coexist.

**251: *Invalid model***

The only models allowed are 24-bit models.

**252: *Segment used in conflicting contexts***

An attempt was made to place the same segment from the same module in two different ROM/RAM sections, or to demand multiple placement in one section.

**253: Segment type incompatible with current model:***filename(module-name), segment-type*

The segment type in the module is not compatible with the model. That is, if no model control was specified, a far code, far data, far const or high code segment was found. If a 24-bit model in compatible mode was specified, a code or far code segment was found. If a 24-bit model in extended mode was specified, a code or high code segment was found.

**254: Internal error, please report: message**

This error should not occur. If it does, report it to your local TASKING representative.

**255: This DEMO RL196 has reached its limit.**

You have a restricted demo version of the linker. Contact TASKING for a registered version.

**256: Too many input files for this DEMO RL196: filename.**

You have a restricted demo version of the linker. Contact TASKING for a registered version.

**257: Conflict between selection of OMF96 version.**

You may have used features not present in the selected OMF96 version. Check for the correct omf control.

**258: Invalid OMF96 version requested.**

You can only use omf ( 0 ), omf ( 1 ) or omf ( 2 ) for the OMF96 versions V2.0, V3.0 or V3.2 respectively.

**259: Control control is obsolete, use control instead**

The control oo1 is no longer valid. It has been replaced by the omf control.

**260: Invalid range for STACK: addr1-addr2**

The address range specified in the ram control does not conform to the component's stack space. The minimum address of *addr1* is 1AH and the maximum address of *addr2* is 0FFFFH for data segments of 0FFFFFFH for far data segments.

**261: Control invalid in current context: control**

An invalid control is specified on the command line. Check your controls.

**262: *Expression evaluation stack overflow***

The expression which is being evaluated is too complex. Try to simplify the expression in the assembler.

**263: *Expression evaluation stack underflow***

This error should not occur. If it does, report it to your local TASKING representative.

**264: *Cannot swap expressions on evaluation stack***

This error should not occur. If it does, report it to your local TASKING representative.

**265: *Control only valid for model NP or NU.***

The `np_rsvup6` control is specified without using an NP or NU model.

**9.1.2.2 ARGUMENT ERROR MESSAGES**

RL196 displays fatal argument errors in the following form:

```
FATAL ARGUMENT ERROR num: message
```

where:

*num* is an error number.

*message* is a message describing the cause of the error.

The following list of error messages provides their decimal codes and their meanings.

**0: *Unexpected end of argument: arg***

Check and correct the syntax.

**1: *Control or option cannot be negated: name***

The control *name* cannot have a no prefix, or the option *name* cannot have a minus sign appended. Remove the negation.

**2: *Syntax error in control: control***

Check and correct the syntax.

**3: *Argument expected for control or option: name***

Specify an argument to *name*.

**4: *Syntax error in option: option***

Check and correct the syntax.

**5: *Unknown option specified: name***

Replace *name* with the correct option.

**6: *Maximum depth in buffer stack reached***

The control or option has too many argument levels. Reduce the number of argument levels.

**7: *Buffer stack is empty***

This error should not occur. If it does, report it to your local TASKING representative.

**8: *Argument too long***

Reduce the length of the argument.

**9: *Unexpected argument for control: name***

Control *name* cannot have an argument. Remove the argument.

**10: *Unexpected internal error: message***

This error should not occur. If it does, report it to your local TASKING representative.

**9.1.2.3 MEMORY ERROR MESSAGES**

RL196 displays fatal memory errors in the following form:

```
FATAL MEMORY ERROR num: message
```

where:

*num* is an error number.

*message* is a message describing the cause of the error.

The following list of error messages provides their decimal codes and their meanings.

**0: *Cannot allocate memory block of size size***

**1: *Cannot reallocate memory block to size size***

The memory available for execution of RL196 has been exhausted. This error is usually caused by the program containing too many external or public symbols, or containing a large number of publics or externals references when the `ixref` control was specified. In the latter case, link with `noixref`, the default mode. Not enough free conventional memory check can also cause this error.

### **9.1.2.4 I/O ERROR MESSAGES**

RL196 displays fatal I/O errors in the following form:

```
FATAL I/O ERROR num: message
```

where:

`num` is an error number.

`message` is a message describing the cause of the error.

The following list of error messages provides their decimal codes and their meanings.

**0: *Unexpected end of file detected***

Check your file.

**1: *Cannot write to standard input***

Specify a file or standard output to write to. Standard input is used for input only.

**2: *Cannot read from standard output***

Specify a file or standard input to read from. Standard output is used for output only.

**3: *Filename too long***

Give your file a shorter name.

**4: *Filename not conform DOS standard***

Check your DOS Reference Manual for the correct filename syntax.

**5: *Cannot read from null device***

Specify another device or filename to read from.

**6: *Cannot rename :WORK:***

This is a temporary file. So, you cannot rename it.

**9.1.3 ERROR MESSAGES**

The linker displays RL196 errors in the following form:

```
ERROR num: message
```

Where:

*num* is an error number.

*message* is a message describing the cause of the error.

The following list of error messages provides their decimal codes and their meanings.

**101: *Code memory overlap: addr1-addr2***

Two or more absolute segments occupy the memory in the given inclusive range. One of the absolute segments is a code segment. Any overlap with the reserved section (address 0 to 1AH) also triggers this error.

**102: *Erroneous input module: filename(module\_name)***

The specified module generated by the translator or a previous link step contains erroneous data. Warnings do not have this effect. RL196 marks an output object module as erroneous. Be sure that all input modules are processed without any errors.

**103: Memory overflow: segment type in filename(module\_name)**

RL196 was unable to allocate the specified relocatable segment in the appropriate memory section. Therefore, the specified segment does not appear in the segment map. All of the symbols (and lines) that are relative to the above segment are improperly located in the output object module. The same is true for code sections if they belong to a code segment for which this error is issued.

**105: Incompatible stack segments: filename(module\_name)**

The specified module contains a stack segment that is incompatible with the stack segments already processed. Incompatibility between stack segments occurs when more than one stack segment exists, and one or more of these segments is absolute. RL196 takes no action on the indicated stack segment.

**110: No room for relocatable segment in the specified range: filename(module\_name), segment\_type(addr1-addr2)**

This message indicates that RL196 was unable to allocate memory in the requested range for the specified segment because of lack of memory space. For example, memory allocation can fail because the segment size was larger than the specified range, or the segment does not fit in the remaining memory.

**111: Instruction operand doesn't meet alignment requirement in filename(module\_name) at segment\_type(offset)**

A code segment of the specified module contains the indicated violation. The *offset* placeholder is the offset of the first byte of the operand reference (relative to the beginning of the segment in a relocatable segment). In absolute segments, *offset* is the absolute address of the operand reference. The problem can be caused, for example, by using a byte-type symbol where a word-type symbol is needed or by some wrong assumptions about an external symbol.



**112: Instruction operand out of range in a JUMP/CALL instruction in filename(module\_name) at segment\_type(offset)**

A code segment of the specified module contains the indicated violation. The *offset* placeholder is the offset of the first byte of the operand reference, relative to the beginning of the segment in a relocatable segment. In absolute segments, *offset* is the absolute address of the operand reference. This error occurs when the distance between two statements in the absolute code segments exceeds the maximum size that can be specified with the selected jump or call instruction.

**113: Instruction operand out of range in filename(module\_name) at segment\_type(offset)**

A code segment of the specified module contains the indicated violation. The *offset* placeholder is the offset of the first byte of the operand reference (relative to the beginning of the segment in a relocatable segment). In absolute segments, *offset* is the absolute address of the operand reference. Basically, the error results from the same kind of mistakes that occurred in Errors 111 and 112.

Another possible cause: you compiled your program with `registers(all)`, the default for C196 programs. The compiler assumes that all symbols are placed in the register space and generates assembly instructions with register operands. If your program contains symbols located in external memory, the assembly instructions generated are not valid for external memory access. If you have variables in external memory, do not compile with this option: `registers(all)`.

**115: Module not compiled for windows: filename(module\_name)**

For C196 programs, one of the input modules was not compiled with the windows control. For ASM196 programs, no reference to `?wsr` was found. See Chapter *Processor Registers* of the *80C196 C Compiler User's Guide* for an example of an ASM196 module written for vertical windows.

**116: Too many global registers**

During incremental links, RL196 locates all register segments in the unmapped portion of the register space and the overlay segments in the mapped area of the register space. This error can occur when the register segment requires a larger space than the unallocated unmapped register space, or a register segment was allocated above an overlay segment. This error can also occur if the total requirement of the register segments is too large that the registers were allocated beyond `0FFH`. Decrease the number of global registers in your modules.

**117: *Illegal forward type reference***

A module is specified in a command line locating control (`ram`, `rom`, `romcode`, `romdata`), but the referenced segment is not present in the module.

**118: *Invalid expression operand.***

A module is specified in a command line locating control (`ram`, `rom`, `romcode`, `romdata`), but the referenced segment is not present in the module.

**119: *Invalid floating point expression***

An operand is non-integral, but the operator requires integral operands. That is, for example, NOT, AND, OR, XOR all require integral operands. Check the assembler source file.

**120: *Character string value expected***

A string is expected. Check the assembler source file.

**121: *Operator "operator" can only be used in absolute expression***

The LOW or HIGH keyword is not used with an absolute expression. Check the assembler source file.

**122: *Attempt to divide by zero.***

The linker encounter an expression where an attempt is made to divide by zero. Check your source code.

**123: *Floating point constant underflow***

The absolute value of floating-point constant must be above 1.17E-38

**124: *Floating point constant overflow***

The absolute value of floating-point constant must be below 3.37E38

**125: *More than one absolute segment definition in module***

*module\_name: segment segment\_name*

When you have more than one module with the same name, you can only have one absolute segment definition per segment. Change your source code so only one segment is defined absolute.

**126: *Initialized data was not located***

When using initialized data, the linker needs to locate two sections: one in ROM and one in RAM. Either one or both of these sections could not be located.

**127: *More than one user defined stack in the input modules***

There is more than one module in which a SSEG segment is defined. Change your source code so only one SSEG is present for each project.

**9.1.4 WARNINGS**

The linker displays warning messages in the following form:

```
WARNING num: message
```

where:

*num* is a warning number.

*message* is a message describing the cause of the warning.

The following list of warning messages provides their decimal codes and their meanings.

**1: *Symbol attribute mismatch: symbol\_name, defined in filename(module\_name), referenced in filename(module\_name)***

The attributes of the specified symbol (external or public) in the second module do not match the attributes in the first module. To find the actual attributes of the symbol in the output object file, look at the symbol table or the ixref listing in the .m96 print file.

**2: *Unresolved external symbol: external\_name in filename(module\_name)***

The specified symbol was declared as external in the specified module. No public symbol with the same name was found in any of the input modules. This error message is issued only for the first module that contains this unresolved external. Look at the ixref listing for a complete list of modules in which this name was declared external. This warning does not imply that the module actually used that external symbol. Such a case is indicated by Warning 4.

**3: Multiple public definition:** *public\_name, filename(module\_name) and in filename(module\_name)*

The specified symbol was declared as public in the first specified module and also in the second module. RL196 does not use the declaration in the second module. The only exception is that in the ixref listing, the name of the second module also appears in the line corresponding to the specified symbol.

**4: Reference made to unresolved external:** *external\_name in filename(module\_name) at segment\_type(offset)*

No public definition was found for the referenced external symbol. The *offset* placeholder indicates the location in which the reference was made. This message appears once per each reference to the specified symbol.

**5: Module name not unique:** *filename(module\_name)*

Another module with the same name has been processed. RL196 does not process the specified module of the specified file. Recheck all your module names or use the `uniquemods` control.

**6: More than one MAIN module:** *filename(module\_name)*

The specified module in the specified file was not the first processed module marked as `main`. For example, if your program contains four input modules all marked as `main`, RL196 issues this error for the last three modules. The output object module is marked as `main`.

**7: Specified stack size too small:** *size*

The size you specified in the `stacksize` control is less than the computed size of the stack based on the contents of the input file(s). The specified stack size overrides the computed stack size (see also WARNING 9).

**8: Illegal specified stack size, should be even.**

The stack size must be an even number because stack operations are performed on word items. RL196 adds 1 to the specified stack size (modulo  $2^{16}$ ).

**9: Stack already located, "STACKSIZE" ignored.**

The `stacksize` control was specified but the stack segment of the input modules is already absolute (i.e., located). RL196 takes no action on the `stacksize` control.

**10: Data memory overlap: *addr1-addr2***

The memory in the given (inclusive) range is occupied by two or more absolute segments. None of the overlapping segments is a code segment.

**11: STACKSIZE parameter is odd (=num), incremented to make it even.**

The stack size must be an even number because stack operations are performed on word items. RL196 adds 1 to the specified stack size (modulo  $2^{16}$ ).

**12: Specified stack size too large**

The sum of stack size calculated from the total stack segments in the input files and the increment specified in the invocation line is greater than 0FFFEH. Stack size is set to 0FFFEH.

**13: Stack already located, "NOABSSTACK" ignored.**

The noabsstack control was specified but the stack segment of the input modules is already absolute (i.e., located). RL196 takes no action on the noabsstack control.

**14: No MAIN module**

None of the input modules is a main module. Consequently, the output object module also is not marked as main. For C196 applications, mark one module as main using ASM196 or include the `cstart.obj` in your RL196 invocation.

**16: Symbol defined out of segment: *symbol\_name* in *filename(module\_name)* at *segment\_type(offset)***

The symbol was defined outside the segment to which it belongs. This error usually occurs when a symbol was defined via the `equ` directive, in ASM196.

**17: Absolute segment does not fit: *filename(module\_name)*, *segment\_type(addr1-addr2)***

The absolute segment you named to occupy the specified range does not fit inside one of the corresponding memory sections (code, rom, data, stack, ram, register, or overlay). This warning can be the result of employing an incremental link-locate while changing the ROM and/or RAM sections between the steps.

**20: Type definition too complex**

The specified module contains a type definition that is too complex to be processed by RL196. The linker simplifies the definition. In some cases, the too-complex type definition can be the result of two or more type definitions from different modules. If so, the linker generates the error message on only one of definitions.

**21: A direct call between two overlaid modules: *symbol\_name*, defined in *filename(module\_name)*, referenced in *filename(module\_name)***

The second module specified contains a call or a jump to the specified public symbol of the first module. However, the `regoverlay` control specifies that the two modules can be overlaid.



If you specified `regoverlay`:

- The overlaying takes place.
- The warning indicates a possible, but not necessarily wrong, overlaying.
- Lack of such errors does not imply a correct overlaying. See the `regoverlay` control entry in Chapter 2.

**22: Too many global registers**

The total requirement of the register segments for the input modules was so large that it exceeded the window base of the smallest window (0E0H). In this case, RL196 could not generate vertical windows and uses no register space above 0FFH. RL196 then locates the overlay segments in the remaining unallocated register space up to 0FFH. If all of your overlay segments do not fit under 0FFH, RL196 generates a memory overflow error. To resolve this problem, reduce the number of global register variables.

**23: Window size specified too large, ignored**

RL196 cannot generate a window with the window size you specified in the `window size` control. The linker uses the biggest possible window size.

**24: Module *module* does not have the expected segment(s): *segment***

A module is specified in a command line locating control (`ram`, `rom`, `romcode`, `romdata`), but the referenced segment is not present in the module.

**25: Possible OMF version clash in `filename(module_name)`**

The linker is invoked with a lower version of OMF than the one of the input module. Invoke the linker with the correct `omf` version control.

**26: Expression name in `filename(module_name)` contains unresolved references**

The expression could not be solved because it contains an unresolved external. See also warning #2.

**27: FLOAT truncated to INT**

A float is casted to an integer during expression evaluation. This can result in a loss of precision. Check if the cast is necessary.

**28: CODE2HIGH control is disabled for 16-bit models**

This control was used while a 16-bit model was in effect. See description `code2high` control.

**9.2 OH196 ERROR MESSAGES**

OH196 errors are always fatal errors. When an error occurs, processing of the object file is stopped and one of the following error messages is issued:

**\*\*\* ERROR – invocation should be: OH196 <infile> [TO outfile]**

This message means that the invocation syntax is incorrect. OH196 expected the `to` keyword but found something else. Reinvoke OH196 using the correct syntax.

**\*\*\* ERROR – input is not an absolute object file**

This message means that the object file is not absolutely located.

**\*\*\* ERROR – input has a record longer than 32K, sorry**

Make sure the records in the input object file are less than 32K bytes. Reduce record size in your applications code space by breaking modules into smaller functions, procedures, or subroutines. Reduce the size of a data record, for example, a large array, by breaking it into two or more smaller structures.

**\*\*\* ERROR - on Reading OBJECT: I/O error in invalid object file.**

This messages means that OH196 has detected an invalid format. Be sure the input object file is available and that it is an absolutely located file.

### **9.3 LIB196 ERROR MESSAGES**

The following is a list of LIB196 error messages and their probable causes:

*pathname*, **ATTEMPT TO ADD DUPLICATE MODULE**

The specified module name already appears within the library.

*pathname*, **BAD RECORD SEQUENCE**

This error is usually caused by an I/O error or a translation error.

*pathname*, **CHECKSUM ERROR**

The specified file has an error in one of its checksum fields. This is usually the result of an I/O error.

*pathname*, **DUPLICATE SYMBOL IN INPUT**

You have attempted to add or replace a module containing a public symbol that is already within the library.

*pathname*, **FILE ALREADY EXISTS**

The specified file in the create command already exists. Choose another name for the library.

*pathname*, **ILLEGAL RECORD FORMAT**

This error is usually caused by an I/O error or a translation error.

**INSUFFICIENT MEMORY**

LIB196 cannot execute the command because it requires more memory than the amount of memory available in the system.

**INVALID MODULE NAME**

The specified module name contains an invalid character or starts with a digit.



*pathname, **NOT LIBRARY***

The specified file is not a library.

***INVALID SYNTAX***

The command was not entered properly. Reenter it using the correct syntax.

***MODULE NAME TOO LONG***

The specified module name exceeds 40 characters.

***RIGHT PARENTHESIS EXPECTED***

A " ) " is missing in the command.

***UNRECOGNIZED COMMAND***

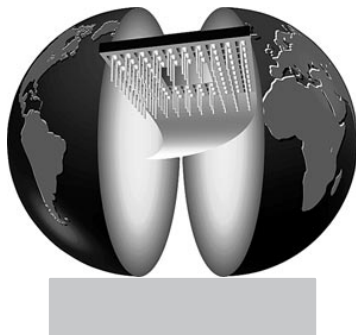
An illegal or misspelled command was entered. The only commands are add, create, delete, exit, extract, help, list, replace, and their respective abbreviations.

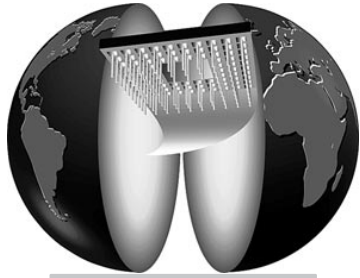
# APPENDIX

## GLOSSARY

---

# A





A

APPENDIX

**A**

**absolute address.** An address that corresponds directly to a storage location in the processor's address space. See relocatable address.

**absolute object file.** An object file containing no relocatable segments.

**absolute segment.** A segment of code or data absolutely located at a specific address.

**address.** A specific memory location.

**alignment.** The arrangement of data in memory relative to the byte boundaries of the memory location.

**B**

**base.** (1) A term used in logarithms and exponentials. In both contexts, it is a number that is being raised to a power.

(2) A number that defines the representation being used for a string of digits. Base 2 is the binary representation.

**base address.** A starting address from which an absolute address can be calculated by combination with an offset.

**bias.** A constant that is added to the true exponent of a real number to obtain the field of that number's floating-point representation in the 80C196 processor. To obtain the true exponent, you must subtract the bias from the given exponent. For the 80C196, the bias is 127.

**binary point.** An entity just like a decimal point, except that it exists in binary numbers. Each binary digit to the right of the binary point is multiplied by an increasing negative power of two.

**C**

**calling convention.** Object code inserted by the compiler to handle function calls.

**code segment.** An address space containing instructions and constants.

**constant.** A value that does not change during execution.

**control.** A command-line parameter that determines features or actions of the program being invoked.

## D

**data segment.** An address space containing data.

**data type.** A format for storing or displaying a value.

**debug information.** Information produced in the object file by the translator or linker to aid in the process of symbolic debugging.

**denormalized number.** A number whose most-significant digit is a 0.

## E

**error.** An exception that does not immediately terminate the program's operation but can cause an invalid object module.

**exception.** Any of the six conditions (invalid operation, denormal, zero divide, overflow, underflow, and precision) detected by the FPAL96 library and signalled by status flags or by status flags and exception handlers.

**exponent.** (1) Any number that indicates the power to which another number is raised. (2) A field of a floating-point number which indicates the magnitude of the number.

**external reference.** A reference to a location in a different object module via a data pointer or function call.

**external symbol.** A symbol used in the current module but defined in another module.

## F

**fatal error.** An unrecoverable error detected by the executing program.

**fixup.** Instructions placed in the object file that allow RL196 to fix undetermined calls in the code image.

**fraction.** The part of a floating-point significand that lies to the right of the binary point.

**I**

**include file.** Source text files named in an `include` compiler/assembler control or in a `#include` preprocessor directive.

**incremental linking.** Linking modules in small subgroups before linking the subgroups together.

**integral types.** Types that include all forms of integers, characters, and enumerations.

**K**

**keyword.** A character string that has special meaning to the program. See reserved word.

**L**

**library file.** A file containing a collection of linkable object modules indexed by module name.

**listing controls.** Controls that manipulate the format of the print or listing file.

**listing file.** User-readable text recording and summarizing the linking process.

**local symbols.** Symbols that are defined and used in only one module of a program.

**M**

**mantissa.** The significand of a floating-point number.

**map file.** Description of the layout of a linked program in memory.

**memory allocation.** The manner in which memory is assigned to code and data.

**module.** A separately translated part of a program.

**N**

**normalized number.** A number whose most-significant digit is a 1.

**Not-a-Number.** Value in floating-point format that does not represent any real number.

**O**

**object code.** Executable instructions and associated data in binary format.

**object file.** File containing the translated module.

**object module.** Formatted object code resulting from translation.

**offset.** A byte address within a segment.

**OH196.** 80C196 object code to hexadecimal conversion utility.

**opcode fetch.** Reading an instruction from memory.

**P**

**parameter.** A variable element in a command, such as a value, argument, or identifier.

**pathname.** The name of a directory or file relative to a given directory.

**print file.** See map file.

**Q**

**quasi-absolute file.** An object file that contains a relocatable stack segment.

**R**

**RAM.** Random access memory

**reentrant.** A function that calls itself or gets called again in a call loop.

**register.** A high-speed storage location on a processor chip.

**register file.** 80C196 on-chip memory used for high-speed data access and for hardware control; also called register memory.

**relative address.** See relocatable address.

**relocatable address.** A symbolic address generated by a language translator as a placeholder for an absolute address. The absolute address can be evaluated at a later time by language utilities.

**relocatable object file.** File containing code and data whose location is defined at load time or run time.

**reserved word.** A character or character string defined and used by the program.

**RL196 linker.** 80C196 relocation and linking utility used in preparing object code for execution.

**ROM.** Read-only memory.

**run-time.** The time during which a program is executing.

## S

**scalar.** A single value.

**scope.** The section of a program within which a symbol is recognized.

**search path.** The list of directories that the compiler or the host system can search to find a filename.

**segment.** A block of code or data that fits into an addressable block of memory.

**SFRs.** Special function registers: part of the register file of the 80C196 component used for hardware control.

**Significand.** The part of a floating-point number that consists of a leading bit to the left of the binary point and a fraction to the right.

**stack pointer.** Processor register that contains the address of the top of the stack.



**stack segment.** Portion of memory reserved for dynamic use during execution.

**symbol table.** A table in the object file containing information about the symbols used in the program.

## T

**target.** System on which the application program executes.

**target system.** The hardware and operating system for which the user is developing an application.

**translator.** An assembler, interpreter, or compiler.

**type casting.** Changing the representation of a value from one data type to another.

**type checking.** Test performed by the linker to see if two symbols of the same name have the same attribute.

## U

**unresolved external.** A symbol that is not matched by a public symbol in one of the input modules.

## V

**variable.** A quantity that can assume any of a set of values, or a symbol that refers to a value.

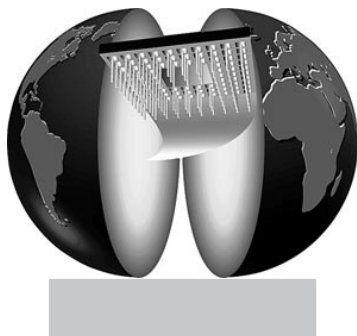
## W

**work files.** Files created and deleted by the development tool during translation.

# INDEX

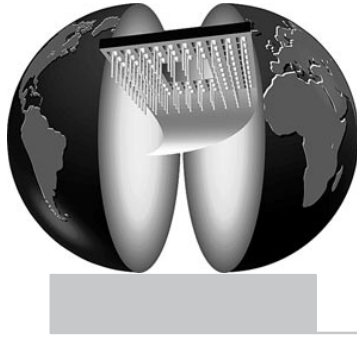
## INDEX

---



# INDEX

---



# Symbols

- .abs extension, 1-7
- .DEFAULT, 8-11
- .DONE, 8-11
- .IGNORE, 8-11
- .INIT, 8-11
- .lst extension, 1-7
- .m96 extension, 1-7
- .m96 file, 2-30
- .PRECIOUS, 8-12
- .SILENT, 8-12
- .SUFFIXES, 8-12
- ?MEMORY\_SIZE symbol, 2-15, 9-3
- ?wsr variable, 2-21
- : (colon), 2-23, 4-3
- #include directive, 5-19
- (minus sign), 2-23, 4-3
- = (equal sign), 2-23, 4-3
- \_BOTTOM\_OF\_STACK\_, 2-13
- \_HEAP\_END\_ symbol, 2-15
- \_HEAP\_START\_ symbol, 2-15
- \_INIT\_TABLE\_START\_, 2-9
- \_TOP\_OF\_STACK\_, 2-13

# Numbers

- 80C196KC processor, 2-20
- 80C196KR processor, 2-20
- 80C196NT microcontrollers, 5-10, 5-19
  - Compatible mode*, 5-10
  - Extended mode*, 5-10

# A

- a command, 4-6
- Absolute object file, 1-7
  - Creating an*, 2-38, 2-46
- Absolute segment, 2-4
- absstack control, 2-12, 2-16, 2-46

- Addition, 6-9
- Administrative operations, 6-3
  - fpcleb function*, 6-3, 6-12
  - fpinit function*, 6-3, 6-19
  - fppldcw function*, 6-3, 6-22
  - fpprstor function*, 6-3, 6-41
  - fpssave function*, 6-3, 6-43
  - fpseteb function*, 6-3, 6-45
  - fpstcw function*, 6-3, 6-51
  - fpstsw function*, 6-3, 6-58
- Alignment, 2-31
- as control. *See* absstack control
- ASM196 applications, 1-5
- Attribute field, 2-34, 2-35
- Audience description, 1-6

# B

- Base address, 2-4, 2-31
- BASED attribute, 2-34
- Batch files, 2-39
- Binary operations, 6-5
  - fpadd function*, 6-5, 6-9
  - fpcompq function*, 6-5, 6-14
  - fpcomps function*, 6-5, 6-14
  - fpdiv function*, 6-5, 6-16
  - fpmul function*, 6-5, 6-31
  - fprem function*, 6-5, 6-36
  - fpssub function*, 6-5, 6-60
- Binary point, 5-4
- Binary scientific notation, 5-3
- Bit mask, 7-3
- bottomup control, 2-48
- bu control. *See* bottomup control
- Built-in variables, 1-5, 5-13
  - Control word*, 1-5, 5-13, 5-14
  - Status word*, 1-5, 5-13, 5-16

# C

- c command, 4-7
- C196 applications, 1-5
- C196LIB environment variable, 2-27
- Calling graph, 2-53, 2-91
- Calls relationship. *See* regoverlay control
- case control, 2-49
- Case sensitivity, 2-49, 4-4
- case statements, 2-17
- CCB (Chip Configuration Byte), 2-17
- CCR (Chip Configuration Register), 2-17
- ch control. *See* code2high control
- Code segment, 1-4, 2-11, 9-15
  - Type*, 2-34, 2-35, 2-61
- code2high control, 2-51
- Command file, 2-41
- Comparison, 6-14
- Compatible mode, 5-10, 5-19
- compatible mode, 2-65
- Constants, Loading, 6-29
- Control file, 2-25
  - Appending*, 2-25
- Control Variables, 5-13
- Control word, 1-5, 5-14, 6-51, 7-5
  - Default value*, 6-19
  - Default values*, 5-15
  - Loading values*, 5-15, 6-22
- Controlling FPAL96, 5-13
- Controls
  - absstack*, 2-46
  - bottomup*, 2-48
  - case*, 2-49
  - code2high*, 2-51
  - dataoverlay*, 2-52
  - farcode*, 2-71
  - farconst*, 2-72
  - fardata*, 2-73
  - heap*, 2-56
  - ignoreabs*, 2-57

- inittable*, 2-58
- inst*, 2-59
- ixref*, 2-61
- limit\_bitno*, 2-62
- Linking*, 2-45
- list*, 2-63
- Listing*, 2-45
- Locating*, 2-45
- model*, 2-65
- name*, 2-69
- nearcode*, 2-71
- nearconst*, 2-72
- neardata*, 2-73
- np\_rsvup6*, 2-74
- omf*, 2-75
- pageprint*, 2-76
- pagewidth*, 2-77
- print*, 2-78
- purge*, 2-80
- quietwarns*, 2-82
- ram*, 2-83
- regfirst*, 2-88
- register*, 2-86
- regoverlay*, 2-90
- RL196*, 2-22
- rom*, 2-95
- romcode*, 2-99
- romdata*, 2-102
- searchlib*, 2-105
- sfr*, 2-106
- stacksize*, 2-107
- typecheck*, 2-109
- uniquemods*, 2-111
- warning*, 2-112
- windowsize*, 2-113
- Conventions, 1-8, 5-12
  - Naming*, 5-12
  - Parameter passing*, 5-12
- Conversions
  - Controlling*, 5-15
  - Decimal and Floating point*, 6-24, 6-53

*Floating point format*, 6-49  
*Long Integer and Floating point*,  
 6-27, 6-56  
*Long Unsigned Integer and Floating  
 point*, 6-27, 6-56  
 Creating libraries, 1-4  
 Cross-reference, 2-61  
*Listing*, 1-4  
*See also ixref control*  
 cs control. *See* case control  
 Customer comments, 1-8  
 Customer service hotline, 1-8

## D

d command, 4-8  
 Data representation, 5-1  
 Data segment, Types, 2-34, 2-35, 2-61  
*entry*, 2-34  
 Data segments, Types  
*array*, 2-34  
*bit*, 2-34  
*byte*, 2-34  
*entry*, 2-34  
*enum*, 2-34  
*farptr*, 2-34  
*fpl\_proc*, 2-34  
*integer*, 2-34  
*label*, 2-34  
*list*, 2-34  
*long*, 2-34  
*longint*, 2-34  
*null*, 2-34  
*pointer*, 2-34  
*procedure*, 2-34  
*ptr*, 2-34  
*real*, 2-34  
*scalar*, 2-34  
*sgn\_int*, 2-34  
*shortint*, 2-34  
*structure*, 2-34  
*union*, 2-34

*unsgn\_int*, 2-34  
*vpl\_proc*, 2-34  
*whole*, 2-34  
*word*, 2-34  
*wsr\_ptr*, 2-34  
 dataoverlay control, 2-52  
 debug control, 2-37  
 Debugging, 2-37  
*with ICE-196PC*, 2-107  
 Decimals, 5-11  
*Declaration of*, 5-11  
*Exponent*, 5-11  
*Mantissa*, 5-11  
 Declaring functions, 5-18  
 Definition records, 2-5  
*External*, 2-5  
*Public*, 2-5  
 Denormal exception, 5-9, 7-8  
 Denormalization, 7-7  
 Denormalized floating point numbers,  
 5-4  
 Division, 6-16  
 do control. *See* dataoverlay control  
 Double data type, 5-5  
 Dummy file, 2-15  
 Dynamic memory allocation, 2-15,  
 2-46  
 Dynamic segment type, 2-34, 2-35

## E

else, 8-6  
 endif, 8-6  
 environment variable  
*C196LIB*, 2-27  
*HOME*, 8-5  
 Error byte, 5-13, 5-17, 7-7  
*Clearing*, 6-12  
*Exception flags*, 5-17  
 Error messages, 2-36, 9-1, 9-4  
*Fatal*, 9-3  
*LIB196*, 9-23

*Location*, 2-36  
*OH196*, 9-22  
*RL196*, 9-14  
*Warnings*, 9-3  
 Example routines, 5-18, 5-21  
*fpadd function*, 5-25  
*fpldddec function*, 5-22  
*fpst function*, 5-24  
 Exception flags, 5-17  
 Exception handler, 5-9  
     *Creating an*, 7-8  
     *Default*, 5-14, 6-19  
     *Setting*, 5-14, 6-45  
     *Tasks*, 7-8  
     *User-defined*, 5-14  
     *Using floating-point functions*, 7-10  
 Exception handling, 7-1  
     *Debugging with operation codes*, 7-4  
 Exceptions, 7-1  
     *Denormal*, 5-9, 7-1, 7-8  
     *Invalid Operation*, 7-1, 7-5  
     *Invalid-operation exception*, 5-9  
     *Operation Codes*, 7-3, 7-4  
     *Overflow*, 7-1, 7-6  
     *Precision*, 7-1, 7-7  
     *Processing*, 5-17  
     *Underflow*, 7-1, 7-7  
     *Zero divide*, 7-1, 7-6  
 Executable files. *See* absolute object file  
 Exponent, 5-11  
 Exponent field, 5-3  
 Extended mode, 5-10, 5-19  
 extended mode, 2-65  
 Extensions, 1-7  
 External data formats, 5-1  
     *Decimals*, 5-1, 5-11  
     *Integers*, 5-1, 5-11  
     *Real floating point numbers*, 5-1  
 External references, 1-4  
 extrn directive, 5-19

## F

farcode control, 2-71  
 farconst control, 2-72  
 fardata control, 2-73  
 Fatal error messages, RL196, 9-4  
 fc control. *See* farcode control  
 fd control. *See* fardata control  
 Filename extensions, 1-7  
 Finite nonzero number, 7-6  
 First fit/decreasing size algorithm, 2-12  
 fk control. *See* farconst control  
 Float data type, 5-5  
 Floating point number, Format  
     *Exponent field*, 5-3  
     *Fraction field*, 5-3  
     *Sign field*, 5-3  
 Floating point number format, 5-3  
     *Fraction field, Format*, 5-9  
     *Relationship between exponent and fraction*, 5-5  
 Floating point numbers, 5-3  
     *Addition*, 6-9  
     *Comparison*, 6-14  
     *Denormalized*, 5-8  
     *Division*, 6-16  
     *Examples*, 5-3  
     *Multiplication*, 6-31  
     *Normalized*, 5-8  
     *Range*, 5-5  
     *Remainder*, 6-36  
     *Square root*, 6-47  
     *Subtraction*, 6-60  
 Floating-point accumulator. *See* FPACC accumulator  
 fpabs function, 6-4, 6-7  
 FPACC, Setting to negative, 6-34  
 FPACC accumulator, 1-5, 5-13, 6-3, 7-8  
     *Loading*, 6-20

*Setting to positive*, 6-7  
 fpadd function, 6-5, 6-9  
 FPAL96 library  
     *File*, 1-6  
     *Initializing*, 5-20, 6-19  
     *Linking*, 5-20  
     *Operations*, 6-1  
 FPAL96 library files, fpal96.lib, 5-19  
 fpcleb function, 5-18, 6-3, 6-12  
 fpcompq function, 5-13, 5-16, 6-5, 6-14  
 fpcomps function, 5-13, 5-16, 6-5, 6-14  
 fpcomps function, 7-5  
 fpdiv function, 6-5, 6-16  
 fpinit function, 5-18, 5-20, 6-3, 6-19  
 fpld function, 6-3, 6-20  
 fpld1 function, 6-3, 6-29  
 fpldcw function, 5-15, 6-3, 6-22  
 fplddc function, 6-3, 6-24  
 fpldint function, 6-3, 6-27  
 fplduint function, 6-3, 6-27  
 fpldz function, 6-3, 6-29  
 fpmul function, 6-5, 6-31  
 fpneg function, 6-4, 6-34  
 fprem function, 6-5, 6-36  
 fprndint function, 6-4, 6-39  
 fprstor function, 5-18, 6-3, 6-41, 7-10  
 fpsave function, 6-3, 6-43, 7-10  
 fpseteh function, 5-14, 6-3, 6-45, 7-5  
 fpsqrt function, 6-4, 6-47  
 fpst function, 6-4, 6-49  
 fpstcw function, 6-3, 6-51  
 fpstdec function, 6-4, 6-53  
 fpstint function, 6-4, 6-56  
 fpstsw function, 5-17, 6-3, 6-58  
 fpstuint function, 6-4, 6-56  
 fpsub function, 6-5, 6-60  
 Fraction field, 5-3  
     *Format*  
         *Address field*, 5-9  
         *Exception field*, 5-9

Functions, Declaration, 5-18  
     *in ASM196*, 5-18  
     *in C196*, 5-19

## G

Global register variables, 2-21  
 Global symbols  
     *External*, 2-5  
     *Public*, 2-5  
 glossary, A-1

## H

he control. *See* heap control  
 heap control, 2-15, 2-56  
 heap space, 2-15, 2-56  
 hex file, 3-4  
 High code segment, 2-68  
 HOME, 8-5

## I

ia control. *See* ignoreabs control  
 IEEE Standard, 1-5  
     *Format supported*, 1-5  
 ifdef, 8-6  
 ifndef, 8-6  
 ignoreabs control, 2-57  
 in control. *See* inst control  
 Include file, 5-18, 5-19  
 Incremental linking, 2-4, 2-15  
 Infinity operands, Using, 5-7  
 Initialization table, 2-58  
 Initialize variables, 2-9  
 Initializing FPAL96, 5-20, 6-19, 7-10  
 inittable control, 2-58  
 inst control, 2-59



INST pin, 2-16  
*Behavior*, 2-17  
*Chip Configuration Byte*, 2-17  
*Chip Configuration Register*, 2-17  
*Interrupt vector table*, 2-17  
*Opcode fetch*, 2-17  
*Program constants and variables*, 2-17  
*Vector tables*, 2-17  
*Hardware development guidelines*, 2-18  
*Overlapping memory scheme*, 2-17  
*RL196 invocation example*, 2-19  
 Instruction pin. *See* INST pin  
 Integers, 5-11  
 Intermodule cross-reference listing, 2-35, 2-61  
*See also ixref control*  
 Interval arithmetic, Implementing, 5-16  
 Invalid-operation exception, 5-9, 6-34, 7-5  
 Invocation line  
*LIB196*, 4-3  
*OH196*, 3-3  
*RL196*, 2-22  
 invocation mk196, 8-3  
 it control. *See* inittable control  
 ix control. *See* ixref control  
 ixref control, 2-61

## L

l command, 4-10  
 lb control. *See* limit\_bitno control  
 li control. *See* list control  
*LIB196*, Character set, 4-4  
*LIB196* commands, 4-4  
*LIB196* error messages, 9-23  
*LIB196* invocation line, 4-3  
*Interactive mode*, 1-4  
*Non-interactive mode*, 1-4  
*LIB196* library manager, 1-4, 4-1

*LIB196* options, 4-3  
 Libraries, 2-26  
 Library files, 1-4  
*Search path*, 2-27  
 limit\_bitno control, 2-62  
 Line numbers, 2-64, 2-81  
 lines, 2-63, 2-80  
*See also list control; purge control*  
 Link summary, 2-30  
 Link summary information, 2-31  
 Linkage, 1-4  
 Linking the FPAL96 library, 5-18, 5-20  
 list control, 2-31, 2-32, 2-63  
 ln. *See* lines  
 Load operations, 6-3  
*fpld function*, 6-3, 6-20  
*fpld1 function*, 6-3, 6-29  
*fpldddec function*, 6-3, 6-24  
*fpldint function*, 6-3, 6-27  
*fplduint function*, 6-3, 6-27  
*fpldz function*, 6-3, 6-29  
 Loading constants, 6-29  
 Loading the FPACC, 6-20  
 Local register variables, 2-21  
 Local symbols, 2-64, 2-81  
 Long double data type, 5-5  
 lp command, 4-10

## M

Main module, 2-4, 2-26  
 maintain programs, 8-3  
 Make Utility mk196, 2-38, 8-1  
 Mantissa, 5-11  
 Map file, Creating a, 2-78  
 Matching segment type, 2-5, 2-109  
 Matching symbol type, 2-5, 2-109  
 md control. *See* model control  
 Memory allocation, 2-11, 2-88  
 Memory segment, 2-11  
*Code*, 2-11  
*Data*, 2-11

- Overlay*, 2-11
- Register*, 2-11
- MEMORY symbol, 2-15, 9-3
- Messages, 9-1
- mk196
  - .DEFAULT target*, 8-11
  - .DONE target*, 8-11
  - .IGNORE target*, 8-11
  - .INIT target*, 8-11
  - .PRECIOUS target*, 8-12
  - .SILENT target*, 8-12
  - .SUFFIXES target*, 8-12
  - comment lines*, 8-6
  - conditional processing*, 8-6
  - diagnostics*, 8-15
  - dry run*, 8-4
  - example*, 8-14
  - exist function*, 8-10
  - export line*, 8-6
  - functions*, 8-9
  - ifdef*, 8-6
  - implicit rules*, 8-13
  - include line*, 8-6
  - macro definition*, 8-5
  - macro MAKE*, 8-7
  - macro MAKEFLAGS*, 8-8
  - macro PRODDIR*, 8-8
  - macro SHELLCMD*, 8-8
  - macros*, 8-7
  - Makefile*, 8-15
  - makefile*, 8-5, 8-15
  - match function*, 8-9
  - mk196.mk*, 8-15
  - nexist function*, 8-10
  - options*, 8-3
  - protect function*, 8-10
  - question mode*, 8-4
  - rules*, 8-12
  - separate function*, 8-9
  - silent mode*, 8-4
  - special macros*, 8-7
  - special targets*, 8-11
  - targets*, 8-10

- touch target file*, 8-4
- mk196 make utility, 8-1
- model control, 2-65
- Modes
  - Compatible*, 5-10, 5-19
  - Extended*, 5-10, 5-19
- Modules, Unique, 2-111
- Most-significant digit, 5-4
- Multiplication, 6-31

## N

- na control. *See* name control
- name control, 2-69
- Naming conventions, 5-12
- NaNs, 5-9, 6-14
  - Quiet NaNs*, 5-9
  - Signalling NaNs*, 5-9
- nc control. *See* nearcode control
- nd control. *See* neardata control
- nearcode control, 2-71
- nearconst control, 2-72
- neardata control, 2-73
- nk control. *See* nearconst control
- noabsstack control, 2-9, 2-46
- noas control. *See* noabsstack control
- nobottomup control, 2-48
- nobu control. *See* nobottomup control
- nocase control, 2-49
- noch control. *See* nocode2high control
- nocode2high control, 2-51
- nocs control. *See* nocase control
- nodataoverlay control, 2-52
- nodo control. *See* nodataoverlay control
- nohe control. *See* noheap control
- noheap control, 2-56
- noia control. *See* noignoreabs control
- noignoreabs control, 2-57
- noin control. *See* noinst control
- noinitable control, 2-58

noinst control, 2-59  
 noit control. *See* noinittable control  
 noix control. *See* noixref control  
 noixref control, 2-61  
 nolb control. *See* nolimit\_bitno control  
 noli control. *See* nolist control  
 nolimit\_bitno control, 2-62  
 nolist control, 2-63  
 nonp\_rsvup6 control, 2-74  
 noov control. *See* noregoverlay control  
 nopr control. *See* noprint control  
 noprint control, 2-78  
 nopu control. *See* nopurge control  
 nopurge control, 2-37, 2-80  
 noquietwarns control, 2-82  
 noqw control, 2-82  
 noregfirst control, 2-88  
 noregoverlay control, 2-90  
 norf control. *See* notypecheck control  
 Normalized floating point, 5-3, 5-8  
 nosfr control, 2-106  
 Not-a-Number. *See* NaNs  
 notc control. *See* notypecheck control  
 notypecheck control, 2-5, 2-109  
 noun control. *See* nouniquemods control  
 nouniquemods control, 2-111  
 nowa control. *See* notypecheck control  
 nowarning control, 2-112  
 np\_rsvup6 control, 2-74  
 null  
     *Segment type, 2-34, 2-61*  
     *Symbol type, 2-34, 2-61*

## O

Object library file, 2-26, 2-28  
 OH object-to-hexadecimal converter,  
     1-7  
 OH196 converter, Error messages, 9-22

OH196 object-to-hexadecimal  
     converter, 3-1  
     *Invocation line, 3-3*  
     *Output file, 3-4*  
     *Record type, 3-5*  
 omf control, 2-10, 2-75  
 OMF96  
     *combining formats, 2-10*  
     *global initialization, 2-10*  
     *version 3.0 limitations, 2-11*  
 Opcode fetch, 2-16  
 Operations  
     *Administrative, 1-5, 6-1*  
     *Load and store, 1-5, 6-1*  
     *Unary and binary, 1-5, 6-1*  
 Options  
     *LIB196, 4-3*  
     *mk196, 8-3*  
     *RL196, 2-23*  
     *Table, 2-23*  
     *Turning off/on, 2-23*  
 Order of allocation, 2-11  
 Ordinary object file, 2-22, 2-27, 4-6  
 Output object module, Naming, 2-69  
 ov control. *See* regoverlay control  
 Overflow exception, 7-6  
     *Rounding results, 7-6*  
 Overlapping memory scheme, 2-17  
 Overlapping ROM and RAM, 2-16,  
     2-59  
 Overlay segment, 2-14, 2-22, 2-52,  
     2-90  
     *Type, 2-34, 2-35, 2-61*  
 Overlaying segments, 2-52, 2-90  
 Overview, 1-1

## P

pageprint control, 2-76  
 pagewidth control, 2-77

Parameter passing conventions, 5-12  
 Performing fixups, 2-9  
 pl. *See* public  
 PLMREG register, 5-12, 5-19  
 pp control. *See* pageprint control  
 pr control. *See* print control  
 Precision exception, 7-7  
 print control, 2-78  
 Print file, 2-63  
 PROM programmer, 1-7  
   *Loading*, 1-7  
 pu control. *See* purge control  
 Public symbol, 2-5, 2-16, 2-80  
 public, 2-63, 2-80  
   *See also list control; purge control*  
 Publiconly object file, 1-4, 2-26, 2-29  
 purge control, 2-80  
   *Conserving space*, 2-38  
 pw control. *See* pagewidth control

## Q

Quasi-absolute object file, 2-4, 2-38, 2-46  
   *Creating an*, 2-38  
 Quiet NaNs, 5-9  
   *Address field*, 5-9  
   *Exception field*, 5-9  
   *Operands*, 5-10  
 quietwarns control, 2-82  
 qw control, 2-82

## R

r command, 4-11  
 ra control. *See* ram control  
 ram control, 2-11, 2-12, 2-83  
 RAM memory, 1-4, 2-11, 2-15, 9-6, 9-7  
   *Designating*, 2-83

rc control. *See* rom control  
 rd control. *See* rom control  
 Real data type, 5-5  
 reconstruct programs, 8-3  
 Reg. *See* Register segment  
 regfirst control, 2-11, 2-88  
 Register, Allocation, 2-20  
   *Overlay segment*, 2-20  
   *Register segment*, 2-20  
 Register overlaying, 2-14  
 Register segment, 1-4, 2-11, 2-22, 2-86  
   *Type*, 2-34, 2-35, 2-61  
 Registers  
   *16-bit direct addressing mode*, 2-20  
   *8-bit direct addressing mode*, 2-20  
   *Range*, 2-86  
 registers control, 2-11, 2-22, 2-86  
 regoverlay control, 2-11, 2-12, 2-14, 2-88, 2-90, 9-21  
 Relocatable segment, 2-4, 2-12  
 Relocatable symbol, 2-9  
 Relocation, 1-4  
 Remainder, 6-36  
 Resolving external references, 2-5  
 Restoring FPAL96 status, 6-41  
 return statement, 7-10  
 rf control. *See* typecheck control  
 rg control. *See* registers control  
 RL196 controls, 2-22, 2-43  
   *Linking*, 2-43  
   *Listing*, 2-43  
   *Locating*, 2-43  
 RL196 error messages, 9-3, 9-14  
   *Fatal*, 9-4  
   *Warning messages*, 9-18  
 RL196 input modules  
   *Object library file*, 2-26, 2-28  
   *Conditional processing*, 2-28  
   *Syntax*, 2-28  
   *Ordinary object file*, 2-27  
   *Publiconly object file*, 2-26, 2-29

*Selecting*, 2-26  
 RL196 invocation line, 2-22  
*Input list*, 2-22  
*Object library file*, 2-28  
*Ordinary object file*, 2-27  
*Publiconly object file*, 2-29

RL196 linker, 1-5, 5-20  
*Invocation line*, 5-20

RL196 linker and locator, 2-1  
*Major functions*, 2-3

RL196 linker/locator, 1-4

RL196 options, 2-23

-?, 2-25  
 -as, 2-46  
 -bu, 2-48  
 -case, 2-49  
 -cb, 2-51  
 -f, 2-25  
 -be, 2-56  
 -ia, 2-57  
 -in, 2-59  
 -ix, 2-61  
 -L, 2-105  
 -lb, 2-62  
 -M, 2-78  
 -md, 2-65  
 -omf, 2-75  
 -pw, 2-77  
 -QW, 2-82  
 -rf, 2-88  
 -S, 2-106  
 -ss, 2-107  
 -tc, 2-109  
 -um, 2-111  
 -V, 2-25  
 -W, 2-112  
 -ws, 2-113

RL196 outputs

*Console display*, 9-3  
*Object file*, 2-30  
*Print file*, 2-30

ro control. *See* rom control

rom control, 2-11, 2-12, 2-95

ROM memory, 1-4, 2-11, 9-6, 9-7

*Designating*, 2-95, 2-99, 2-102

ROM/RAM overlapping, 2-59

romcode control, 2-11, 2-12, 2-99

romdata control, 2-11, 2-12, 2-102

Round to integer, 6-39

Rounding Modes, 5-14

Rounding modes, 5-15

*Default*, 5-15

*Directed*, 5-16

*Most accurate*, 5-15

*Round down*, 5-15

*Round to the nearest*, 5-15

*Round up*, 5-15

*Truncate*, 5-15

## S

Saving FPAL96 status, 6-43

sb. *See* symbols

searchlib control, 2-105

Segment

*Map*, 1-4, 2-31

*Option*, 2-31

*Type*, 2-109

*Type matching*, 2-109

*Types*, 2-34, 2-35

Segment placement, 2-4

Segment type matching, 2-6

Segment types, 2-61

*farcode*, 2-34, 2-68

*farconst*, 2-34

*fardata*, 2-34

*highcode*, 2-51, 2-68

segments, 2-63, 2-80

*See also list control; purge control*

Selecting FPAL96 library, 5-19

sfr control, 2-106

Sign field, 5-3

Sign-off message, 9-4

Sign-on message, 9-3

Signalling NaNs, 5-9  
 sl control. *See* searchlib control  
 sm. *See* segments  
 Software development process, 1-3  
 Special floating point numbers, 5-5  
     *Denormal*, 5-5  
     *Denormalized Numbers*, 5-8  
     *Infinity*, 5-5, 5-7  
     *NaN*, 5-5, 5-9  
     *Zero*, 5-5  
 Square root, 6-47  
 ss control. *See* stacksize control  
 st. *See* stack segment  
 Stack  
     *\_BOTTOM\_OF\_STACK\_*, 2-13  
     *\_TOP\_OF\_STACK\_*, 2-13  
 Stack overflow, 2-13  
 Stack segment, 2-12  
     *See also absstack control; stacksize control*  
     *Size*, 2-107  
     *Type*, 2-34, 2-35, 2-61  
 stack segment, 2-26  
 stacksize control, 2-107, 9-19  
 static storage type, 2-22  
 Status word, 1-5, 5-16, 6-14, 6-58, 7-5  
     *Default value*, 6-19  
     *Error byte*, 5-17  
     *Clearing*, 6-12  
     *Format*, 5-16  
     *STAT field*, 5-16  
     *Processing exceptions using*, 5-17  
 Store operations, 6-4  
     *fpst function*, 6-4, 6-49  
     *fpstdec function*, 6-4, 6-53  
     *fpstint function*, 6-4, 6-56  
     *fpstuint function*, 6-4, 6-56  
 Suffix rules, 1-7  
 Summary of options, 2-23  
 switch statements, 2-17  
 Symbol  
     *?MEMORY\_SIZE*, 2-15  
     *\_HEAP\_END\_*, 2-15

*\_HEAP\_START\_*, 2-15  
     *MEMORY*, 2-15  
 Symbol base, 2-34  
 Symbol references, 2-5  
 Symbol table, 1-4, 2-32  
     *Value*, 2-32  
 Symbol type matching, 2-6  
 Symbol types, 2-61  
 symbols, 2-63, 2-80  
     *See also list control; purge control*

## T

tc control. *See* typecheck control  
 Two's complement, 5-11  
 Type checking, 2-5, 2-109  
 Type matching  
     *Segment*, 2-6  
     *Symbol*, 2-6  
 Type mismatch, 2-8  
     *Output of RL196*, 2-8  
     *Warning*, 2-8  
 typecheck control, 2-5, 2-109

## U

um control. *See* uniuquemods control  
 Unallocated segments, 2-12  
 Unary operations, 6-4  
     *fpabs function*, 6-4, 6-7  
     *fpneg function*, 6-4, 6-34  
     *fprndint function*, 6-4, 6-39  
     *fpsqrt function*, 6-4, 6-47  
 Underflow exception, 7-7  
 Uninitialized variables, Detecting, 5-9  
 uniuquemods control, 2-111  
 Unresolved external symbols, 2-5  
 Unresolved references, 1-4  
 update programs, 8-3

## Utilities

*LIB196*, 4-1*mk196*, 8-1*OH196*, 3-1*RL196*, 2-22**V**

Value field, 2-34

Variable initialization, 2-9

Vector tables, 2-17

Vertical windows, 2-20, 2-86, 2-113

*Register allocation scheme*, 2-21*Size*, 2-113*Using*, 2-20VWindows. *See* Vertical windows**W**wa control. *See* typecheck control

warning control, 2-112

Warning messages, 2-108

*RL196*, 9-18*Set to non-zero*, 2-112

## Windows

*Horizontal*, 2-20*Mapping*, 2-20*Vertical*, 2-20

window size control, 2-22, 2-113

ws control. *See* window size control

WSR management code, 2-21

**X**

x command, 4-9

**Z**Zero operands, *Using*, 5-5

Zero-divide exception, 7-6