



Development Solutions

MCS®-96 Utilities User's Guide

MCS®-96

Utilities User's Guide

Typographical Conventions

This manual uses the following conventions:

<i>This type style</i>	indicates an element of syntax, a reserved word, a keyword, a filename, computer output, or part of a program example. The text appears in lowercase unless uppercase is significant. l is lowercase letter L in examples. 1 is the number 1 in examples. O is the uppercase O in examples. Ø is the number 0 in examples.
<i>This type style</i>	indicates the exact characters you type as input.
<i>This type style</i>	indicates a placeholder for an identifier, an expression, a string, a symbol, or a value. Substitute one of these items for the placeholder.

MCS[®]-96 UTILITIES USER'S GUIDE

Order Number: 122355-004

REV.	REVISION HISTORY	DATE
-001	Original Issue.	07/85
-002	Revised to conform to software upgrade.	05/88
-003	Added vertical windowing support information.	12/90
-004	Revised for clarity and completeness.	11/91

In the United States, additional copies of this manual or other Intel literature may be obtained by writing:

Literature Distribution Center
Intel Corporation
P.O. Box 7641
Mt. Prospect, IL 60056-7641

Or you can call the following toll-free number:

1-800-548-4725

In locations outside the United States, obtain additional copies of Intel documentation by contacting your local Intel sales office. For your convenience, international sales office addresses are printed on the last page of this document.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's Software License Agreement, or in the case of software delivered to the government, in accordance with the software license agreement as defined in FAR 52.227-7013.

No part of this document may be copied or reproduced in any form or by any means without prior written consent of Intel Corporation.

Intel Corporation retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products. (Registered trademarks are followed by a superscripted ®.)

376	i486	Intel386	LANSelect®	READY-LAN
Above	i487	Intel387	LANShell®	Reference Point®
ActionMedia	i750®	Intel486	LANsight	RMX/80
BITBUS	i860	Intel487	LANSpace®	SatisFAXtion
Code Builder	i960	intel inside	LANSpool®	SnapIn 386
DeskWare	†	Intellec®	MAPNET	StorageBroker
DVI®	i®	iPSC®	Matched	SugarCube
ETOX	ICE	iRMX®	MCS®	SUPERCHARGER
FaxBACK	iLBX	iSBC®	Media Mail	The Computer Inside
Grand Challenge	Inboard	iSBX	NetPort	Token Express
i287	Intel®	iWARP	OpenNET	Visual Edge
i386	Intel287	LANPrint®	PRO750	WYPIWYF
i387				

1. Introduction

2. Objectives

3. Methodology

4. Results

5. Discussion

6. Conclusion

7. References

8. Appendix

Contents

Getting Started	vii
-----------------------	-----

Chapter 1 Overview

1.1	MCS [®] -96 Utilities and the Development Process	1-1
1.1.1	Relocation and Linkage	1-2
1.1.2	Use of Libraries	1-3
1.1.3	ROM and PROM Versions	1-4
1.1.4	Keeping Track of Files	1-5
1.2	About This Manual	1-5
1.3	Related Publications	1-5
1.4	Conventions	1-6
1.5	Problem Reporting	1-6
1.6	Trademarks	1-7

Chapter 2 The RL96 Linker

2.1	Resolving External References	2-2
2.1.1	Type Checking	2-3
2.1.2	Performing Fix-ups	2-7
2.2	Memory Allocation	2-8
2.2.1	Register Overlaying	2-9
2.2.2	Providing Means for User's Dynamic Memory Allocation	2-9
2.2.3	Overlapping ROM and RAM Memory Using the INST Pin	2-11
2.2.4	Support for Vertical Windows	2-17
2.3	Invoking RL96	2-19
2.3.1	Input List	2-20
2.3.2	Output Files	2-23
2.4	Invocation Examples	2-32

2.5	Automatically Invoking Multiple Commands	2-33
2.5.1	Using DOS Batch Files	2-33
2.5.2	Using DOS Command Files	2-35
2.6	RL96 Controls	2-37
	/f	2-40
	absstack	2-42
	inst	2-43
	ixref	2-44
	list	2-45
	name	2-47
	pagewidth	2-49
	print	2-50
	purge	2-51
	ram	2-53
	registers	2-56
	regoverlay	2-58
	rom	2-63
	stacksize	2-66
	typecheck	2-68
	window size	2-69

Chapter 3 The OH Converter

3.1	Invocation Syntax	3-1
3.2	Examples	3-1
3.3	Output File	3-2

Chapter 4 The LIB96 Librarian

4.1	Invoking LIB96	4-1
4.1.1	Noninteractive Mode	4-1
4.1.2	Interactive Mode	4-2
4.1.3	Character Set	4-2
4.2	LIB96 Commands	4-3
4.3	Command Descriptions	4-3
	add	4-4
	create	4-6
	delete	4-7
	exit	4-8
	extract	4-9

help	4-10
list	4-11
replace	4-12

Chapter 5 Messages and Error Recovery

5.1	RL96 Messages	5-1
5.1.1	Console Messages	5-2
5.1.2	Fatal Errors	5-3
5.1.3	Error Messages	5-8
5.1.4	Warnings	5-11
5.2	OH Error Messages	5-16
5.3	LIB96 Error Messages	5-16

Glossary

Index

Service Information	Inside Back Cover
---------------------------	-------------------

Figures

1-1	The MCS®-96 Application Development Process	1-2
1-2	Library Linkage by RL96	1-4
2-1	112-Kilobyte Overlapping Code and Data Memory Map	2-15
2-2	80C196KC Register Allocation Scheme	2-18
2-3	Sample Link Summary	2-25
2-4	Sample Symbol Table	2-26
2-5	Sample Intermodule Cross-reference	2-30

Tables

1-1	Technical Publications Library	1-6
2-1	Segment Type Matching	2-4
2-2	ASM96 - iC-96 Symbol Type Matching	2-5
2-3	RL96 Controls	2-38
4-1	LIB96 Commands	4-3



Getting Started

This page of the *MCS®-96 Utilities User's Guide* tells you where to find information needed to use the RL96, OH, and LIB96 utilities. For information on this manual and related publications, see Section 1.3.

Installing the Utilities

To install the MCS®-96 utilities, see the installation section in the *ASM96 Assembler User's Guide for DOS Systems* or the *iC-96 Compiler User's Guide for DOS Systems*. That section also explains the system requirements, environment variable settings, and directory structure to set up your system for the translator and utilities.

Making Efficient Use of RL96

To understand how RL96 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.

Running the RL96 Linker

See Chapter 2 to learn how to invoke the linker from the prompt or a batch file and how to use the linking and locating controls. This chapter includes a controls reference section that explains each control in detail.

Determining the Cause of Errors

If errors occur during linking, you can find a list of errors and possible fixes in Chapter 5.

Converting OMF Object Files to Hexadecimal Format

Third-party vendors' PROM programmers do not always accept the MCS-96 Absolute Module Format. In this case, you can run the absolute object file created by RL96 through the OH utility which produces a file in hexadecimal format. Read Chapter 3 for instructions on how to run OH.

Creating Libraries with the LIB96 Library Manager

To learn how to create your own library, read Chapter 4. This chapter also describes procedures for maintaining the libraries using the LIB96 commands.

Comments, Problems, and Service

If you need service or assistance with any of the utilities, see the inside back cover for instructions.

Intel values your input. Please suggest changes to the manual using the white reader-comment card at the back of the manual.

Please suggest changes and report problems with the utilities using the yellow customer-comment/problem-report card at the back of this manual.

Contents

Overview

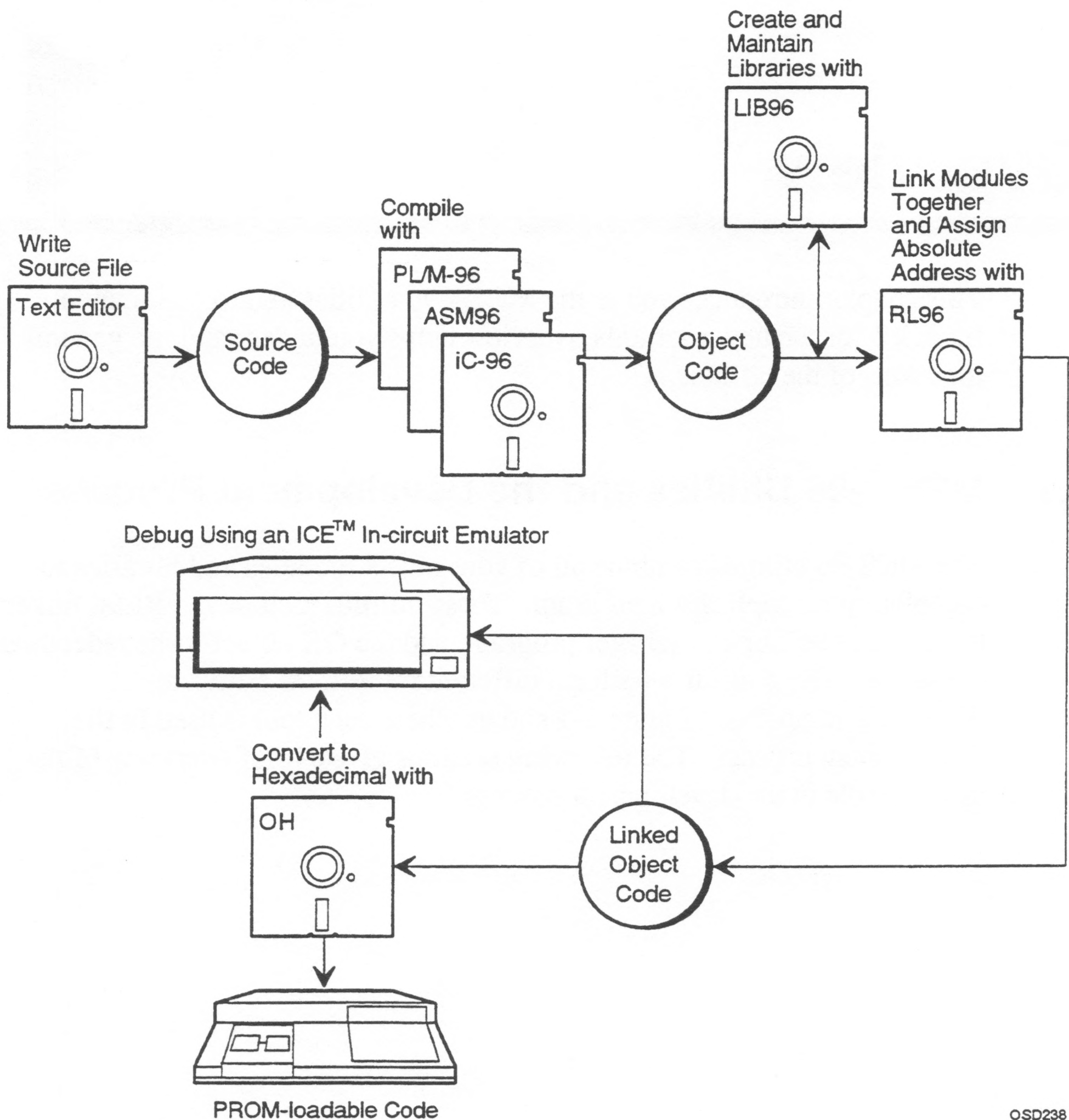
1.1	MCS [®] -96 Utilities and the Development Process	1-1
1.1.1	Relocation and Linkage	1-2
1.1.2	Use of Libraries	1-3
1.1.3	ROM and PROM Versions	1-4
1.1.4	Keeping Track of Files	1-5
1.2	About This Manual	1-5
1.3	Related Publications	1-5
1.4	Conventions	1-6
1.5	Problem Reporting	1-6
1.6	Trademarks	1-6

Overview

This chapter introduces you to the MCS[®]-96 utilities and to this manual. Intended for the new user, this overview helps you understand the general functions of the utilities.

1.1 MCS[®]-96 Utilities and the Development Process

The MCS-96 utilities combine all of your object modules and libraries to complete your application program. These utilities include the RL96 linker locator, LIB96 library manager program, and the OH 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.



OSD238

Figure 1-1 The MCS®-96 Application Development Process

1.1.1 Relocation and Linkage

After all of the modules are translated, the RL96 linker/locator processes the object module files. RL96 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

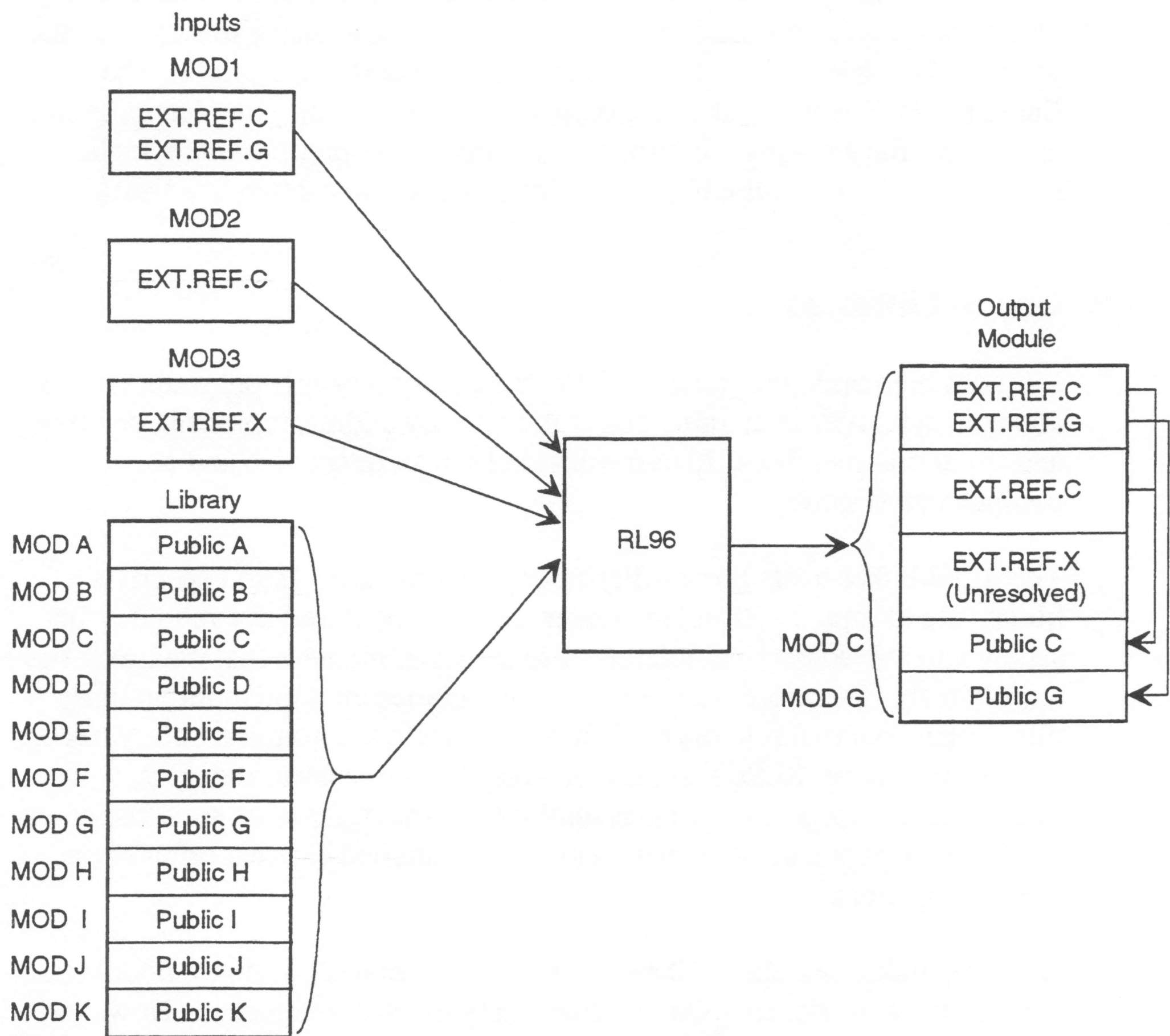
segments (both overlayable and non-overlayable) to a register area. RL96 also resolves all references between modules, possibly using library files and publiconly files. Publiconly files are discussed in Chapter 3. The linker produces both an absolute object module file of the complete program and a print file showing the results of the link/locate process, including a segment map, a symbol table, and an intermodule cross-reference listing.

1.1.2 Use of Libraries

Libraries help build programs. LIB96, the library manager program, creates and maintains files containing object modules by adding, replacing, deleting, and extracting members. LIB96 works both in an interactive and a noninteractive mode.

The RL96 linker treats library files in a special manner. If you specify a library file as input to RL96, 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, RL96 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, RL96 provides the means to unconditionally include a library module even if the library module does not satisfy any external reference. See Section 2.4 for more information on how to select input modules. Figure 1-2 shows RL96 handling a library file in the conditional manner.



OSD986

Figure 1-2 Library Linkage by RL96

1.1.3 ROM and PROM Versions

You can load the absolute object module produced by RL96 into members of the MCS-96 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 OH utility, discussed in Chapter 6, to perform this function. See the *16-Bit Microcontroller Handbook*, listed in Section 1.3, for details on the available microcontroller versions.

1.1.4 Keeping Track of Files

The RL96 linker uses 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 no extension.

1.2 About This Manual

This manual describes the operation of the RL96 linker, OH object-to-hexadecimal converter and the LIB96 library manager program. This manual also provides details on how to use the RL96 controls and the LIB96 library commands. You can also find a format description of the hexadecimal records generated by the OH utility.

To effectively use the MCS-96 utilities, you must be familiar with the 8096 architecture, programming in assembly language or a high-level language, and the software development process.

1.3 Related Publications

This manual is part of the manual sets for ASM96, PL/M-96, and iC-96. The order number for each manual is listed below.

You can order additional publications from Intel's Literature Distribution Center by calling the toll free number listed on the back cover of this manual.

Table 1-1 lists titles and order numbers of publications containing more information on the 8096 architecture, the 8096 instruction set, and the types of 8096 applications. These publications are available from Intel's Product Literature Sales Department.

Table 1-1 Technical Publications Library

Title	Order Number
<i>ASM96 Assembler User's Guide for DOS Systems</i>	122350
<i>iC-96 Compiler User's Guide for DOS Systems</i>	481195
<i>PL/M-96 User's Guide for DOS Systems</i>	481644
<i>8096 Floating-point Library Supplement</i>	122365
<i>16-Bit Microcontroller Handbook</i>	270646
<i>Automotive Products Handbook</i>	270646
<i>Embedded Applications</i>	270648
<i>The 16-bit 8096: Programming, Interfacing, Applications</i> by Ron Katz and Howard Moyet, MTI Publications, 1986	

1.4 Conventions

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

This manual also uses the conventions listed in the Notational Convention found on the back of the front cover of this manual.

1.5 Problem Reporting

If you need assistance with any of the utilities, see the inside back cover of this manual for Intel's Customer Support telephone number.

Intel values your input. You will find a reader response card and three customer comment/problem report forms at the back of this manual. Please return the reader response card with suggestions or corrections for the improvement of this manual. Use the yellow customer comment/problem report form to report problems with the product.

1.6 Trademarks

Intel and MCS are registered trademarks and ICE is a trademark of Intel Corporation.

Contents

The RL96 Linker

2.1	Resolving External References	2-2
2.1.1	Type Checking	2-3
2.1.1.1	Segment Type Matching	2-3
2.1.1.2	Symbol Type Matching	2-4
2.1.1.3	Imposing the More Restricted Type	2-6
2.1.1.4	Mismatched Types	2-7
2.1.2	Performing Fix-ups	2-7
2.2	Memory Allocation	2-8
2.2.1	Register Overlaying	2-9
2.2.2	Providing Means for User's Dynamic Memory Allocation	2-9
2.2.3	Overlapping ROM and RAM Memory Using the INST Pin	2-11
2.2.3.1	INST Pin Behavior	2-11
2.2.3.2	Overlapping Memory Scheme Example	2-14
2.2.3.3	Guidelines for Hardware Development	2-15
2.2.3.4	Guidelines for Software Development	2-16
2.2.3.5	Linker/Locator Example Invocation Line	2-16
2.2.3.6	Summary of RL96 INST Usage	2-17
2.2.4	Support for Vertical Windows	2-17
2.3	Invoking RL96	2-19
2.3.1	Input List	2-20
2.3.1.1	Ordinary Object File	2-20
2.3.1.2	Object Library File	2-21
2.3.1.3	Publiconly Object File	2-22
2.3.2	Output Files	2-23
2.3.2.1	Print File	2-23
2.3.2.2	Output Object File	2-31
2.4	Invocation Examples	2-32

2.5	Automatically Invoking Multiple Commands	2-33
2.5.1	Using DOS Batch Files	2-33
2.5.2	Using DOS Command Files	2-35
2.6	RL96 Controls	2-37
	/f	2-40
	absstack	2-42
	inst	2-43
	ixref	2-44
	list	2-45
	name	2-47
	pagewidth	2-49
	print	2-50
	purge	2-51
	ram	2-53
	registers	2-56
	regoverlay	2-58
	rom	2-63
	stacksize	2-66
	typecheck	2-68
	window size	2-69

The RL96 Linker

The RL96 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.

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 RL96 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. RL96 allocates these segments according to the memory allocation process described in Section 2.2. 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. RL96 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 2.6) control to suppress the stack segment to absolute form. When you specify `noabsstack`, the output of RL96 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 RL96. 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 Intel's in-circuit emulators and PROM programmers.

2.1 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. RL96 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 2.1.1 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, RL96 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 RL96 with some new input modules.

2.1.1 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.

2.1.1.1 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.

Table 2-1 Segment Type Matching

SEG. TYPE	CODE	DATA	REG	OVERLAY	STACK	NULL	DYNAMIC
CODE	M					M	
DATA		M	M	M	M	M	
REGISTER		M	M	M	M	M	
OVERLAY		M	M	M	M	M	
STACK		M	M	M	M	M	
NULL	M	M	M	M	M	M	
DYNAMIC							M

2.1.1.2 Symbol Type Matching

Two ASM96 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 iC-96 or PL/M-96 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:

```
declare xsub (*) word data (402, 345, 126);
```

This declaration defines `xsub` to be an constant array of three, with values of 402, 345, and 126 respectively. See the *PL/M-96 User's Guide for DOS Systems*, listed in Chapter 1, 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 ASM96 global variables and iC-96 global variables. An M in the column indicates a match. Since matching for iC-96 is similar to that for PL/M-96, Table 2-2 also represents matching between PL/M-96 global variables and ASM96 global variables.

Table 2-2 ASM96 - iC-96 Symbol Type Matching

ASM96		BYTE	WORD	LONG	REAL	ENTRY	NULL
iC-96	UNSIGNED CHAR	M					M
	UNSIGNED INT		M				M
	UNSIGNED LONG			M			M
	CHAR						M
	INT						M
	LONG						M
	FLOAT						M
	ARRAY						M
	STRUCT						M
	LABEL						M
	FUNCTION						M

ASM96 has no symbol type that matches the symbol types array and structure of iC-96 and PL/M-96, except for `null`. Therefore, to avoid a mismatch warning, you must explicitly or implicitly attach the symbol type `null` to the symbol in the ASM96 module. For example, your iC-96 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 ASM96:

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

2.1.1.3 Imposing the More Restricted Type

The following example presents a situation in which you must impose the more restricted type: an ASM96 variable of type `byte` matches both PL/M-96 `byte` and `shortint` variables. However, these two PL/M-96 variables, as externals, must not simultaneously match a public ASM96 variable of type `byte` because the PL/M-96 variables do not match one another. Therefore, the idea of imposing the more restricted type is added to the type-checking process.

When you impose the more restricted type for segments, the linker assigns the following segment types:

- Whenever two segment types match and one of them is `null`, the resulting segment type is the non-`null` one, unless both are `null`.
- In any other match, if the two globals are the same, either both `public` or both `external`, the resulting segment type is the segment type of the symbol processed first. If the two globals are different (one `public` and one `external`), the resulting segment type is that of the `public` symbol.

For symbols, the linker follows these rules:

- Matching ASM96 `byte` with ASM96 `null` results in ASM96 `byte`.
- Matching ASM96 `byte` with PL/M-96 `byte` results in PL/M-96 `byte`. Recall that PL/M-96 `byte` is more restricted than ASM96 `byte`. This is true with iC-96 as well.

2.1.1.4 Mismatched Types

The RL96 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 RL96 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.

2.1.2 Performing Fix-ups

The object modules that serve as input to RL96 can contain portions of object code left undefined by the language translator or by RL96. 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 5 for complete list of RL96 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 RL96 to store the correct value at location *n*, the translator must include a fix-up that instructs RL96 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 *iC-96 Compiler User's Guide for DOS Systems*, listed in Chapter 1, and Section 2.2.4 for more information on vertical windowing.

2.2 Memory Allocation

RL96 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` (for code segments) and the `ram` (for data and stack segments) controls. The linker locates register segments (overlayable and non-overlayable) in the internal register memory (1AH-0FFH by default) specified by the `registers` control. Section 2.6 describes the `rom`, `ram` and `registers` controls.

RL96 follows this order of allocation:

1. All absolute segments.
2. Relocatable overlay segments of modules specified by the `regoverlay` control.
3. Relocatable register segments and relocatable overlay segments not yet allocated.
4. Relocatable code segments of modules specified by the `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.

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.

RL96 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.

2.2.1 Register Overlaying

Register overlaying is part of the memory allocation process performed by RL96 (see step 2 of the order of allocation in Section 2.2). Each input module has, at most, one relocatable overlay segment. RL96, 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 2.6.

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.

2.2.2 Providing Means for User's Dynamic Memory Allocation

Following the memory allocation process and providing that you specified the `absstack` control, RL96 supplies two public symbols that your program can use for dynamic memory allocation during program execution. The two public symbols are `MEMORY` and `?MEMORY__SIZE`. RL96 finds the largest free RAM section, and assigns its base address to `MEMORY` and its size, in bytes, to `?MEMORY__SIZE`. These two symbols together specify the exact address space of the free RAM section available to you. You can use this feature

either directly through the two public symbols while coding in ASM96 assembly language, or indirectly while coding in a high-level language, such as C or PL/M.

When using dynamic memory, be aware of these requirements:

- If no free RAM section is found, both `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. 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 `MEMORY` and `?MEMORY__SIZE` appear in module `< Dummy >` and file `< Dummy >`. These two 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 `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.

NOTE

The `absstack` control must be in effect during the final linkage. RL96 automatically supplies the `MEMORY` and `?MEMORY__SIZE` symbols during this time. However, the section indicated by these two symbols is 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.

2.2.3 Overlapping ROM and RAM Memory Using the INST Pin

The addressable memory space on the MCS[®]-96 family of components consists of 64 kilobytes, mostly available for code or data memory. The instruction (INST) pin present in most MCS-96 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 RL96 invocation line.

NOTE

Only the ICE[™]-196KB/HX and ICE-196KC/HX emulators currently support overlapping code and data memory schemes. Both of these emulators include mapping and interrogation commands that support the INST pin.

2.2.3.1 INST Pin Behavior

Consider the following when developing an MCS-96 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 MCS-96 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, normally located in code segments, are read as data and therefore, the INST pin is low. The INST pin is also low during write operations to variables.
- **Program Access to Vector Tables.** The iC-96 switch statements and the PL/M-96 case statements create vector tables to transfer program execution control. The processor reads the vector value as data thus forcing the INST pin low. These vector tables of data are stored in the code memory space, therefore burned into the PROM. For example, the following code:

```
main()
{ register int i, a;

  switch (i)
  { case 1: a = 10;
    case 2: a = 9;
    case 3: a = 8;
    case 4: a = 7;
    case 5: a = 6;
    case 6: a = 5;
    default: a = 0;
  }
}
```

Produces the following vector table:

```
          ; Statement 2
main:
  push    ?OVRBASE
  push    ?OVRBASE+2H
          ; Statement 4
  ld      Tmp0,i
  br      @0003
@0004:
          ; Statement 5
  ldbse   a,#0AH
```



```

@0005:
; Statement      6
ldbse a,#9H
@0006:
; Statement      7
ldbse a,#8H
@0007:
; Statement      8
ldbse a,#7H
@0008:
; Statement      9
ldbse a,#6H
@0009:
; Statement     10
ldbse a,#5H
@000A:
; Statement     11
clr    a
; Statement     12
br     @0002
@0003:
dec    Tmp0
cmp    Tmp0, #5H
bh     @000A
add    Tmp0,Tmp0
ld     Tmp0,@000B[Tmp0]
br     [Tmp0]
@000B: dcw    @0004          /* vector table */
dcw    @0005
dcw    @0006
dcw    @0007
dcw    @0008
dcw    @0009
@0002:
; Statement     13
pop    ?OVRBASE+2H
pop    ?OVRBASE
ret

```

NOTE

The processor reads data from these vector tables, rather than fetching instructions (opcodes) from them. To prevent the iC-96 compiler from creating these vector tables and placing them in the code memory space, use the iC-96 `inst` control or add one dummy case statement that is extremely out of the sequential case range. For example, the following statement produces compare operations instead of generating a vector table:

```
switch (i)
{ case 1: a = 10;
  case 2: a = 9;
  case 1000: a = 0;
}
```

2.2.3.2 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.
- Code generated by PL/M-96 case statements and iC-96 switch statements contain vector table data, which is read, intermixed with the code, which is fetched. See Section 2.2.3.1 for information on how to prevent generation of vector table data. Therefore, portions of code that contain case or switch statements must be placed in memory that does not depend on the logical level of the INST pin. This situation can involve memory that does not decode INST for case and switch statements.

- Constants are also read as data and you can scatter them throughout the application code. This example handles constants by making all of them global and placing them in one module. This technique allows constants to be located in the area that unaffected by the INST pin.

2.2.3.3 Guidelines for Hardware Development

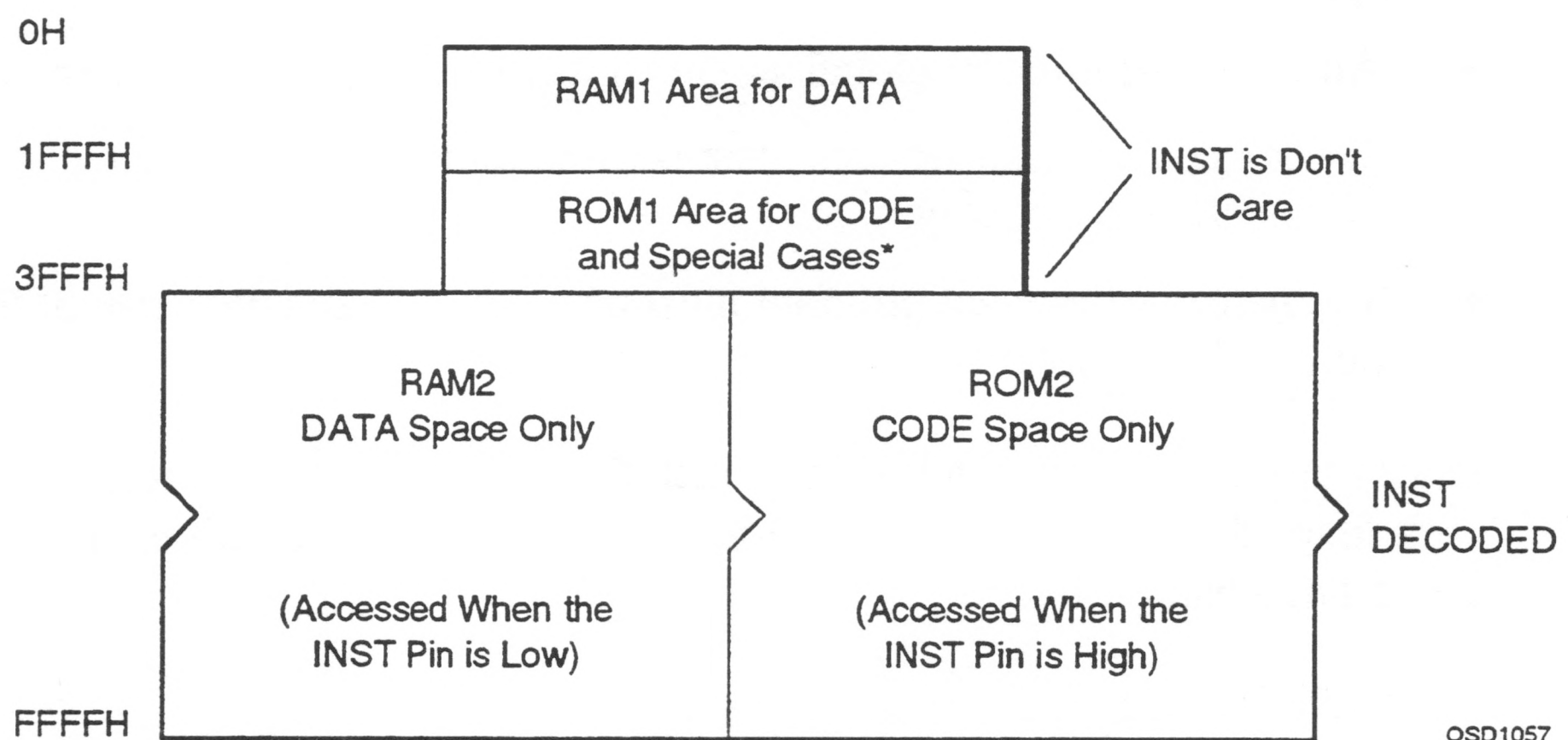
Figure 2-1 depicts the example system. This system provides 16 kilobytes of memory (00H to 3FFFH) not decoded by the INST pin and 48 kilobytes of overlapping code and data memory (4000H to 0FFFFH). The Boolean expressions describing the memory scheme are shown in the following notations. The overbar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

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

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

$$\text{RAM2} = (\text{A15} + \text{A14}) * \overline{\text{INST}}$$

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



OSD1057

* Use this ROM area for constants, PL/M-96 case statements, iC-96 switch statements, interrupt vectors, and other special memory locations.

Figure 2-1 112-Kilobyte Overlapping Code and Data Memory Map

2.2.3.4 Guidelines for Software Development

Follow these guidelines during software development when creating the code for this example. By adhering to these guidelines, you can manipulate the code and data when locating the application software:

- Make all constants global and place them in one module. This module is named `const` in the RL96 linker/locator example that follows.
- Place PL/M-96 case statements and iC-96 switch statements in separate modules. These modules are referred to as `switch` and `case` in the example RL96 invocation line.
- Place the following iC-96 modules, if used, in memory where code and data do not overlap: `ctype`, `printf`
- Place the following FPAL96 modules, if used, in memory where code and data do not overlap: `fpsqrt_mod`, `fpcleb_mod`, `fpcom`, `fpal`, `fpexcept_mod`, `fplddec_mod`, `fpround_mod`, `fpstdec_mod`

2.2.3.5 Linker/Locator Example Invocation Line

The following conditions must hold for this example RL96 invocation line to work properly:

- All constants must be global and must reside in the module `const.obj`.
- Switch and case statements must be in modules called `switch.obj` and `case.obj`, respectively.
- The name of the main program must be `main.obj` and the name of the output file must be `final`.

Place the RL96 invocation line in a batch file to avoid having to retype it each time RL96 is invoked. See Section 2.5 for more information on how to create DOS batch files.

The RL96 invocation line is as follows:

```
rl96 cstart.obj, main.obj, const.obj, switch.obj, &  
case.obj, c96.lib, plm96.lib to final &  
ram(100H-1FFFH, 4000H-FFFFH) &  
rom(2000H-3FFFH(const,switch, case), 4000H-FFFFH) inst
```

2.2.3.6 Summary of RL96 INST Usage

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

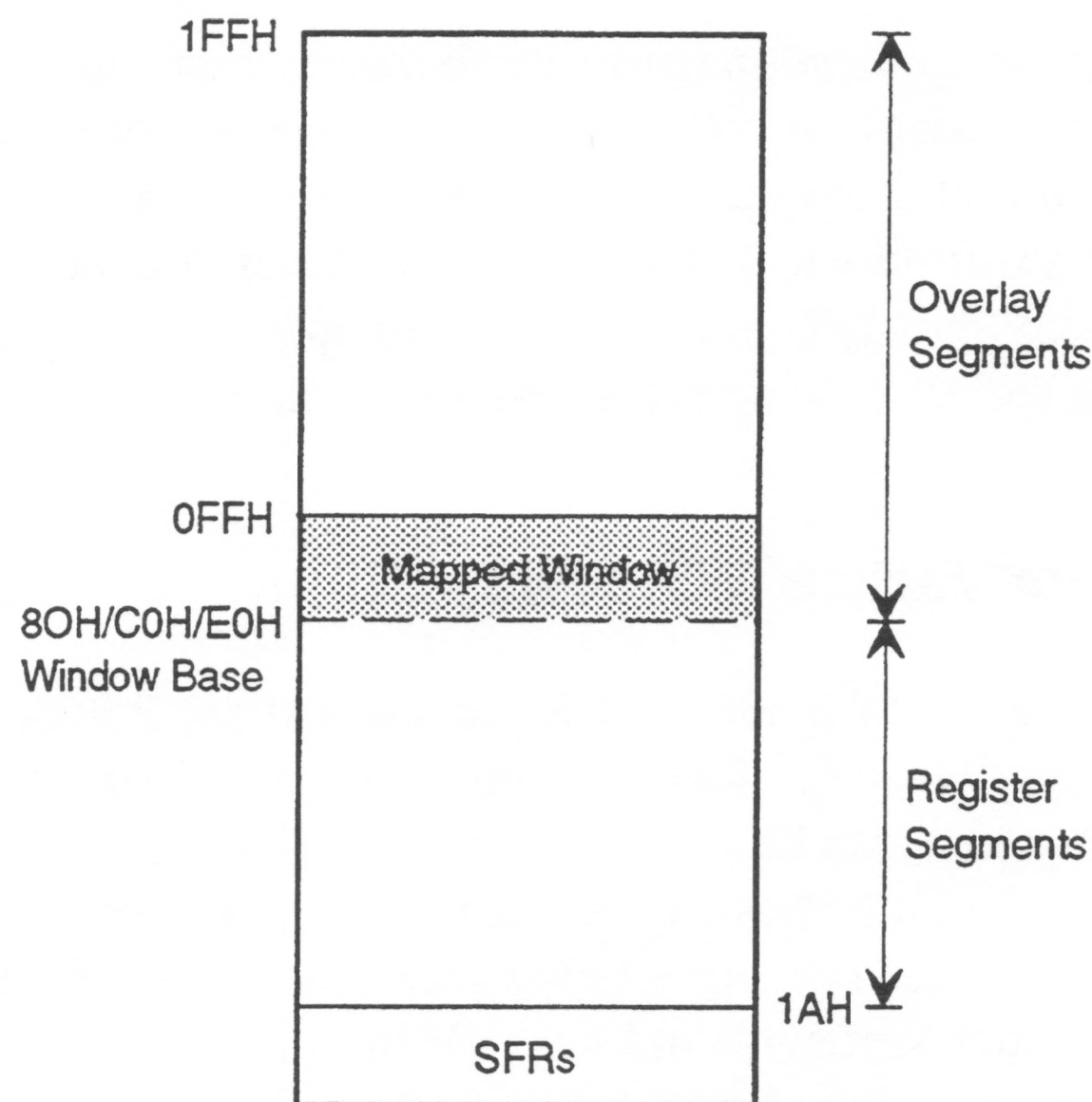
- Make constants global and place them in a separate module. Another alternative is to use the `#define` preprocessor directive in iC-96 or the `literally` keyword in PL/M-96 when defining constants. The compiler places the constant values in the code stream instead of locating them in readable memory.
- Place PL/M-96 `case` and iC-96 `switch` statements in a separate module.
- Design the application hardware so that at least some memory do not contain overlapping code and data. This non-overlapped code and data area is used for constants, `case` statements, and `switch` statements. Addresses from 0 to 16K work well for the non-overlapped area. These addresses already contain a mixture of special locations that are read or written and special locations that are fetched.

2.2.4 Support for Vertical Windows

The 80C196KC and the 80C196KR processors have 256 additional registers from 100H through 1FFH. 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 *16-Bit Embedded Controller Handbook*, listed in Chapter 1, for more information on register windowing.

The 80C196KC and 80C196KR 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 iC-96 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 ASM96 modules, RL96 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.



OSD1053

Figure 2-2 80C196KC Register Allocation Scheme

RL96 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, RL96 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 RL96'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 2.6 for more information on these controls.

2.3 Invoking RL96

The general syntax for the invocation line is:

```
[pathname] rl96 input_list [to output_file] [controls]
```

Where:

- | | |
|--------------------|---|
| <i>pathname</i> | is the device and/or directory where RL96 resides. |
| <i>input_list</i> | 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 <code>publiconly</code> list. These types of files are discussed in the following section. |
| <i>output_file</i> | is the (optional) file that receives the output object module. See Section 2.3.2 for more details. |
| <i>controls</i> | is an optional list of one or more controls. See Section 3.2 for explanation on each control. |

The next several sections give details of the elements of the RL96 invocation command.

2.3.1 Input List

The input list tells RL96 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. You can specify a library file with or without an explicit module list. The following sections describe each of these possibilities.

2.3.1.1 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 RL96. The file consists of a sequence of object modules included in the output object file, although the file usually contains only one module. LIB96 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:

```
> rl96 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`.

2.3.1.2 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 LIB96 librarian creates and maintains an object library file, as explained in Chapter 4. The file consists of a set of object modules. RL96 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, RL96 includes the selection unconditionally, as in the following example:

```
> rl96 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, RL96 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:

```
> rl96 main.obj, sort.obj, debug.obj, &  
>> intrpt.lib(int0, int3, int4), &  
>> fpa196.lib, c96.lib
```

In addition to the modules selected in the previous example, the output file includes modules selected from the libraries `fpa196.lib` and `c96.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 *iC-96 Compiler User's Guide for DOS Systems*, listed in Chapter 1, for more information on linking libraries and object files.

2.3.1.3 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, RL96 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:

```
> rl96 main.obj, sort.obj, debug.obj, &  
>> intrpt.lib(int0, int3, int4,) &  
>> fpal96.lib, c96.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`:

```
> rl96 main
```

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

```
> rl96 main to main.96
```

2.3.2 Output Files

RL96 produces three outputs: screen messages, a print file, and an object file. Screen messages are discussed in Chapter 5. Use the listing controls to choose the information that appears in the print file.

2.3.2.1 Print File

The *filename.m96* (default extension) print file produced by RL96 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, RL96 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. RL96 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)`.

System-id MCS-96 RELOCATOR AND LINKER, x.y
Copyright 1985 Intel Corporation

INPUT FILES: MN.OBJ, ERR.OBJ, PRT.OBJ, PLM96.LIB

OUTPUT FILES: EXAMPL

CONTROLS SPECIFIED IN INVOCATION COMMAND: LI IX OV(MN, ERR) NOAS

PU(LN, SB, PL) RO(2080H-3FFFH) PW(80)

INPUT MODULES INCLUDED:

MN.OBJ(MN) 01/01/90

ERR.OBJ (ERR) 01/01/90

PRT.OBJ(PRT) 01/01/90

PLM96.LIB(PLMREG) 09/21/90

PLM96.LIB(FRAM01) 09/21/90

SEGMENT MAP FOR EXAMPL(MN):

	TYPE	BASE	LENGTH	ALIGNMENT	MODULE NAME
	----	----	-----	-----	-----
**RESERVED*		0000H	001AH		
	REG	001AH	0002H	WORD	FRAM01
	REG	001CH	0008H	ABSOLUTE	PLMREG
	OVRLY	0024H	0015H	WORD	ERR
OVERLAP	OVRLY	0024H	0002H	WORD	MN
GAP		0039H	0001H		
	REG	003AH	0061H	WORD	ERR
GAP		009BH	0001H		
	REG	009CH	0024H	WORD	MN
	DATA	00C0H	022EH	WORD	PRT
GAP		02EEH	1D92H		
	CODE	2080H	0003H	ABSOLUTE	MN
GAP		2083H	0001H		
	CODE	2084H	0DEDH	WORD	ERR
	CODE	2E71H	09CAH	BYTE	PRT
	CODE	383BH	00BBH	BYTE	MN
GAP		38F6H	C70AH		
REL	STACK		0054H	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:
REG	ARRAY	00A6H	MN__FILES__INFO
CODE	ENTRY	38C3H	MN__GLOBAL__CONCLUDE
CODE	ENTRY	2967H	ERR__SET__NEXT
CODE	ENTRY	297AH	ERR__INITIALIZE
REG	NULL	001CH	PLMREG
REG	WORD	001AH	?FRAME01
			MODULE: MN
			LINE#:
		3856H	115
		3856H	116
		3864H	117
		3867H	118
CODE	ARRAY	383BH	COPYRIGHT
REG	STRUCTURE	009CH	MN__STACK
REG	WORD	00A4H	MN__BOTTOM__OF__STACK
REG	ARRAY	00A6H	MN__EFILES__INFO
		3856H	PROC: GLOBAL__INITIALIZE
		38C3H	PROC: MN__GLOBAL__CONCLUDE
OVRLY	WORD	0024H	COMPLETION__CODE
			MODULE: ERR
REG	STRUCTURE BASED	003EH	ERROR__NODE
CODE	STRUCTURE	209EH	INSUFF__M86__ERR__NODE
			MODULE: PLMREG
			MODULE: FRAM01

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, DATA, 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, SHORTINT, INTEGER, LONGINT, POINTER, ARRAY, and STRUCTURE.

The type ENTRY stands for both labels and procedures/functions in PL/M-96 and iC-96 as well as the ENTRY of ASM96.

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 indentation indicates scope. Public symbols are preceded by the key word `PUBLICS` on a separate line; line numbers are preceded by the key word `LINE#:` 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. RL96 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 iC-96 counts every line in the source file.

Intermodule Cross-reference Listing

The optional intermodule cross-reference listing includes an entry for each global symbol. RL96 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 (<code>NULL</code>, <code>CODE</code>, <code>DATA</code>, <code>STACK</code>, <code>REG</code>, <code>OVERLAY</code>) appear exactly the same as the segment type field in the symbol table, except that <code>DYNAMIC</code> 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 -----
?FRAME01	REG WORD	; FRAM01 ERR PRT
ERR__ADD__SPECIAL	CODE ENTRY	; ERR
ERR__GET__ERROR__KIND	CODE ENTRY	; ERR
ERR__HANDLER__FOR__OMF96P	CODE ENTRY	; ERR MN
ERR__INITIALIZE	CODE ENTRY	; ERR MN
ERR__SET__NEXT	CODE ENTRY	; ERR
FIL__GET__NAME	CODE ENTRY	; ** UNRESOLVED ** ER -R
FIL__INITIALIZE	CODE ENTRY	; ** UNRESOLVED ** MN
MN__BOTTOM__OF__STACK	REG WORD	; MN
MN__FILES__INFO	REG ARRAY	; MN ERR
MN__GLOBAL__CONCLUDE	CODE ENTRY	; MN ERR
MN__STACK	REG STRUCTURE	; MN
PLMREG	REG NULL	; PLMREG ERR MN PRT
PRT__CLOSE__FILE	CODE ENTRY	; PRT MN
PRT__LINE__COUNT	CODE ENTRY	; PRT
PRT__MESSAGE	CODE ENTRY	; PRT ERR
PRT__OPEN__FILE	CODE ENTRY	; PRT
PRT__SET__PAGE__LENGTH	CODE ENTRY	; PRT MN
PRT__SET__PAGE__WIDTH	CODE ENTRY	; PRT MN
PRT__SET__SUBTITLE	CODE ENTRY	; PRT MN
PRT__SET__TITLE	CODE ENTRY	; PRT MN
PRT__SET__WRAPAROUND__PARAM	CODE ENTRY	; PRT ERR
PRT__SPACE	CODE ENTRY	; PRT ERR
PRT__TERMINATE__LINE	CODE ENTRY	; PRT ERR

WARNING 2: UNRESOLVED EXTERNAL SYMBOL
 SYMBOL: FIL__INITIALIZE
 MODULE: MN.OBJ(MN)

WARNING 2: UNRESOLVED EXTERNAL SYMBOL
 SYMBOL: FIL__GET__NAME
 MODULE: ERR.OBJ(ERR)

RL96 COMPLETED, 2 WARNING(S), 0 ERROR(S)

Figure 2-5 Sample Intermodule Cross-reference

Error Messages

Error messages appear at the end of the print file. RL96 error messages are categorized as warnings, errors, and fatal errors. Only fatal errors terminate the RL96 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 5 for a complete list of RL96 error messages of all types and their probable causes.

2.3.2.2 Output Object File

RL96 generates an output absolute or quasi-absolute object file that contains:

- The content of the program segments. This information is always present in the RL96 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.

NOTE

To prevent any inappropriate error messages from the PROM programmer during loading of an MCS-96 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, `rl96 main.obj to main.abs`, assigns `main.abs` as the output object file. If you omit the output filename, RL96 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, RL96 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 an Quasi-absolute Object File

If the output file is to be reused by RL96, that is the final absolute file is produced after an incremental linkage, you must specify `noabsstack` in the RL96 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 RL96.

2.4 Invocation Examples

Your RL96 invocation line can list several object module files and controls so that the whole command does not fit in a single invocation line. To continue a long command line, enter an ampersand character (&) just before the line terminator. Type the rest of your command line after the continuation prompt (>>) appears. You can also enter comment lines on the invocation line by placing the comments after the ampersand because the linker takes no action on all characters that appear after the ampersand but before the carriage return/line feed that terminates the line.

2.5 Automatically Invoking Multiple Commands

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

2.5.1 Using DOS Batch Files

A DOS 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 ASM96 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:\mcs96\asm96 %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.

```
C:> c:\mcs96\96a prog1
```

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

```
c:\mcs96\asm96 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 `label`. 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 ASM96 source program, passes the resulting object module to RL96, and invokes OH to convert the final object module to hexadecimal format. RL96 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. asm96 assemble and link:
asm96 %1.a96
rl96 %1.obj, & < %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,proggxs.lib, &
a0961.obj, mylib961.lib
```

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

```
C:> 96a1 prog1
```

When `96a1.bat` executes, DOS invokes ASM96 to assemble `prog1.a96`, then invokes RL96 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.

2.5.2 Using DOS 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:

```
asm96 prog0.a96
asm96 prog1.a96
r196 /f link_obj.lst
exit
```

The `link_obj.lst` file is a directive file used by the `/f` control. See Chapter 3 for more details. This file contains the following:

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

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

```
C:> 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`.
- The ampersand character (&), used to continue a long line, is not valid in a command file. Use the `/f` control to append the contents of the directive file to the command line. See Chapter 3 for more information on the `/f` control.
- `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`:

```
C:> 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`.

2.6 RL96 Controls

All but one of the RL96 controls, namely the `/f` control, modify the default operation of the linker. These controls fit into three functional groups:

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

The `/f` control is an invocation-only control that appends the contents of a directive file to the command line. The directive file can contain all of the object module filenames and controls to be processed by the linker. Use this control instead of using the ampersand character (`&`), to continue a long line, when using a command file. See Section 2.5.2 for more information on command files.

You can enter more than one control on the RL96 invocation line. Separate controls by spaces, not commas. If you enter the same control more than once, a fatal error results and RL96 aborts.

The RL96 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 2-3 lists the RL96 controls by group.

Table 2-3 RL96 Controls

Group	Control Name	Abbreviation	Default
Invocation only	/f		n/a
Listing	ixref/noixref	ix/noix	noixref
	list/nolist	li/noli	list(all) ¹
	pagewidth	pw	pagewidth(120)
	print/noprint	pr/nopr	print(<i>file.m96</i>) ²
Linking	name	na	name(<i>mod_name</i>) ³
	purge/nopurge	pu/nopu	nopurge
	typecheck/notypecheck	tc/notc	typecheck
Locating	absstack/noabsstack	as/noas	absstack
	inst/noinst	on/noin	noinst
	ram	ra	ram(1AH-1FFFH (stack)) ⁴
	registers	rg	registers(1AH-0FFFH) ⁶
	regoverlay/noregoverlay	ov/noov	noregoverlay
	rom	ro	rom(2000H-200FH, 2080H-0FFFFH).
	stacksize	ss	stacksize(<i>total</i>) ⁵
	window size	ws	window size(0)

Notes:

¹ The *all* placeholder stands for publics, symbols, lines, and segments.

² The *file.m96* placeholder indicates that the default is the first input filename followed by the extension .m96.

³ The *mod_name* placeholder indicates that the default name is the name of the first input module.

⁴ The ram 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).

⁵ The *total* placeholder indicates that the default stacksize is the total size of all stack segments from input modules. If *total* is less than 6, the default stacksize is 6.

⁶ Indicates that no vertical windowing is used.

The remainder of this section explains each control in detail. The controls 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 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 Notational Conventions on the inside front cover for special meanings of type styles used in by this manual.

/f

Invocation-only control: appends the contents of a directive file to the command line

Syntax

```
/f directive-file
```

Where:

directive-file is a text file containing object module filenames and/or linker controls.

Discussion

Use this control to append the contents of the directive file to the command line. The `/f` control must be the last control you specify or the linker generates an error. You can specify other invocation elements, such as filenames and controls, before the `/f` control. If you do, make sure that the contents of the directive file, when appended to the command line, validly completes the invocation line. See example 2 for an example of an invalid command line.

The directive 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(+1000h)
```

Examples

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

```
> r196 /f link.lst
```

The file `link.lst` contains the following:

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

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

```
, prog2.obj, ic96.lib
```

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

```
> rl96 prog1.obj, cstart.obj, ss(+2h) /f link.lst
```

This is because the resulting invocation string is as follows:

```
> rl96 prog1.obj, cstart.obj, ss(+2h), prog2.obj,  
ic96.lib
```

absstack

Locating control: determines whether the stack segment of an output module is absolute
default: `absstack`

Syntax

`absstack | noabsstack`

Abbreviation

`as | noas`

Discussion

Use this control to specify whether you want RL96 to absolutely locate the stack segment in the resultant RL96 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 2.2.2 for more information on how to do dynamic memory allocation.

Example

The following example produces no absolute stack:

```
> r196 mod1.obj, mod2.obj noas
```


inst

Locating control: allows ROM and RAM addresses to be overlapped
default: noinst

Syntax

`inst | noinst`

Abbreviation

`in | noin`

Discussion

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 MCS-96 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 2.2.3 for hardware and software development guidelines and a sample RL96 invocation.

ixref

Listing control: include intermodule
cross-reference listing in the print file
default: noixref

Syntax

```
ixref | noixref
```

Abbreviation

```
ix | noix
```

Discussion

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 2.3.2 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:

```
> r196 mod1.obj, mod2.obj ixref
```

Cross-reference

```
print
```

list

Listing control: specifies elements to be included in the print file
default: list(publics, symbols, lines, segments)

Syntax

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

Where:

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

Abbreviation

```
li | noli
```

Discussion

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.

list (continued)

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:

```
> rl96 def.obj list(segments, publics)
```

Cross-reference

print

name

Linking control: assigns a module name
to the output file
default: `name(first_input_module_name)`

Syntax

`name(module_name)`

Where:

`module_name` is a string of characters.

Abbreviation

na

Discussion

Use this control to assign a module name, specified by `module_name`, to the output module produced by RL96. 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.

name (continued)

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

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

NOTE

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 RL96. The output absolute filename is *program1*. The print file produced is *program1.m96*.

```
> rl96 program1.obj name(monitor)
```


pagewidth

Listing control: specifies the maximum number
of characters per line
default: pagewidth(120)

Syntax

```
pagewidth(number)
```

Where:

number is a valid number from 72 to 132. This number can also be specified in hexadecimal format.

Abbreviation

pw

Discussion

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 132, 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.

```
> r196 remote.obj ix pw(90)
```

print

Listing control: directs the listing produced to the specified file
default: `print(first_input_file.m96)`

Syntax

```
print(output_file) | noprint
```

Where:

output_file is a string of characters.

Abbreviation

```
pr | nopr
```

Discussion

Use this control to direct the listing produced by RL96 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`.

```
> r196 mod1.obj, mod2.obj print(96_appl.m96)
```

Cross-references

```
list  
ixref
```

purge

Linking control: specifies which elements
are removed from the output file
default: nopurge

Syntax

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

Where:

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

Abbreviation

```
pu | nopu
```

Discussion

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 RL96 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.

purge (continued)

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

CAUTION

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:

```
> rl96 mod1.obj, mod2.obj purge(publics)
```

Cross-reference

absstack

ram

Locating control: specifies the designated RAM section
default: ram (1AH-1FFFH(stack))

Syntax

```
ram(ram_section [(module_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.

Abbreviation

ra

Discussion

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 0FFFFH.

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. 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

ram (continued)

appear as a pseudo-module name in a module list of this control which implies that RL96 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 segment 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.

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`.

```
> rl96 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

Locating control: specifies the range of registers available to the application

default: registers(1AH-0FFH) or registers(26-255)

Syntax

```
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

Discussion

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 *end_address* must be greater than or equal to its *start_address*.

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 2.2.4 for more information on how RL96 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.

```
> rl96 mod1.obj, mod2.obj, mod3.obj, mod4.obj &  
>> registers(1AH-1FFH) windowsize(64)
```

Cross-reference

window size

regoverlay

Locating control: overlay register
default: noregoverlay

Syntax

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

Where:

overlay_unit $\Rightarrow$ *overlay_element* [{ -> | - } *overlay_element*]

overlay_element $\Rightarrow$ *module_name* | (*overlay_unit* [...])

module_name is a valid name of a module.

Abbreviation

ov | noov

Discussion

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 MCS-96 environment, you can only overlay registers in the relocatable overlay segments. 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 RL96 during memory allocation of overlay segments.

In the calls relationship, the right arrow signifies a hyphen followed by a greater than sign ($->$) or a hyphen followed by a right square bracket ($-]$). 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 ($-]$) can be used on all operating systems.

A right arrow ($->$) designates the calls relationship. *A Calls B* means that module A cannot overlay module B. The calls relationship is transitive; i.e., $A \rightarrow B$ 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 $X \rightarrow Y$ or $Y \rightarrow X$ is specified and neither of the relationships can be deduced (by transitivity), then the linker concludes that

regoverlay (continued)

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 RL96 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.

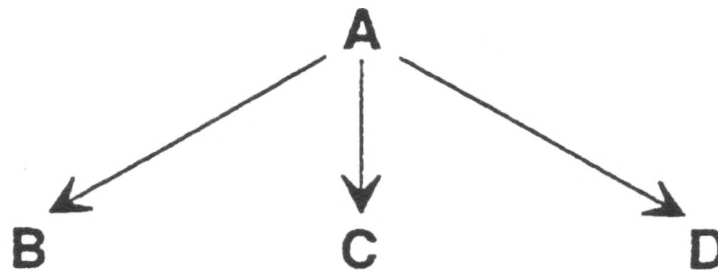
If the linker finds a call or a jump between two overlaid modules, that is, their relocatable overlay segments are overlaid, the linker issues the following warning:

```
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:

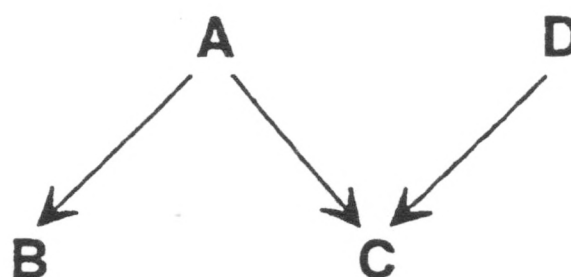


You can ask for the maximum overlaying possible by using either of the control sequences:

- `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:

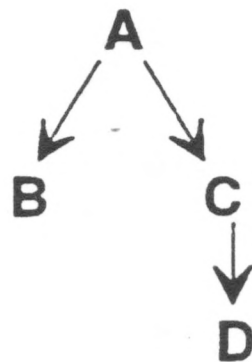
`ov(A -> (B,C), D -> C)`

You can also specify the same structure:

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

regoverlay (continued)

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)
```

Locating control: specifies designated ROM sections
Default: rom(2000H-200FH, 2080H-0FFFFH)

Syntax

```
rom(rom_section [(module_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 a list of valid module names.

Abbreviation

ro

Discussion

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 1000H.
- The *end_address* must be greater than or equal to its *start_address*. The maximum *end_address* is 0FFFFH.

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. 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.

rom (continued)

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.

NOTE

Once you explicitly specify a particular ROM section with this control, you must specify all of the remaining ROM sections.

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.

```
> rl96 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-2FFFH, 3000H-4000H, 5001H-5FFFH. After all absolute segments are allocated, mod1's code is allocated within 2000H-2FFFH, mod7's code also have to be allocated within the same range, but only after mod1's allocation. The code for mod3 is allocated within 5001H-5FFFH, then the rest of the relocatable code segments are allocated within the free memory sections of the range.

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

stacksize

Locating control: specifies the size of the stack segment

default: `stacksize(total)` if *total* ≥ 6

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

Syntax

```
stacksize( [ { + | - } ] 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 0FFFFH.

Abbreviation

SS

Discussion

Use this control to specify or modify the resultant RL96 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 0FFFFH, 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-196PC, ICE-196KB/HX or ICE-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.

```
> rl96 main.obj stacksize(880)
```

typecheck

Linking control: specifies if type checking is performed
default: typecheck

Syntax

typecheck | notypecheck

Abbreviation

tc | notc

Discussion

When this control is in effect, the linker performs type checking during public-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.

RL96 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:

```
> r196 mod1.obj, mod2.obj notypecheck
```

window size

Locating control: specifies the desired window size for vertical windowing

Syntax

```
window size(n)
```

Where:

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

Abbreviation

WS

Discussion

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 iC-96 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 2.2.4 for more information on how the linker allocates register and overlay segments in vertical windows.

window size (continued)

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.

```
> rl96 mod1.obj, mod2.obj, mod3.obj, mod4.obj &  
>> registers(1AH - 01FFH) window size(64)
```

Cross-reference

registers

Contents

The OH Converter

3.1	Invocation Syntax.....	3-1
3.2	Examples.....	3-1
3.3	Output File.....	3-2



The OH Converter

The OH object-to-hexadecimal converter converts an absolute MCS[®]-96 object file to a hexadecimal file for use with tools that do not accept standard Intel object file formats.

3.1 Invocation Syntax

The invocation syntax for OH is:

```
oh abs_objfile [to hexfile]
```

Where:

abs_objfile is an absolute MCS-96 object file created by RL96.

hexfile is the file to contain the hexadecimal output.

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, OH generates a fatal error and aborts the processing of the object file. See Chapter 5 for a complete list of error messages and their causes.

3.2 Examples

1. The following example converts the absolute MCS-96 object file created by RL96, `sort`, to hexadecimal format and places the output into the file `sort.hex`.

```
oh sort
```


2. The following example converts the absolute MCS-96 object file `save1.obj` to hexadecimal format and places the output into the file `save.h96`.

```
oh save1.obj to save.h96
```

3. The following example converts the absolute MCS-96 object file `tot2.obj` to hexadecimal format and places the output into the file `tot2.hex`.

```
oh tot2.obj
```

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). OH 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

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

If the information in an OMF record occupies more than 16 bytes, OH divides it into 16-byte hexadecimal records. If any bytes are left over after this operation, OH combines those remaining bytes with bytes left over from adjacent records in the OH buffer. These adjacent bytes can be part of either a previous or a subsequent record. If no adjacent bytes are available for combination, OH 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
:10209000000401CFE6C201CFE4F010040201CFE4C2A
:1020A00020201CFE7F01004020FE7C2220FE5F01DC
:1020B000000402220FE5C222220FE8F0100401CFEF8
: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 OMF 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 OH converter computes the checksum by first adding the binary representation of the previous bytes, starting with 10H to 01H, in this example. OH 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 OMF record in bytes is one less than a multiple of 16.

Contents

The LIB96 Librarian

4.1	Invoking LIB96.....	4-1
4.1.1	Noninteractive Mode.....	4-1
4.1.2	Interactive Mode.....	4-2
4.1.3	Character Set.....	4-2
4.2	LIB96 Commands.....	4-3
4.3	Command Descriptions.....	4-3
	add.....	4-4
	create.....	4-6
	delete.....	4-7
	exit.....	4-8
	extract.....	4-9
	help.....	4-10
	list.....	4-11
	replace.....	4-12

4

1950

1950-1951

1951



The LIB96 Librarian

LIB96 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 LIB96

You can invoke the LIB96 utility interactively or noninteractively. The following sections describe the syntax of each mode.

4.1.1 Noninteractive Mode

The following is the general syntax for invoking in noninteractive mode:

```
[pathname] lib96 command
```

Where:

pathname is the device and/or directory where LIB96 resides.

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 you need to extend your command line to the second line, use the ampersand (&). When the continuation prompt (>>) appears, type in the rest of your command line. You can enter comments in your command line by placing them between the ampersand and the carriage return that terminates the line.

4.1.2 Interactive Mode

Invoke LIB96 in interactive mode, as follows:

```
[pathname] lib96
```

Where:

pathname is the device or directory on which LIB96 resides.

LIB96 responds with its sign-on message. Enter the LIB96 commands after the LIB96 prompt (*). After each command is executed, another prompt for the next command appears. This process continues until you enter the `exit` command.

Often, you must extend the LIB96 command to beyond one line. Use an ampersand (&) to continue your command on another line. You can place the ampersand continuation character anywhere that a space normally appears in the command line.

Whenever you enter the continuation character, LIB96 begins a new line with the continuation prompt of two asterisks (**). You can then enter the additional line of input. If you cannot complete the command on the second line, use more ampersands to continue the process until the command is completely entered.

You can place a semicolon anywhere on the command line to start a comment. LIB96 takes no action on all characters that appear between the semicolon and the carriage return that terminates the line. If you enter an ampersand to the right of a semicolon, the librarian processes the ampersand as part of a comment and not as the continuation character.

4.1.3 Character Set

The LIB96 character set consists of the letters A through Z, the digits 0 through 9 and the special characters ?, @, and _.

4.2 LIB96 Commands

Table 4-1 summarizes the LIB96 commands.

Table 4-1 LIB96 Commands

Command	Abbrev.	Description
add { <i>file</i> [(<i>module</i> [,...])] } [...] to <i>library_file</i>	a	adds modules to a library
create <i>library_file</i>	c	creates a library file
delete <i>library_file</i> (<i>module</i> [,...])	d	deletes modules from library
exit	e	terminates session with LIB96
extract { <i>file</i> [(<i>module</i> [,...])] } [...] to <i>file</i>	x	extracts modules from libraries
help	h	displays syntax of LIB96 commands
list { <i>file</i> [(<i>module</i> [,...])] } [...] <i>file</i>] [publics]	l [p]	lists modules contained in to libraries, and optionally lists all publics
replace { <i>file</i> [(<i>module</i> [,...])] } [...] in <i>library_file</i>	r	replaces modules in a library

4.3 Command Descriptions

The remainder of this section explains each LIB96 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 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 Notational Conventions on the inside front cover for special meanings of type styles used in by this manual.

add

Add specified file to the library

Syntax

```
add { file [ ( module [...]) ] } [...]  
to library_file
```

Where:

file is the filename of the module.

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

library_file is the name of the library being added to.

Abbreviation

a

Discussion

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, RL96, or the `extract` command of LIB96.

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.

You cannot use the `add` command before you create the library.

Examples

In the following examples, the library commands are entered at the library prompt (*).

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

```
* add sin, cos, tan to user.lib
```

To perform the same command in noninteractive mode, type the following:

```
C:> lib96 add sin, cos, tan to user.lib
```

2. This command adds the three modules `mod1`, `mod2`, and `mod3` of the library `lib.tmp` to the destination library `proj.tom`. In the example, the ampersand continues the command to the second line.

```
* add lib.tmp(mod1, mod2, mod3) to &  
** proj.tom
```


create

Create library file

Syntax

```
create library_file
```

Where:

library_file is the name you give to the new library being created.

Abbreviation

c

Discussion

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 5 for a complete list of error messages.

Example

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

```
* create new.lib
```

delete

Delete specified module from library

Syntax

```
delete library_file ( module_name [...])
```

Where:

library_file is the name of the existing library.

module_name is the name of the module being deleted.

Abbreviation

d

Discussion

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, LIB96 issues a warning.

Example

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

```
* delete new.lib(truth,value)
```

exit

Terminate library manager program

Syntax

`exit`

Abbreviation

`e`

Discussion

Use this command to terminate LIB96 during interactive mode. Control returns to the operating system. In noninteractive mode, LIB96 does not execute the `exit` command.

extract

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

Syntax

```
extract {file [ (module [...]) ]} [...] to obj_file
```

Where:

file is the name of the existing library file.

module is the name of the module copied to the object file.

obj_file is the name of the object file being created.

Abbreviation

x

Discussion

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 library and remain unchanged.

Example

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

```
* extract new.lib(worth,free) to hold.obj
```

help

Displays a summary of the syntax on the console.

Syntax

```
help
```

Abbreviation

```
h
```

Discussion

Use this command to display a summary of the LIB96 command syntax on the screen. The summary display shows you the commands and their arguments. The syntax for all commands appears on the screen as follows:

```
add { file [ (module [...]) ] } [...] to library_file
create library_file
delete library_file (module [...])
exit
extract { file [ ( module [...]) ] } [...] to obj_file
help
list {file [ ( module [...]) ] } [...] [to file] [publics]
replace {file [ ( module [...]) ] } [...] in library_file
```

list

Print the name of the modules inside the library

Syntax

```
list {library_file [ ( module [...]) ]} [...] [to out_file] [publics]
```

Where:

library_file is the name of the library.

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

out_file is the name of the output file.

Abbreviation

l [p]

Discussion

Use this command to print the names of the modules, and optionally, the names of the public symbols to the specified destination output file. If you omit the `to` clause, the librarian directs the listing to the console output. Specify `publics` to list all public symbols contained in those modules with the module names.

Examples

1. List all module names in the library `user.lib`.
* `list user.lib`
2. List all public symbols in the module `temp` in the library `user.lib`.
* `list user.lib(temp) publics`

replace

Replace designated object in library file

Syntax

```
replace { file [ (module [...]) ] } [...] in library_file
```

Where:

file is the name of file containing the module.

module is the name of the new module.

library_file is the name of the existing library.

Abbreviation

r

Discussion

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.
* replace worth, free in new.lib
2. The newer version of time in module counter replaces the older version in library user.lib.
* replace counter(time) in user.lib

Contents

Messages and Error Recovery

5.1	RL96 Messages.....	5-1
	5.1.1 Console Messages.....	5-2
	5.1.2 Fatal Errors.....	5-3
	5.1.3 Error Messages.....	5-8
	5.1.4 Warnings.....	5-11
5.2	OH Error Messages.....	5-16
5.3	LIB96 Error Messages.....	5-16

Messages and Error Recovery

5

This chapter provides a list of all the console, warning, error, and fatal error messages produced by the RL96 linker, OH converter, and LIB96 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.

5.1 RL96 Messages

The RL96 linker generates three different error messages: warning, error, and fatal error. 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, RL96 displays the names of the file and module containing the segment and the segment's type classification, as follows:

```
SEGMENT:  CODE
MODULE:   FILEA(MODB)
```

The handling of stack segments differs from the handling of other segments. RL96 indicates an error in the stack segment by displaying the following:

```
SEGMENT:  STACK
MODULE:   <dummy> (STACK)
```

Note, however, that neither the file < *dummy* > nor the module STACK exists.

RL96 supplies the public symbols `MEMORY` and `?MEMORY__SIZE` 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.

5.1.1 Console Messages

RL96 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 reads:

```
system_id MCS-96 RELOCATOR AND LINKER, Vx.y
```

Where:

system_id is the DOS version number.

x.y is the RL96 version number.

When the linker completes its processing, the following sign-off message is sent to the console:

```
RL96 COMPLETED, nnn WARNING(S), mmm ERROR(S)
```

Where:

nnn is the number of warnings issued by RL96.

mmm is the number of errors issued by RL96.

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.

5.1.2 Fatal Errors

The following list contains all fatal error messages generated by RL96:

FATAL ERROR 201: INVALID COMMAND LINE SYNTAX
NEAR: *token*

A syntax error was detected in the command line near the specified token.

FATAL ERROR 202: INVALID COMMAND LINE. TOKEN TOO LONG
NEAR: *token*

FATAL ERROR 203: EXPECTED ITEM MISSING
NEAR: *token*

An expected item in the command line near the specified token (e.g., an input filename or a filename following to) is missing. A filename longer than 128 characters also causes these errors.

FATAL ERROR 204: INVALID KEYWORD
NEAR: *token*

An invalid keyword was found in the command line near the specified token.

FATAL ERROR 206: INVALID CONSTANT
NEAR: *token*

An illegally constructed constant was found in the command line near the specified token. For example, RL96 found a hexadecimal number that begins with a letter. These hexadecimal numbers must be preceded with a zero (0).

FATAL ERROR 207: INVALID MODULE NAME
NEAR: *token*

The specified token of the command line is an illegal module name. See name control in Section 3.2.

FATAL ERROR 208: INVALID FILE NAME
NEAR: *token*

The specified token of the command line is an illegal filename. See Section 3.1 for more information on filenames.

FATAL ERROR 209: FILE USED IN CONFLICTING CONTEXTS
FILE: *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.

FATAL ERROR 210: I/O ERROR, INPUT FILE
FILE: *filename*
MESSAGE: *message*

The linker detected an I/O error while accessing the specified input file. The message describes the kind of error detected.

FATAL ERROR 211: I/O ERROR, OUTPUT FILE
FILE: *filename*
MESSAGE: *message*

The linker detected an I/O error while accessing the specified output file. The message describes the kind of error detected.

FATAL ERROR 212: I/O ERROR, PRINT FILE
FILE: *filename*
MESSAGE: *message*

The linker detected an I/O error while accessing the specified print file. The message describes the kind of error detected.

FATAL ERROR 213: DUPLICATE KEYWORD
NEAR: *token*

The specified keyword appears in the command line more than once.

FATAL ERROR 215: CHECKSUM ERROR
MODULE: *filename(module_name)*

The linker detected a bad checksum in the specified input module. This message indicates a bad input module.

FATAL ERROR 216: INSUFFICIENT MEMORY

The memory available for execution of RL96 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.

FATAL ERROR 217: NO MODULE TO BE PROCESSED

After scanning all the input files, RL96 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.

FATAL ERROR 218: INVALID INPUT OBJECT FILE 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.

FATAL ERROR 219: NOT AN 8096 OBJECT MODULE MODULE: *filename(module_name)*

The `translator_id` field in the module header record indicates that the specified module is not an MCS-96 object module. Another possible cause is the RL96 version is not compatible with the translator (or tool) that has produced the object file. Check the RL96 version.

FATAL ERROR 221: ADDRESSES NOT IN ASCENDING ORDER NEAR: *token*

Addresses specified in a locating control near the specified token are not in ascending order. For example, the following control line generates this error: `rom(3000H-4000H(mod1),3000H-4000H(mod2))`

FATAL ERROR 222: ADDRESS OUT OF RANGE NEAR: *token*

An address specified in a locating control near the specified token 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`.

FATAL ERROR 223: OVERLAP BETWEEN ROM AND RAM RANGES
NEAR: *token*

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.

FATAL ERROR 224: THE STACK MAY NOT BE SPECIFIED
THERE AS A MODULE
NEAR: *token*

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.

FATAL ERROR 225: INTERNAL PROCESSING ERROR
KIND: *integer*

RL96 has made a processing error. This error message indicates a problem within RL96. Report such errors to Intel Field Service Personnel by calling the appropriate number on the inside back cover of this manual.

FATAL ERROR 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.

FATAL ERROR 227: MODULE USED IN CONFLICTING CONTEXTS
MODULE: *filename(module_name)*

The specified module name has already been specified explicitly in 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.

FATAL ERROR 228: PARAMETER IS NOT ALLOWED IN THAT CONTEXT
NEAR: *token*

The negative form of the control (with the prefix *no*) cannot have parameters. The error is near the specified token.

FATAL ERROR 229: THE PREFIX "NO" IS NOT ALLOWED
FOR THIS CONTROL

NEAR: *token*

The control has no negative form. The error is near the specified token.

FATAL ERROR 230: INVOCATION LINE TOO LONG

The invocation line is too long, that is, it contains too many characters.

FATAL ERROR 231: AN ORDINARY FILE MAY NOT BE SPECIFIED WITH
A MODULE LIST

FILE: *filename*

RL96 process an ordinary file as a whole; modules can only be selected out of a library file.

FATAL ERROR 232: A LIBRARY FILE MAY NOT BE SPECIFIED WITH
"PUBLICSONLY"

FILE: *filename*

Only an ordinary file can be specified as publicsonly. See Section 3.1 for definitions of ordinary files and publicsonly files.

FATAL ERROR 233: THE SPECIFIED MODULE DOES NOT EXIST IN THE
SPECIFIED FILE

MODULE: *filename(module_name)*

The module specified in module list of the library file does not exist. Use the LIB96 list command to display all objects modules in a library.

FATAL ERROR 234: I/O ERROR, IXREF FILE

FILE: *filename*

MESSAGE: *message*

During the preparation of the ixref listing, RL96 uses an auxiliary temporary file. The above error indicates that an I/O error was detected during file access. The message describes the kind of error detected.

FATAL ERROR 235: THE SPECIFIED MODULE DOES NOT EXIST

MODULE: *module_name*

A module specified as a parameter of either the ram, rom or regoverlay control does not exist.

FATAL ERROR 236: PARAMETER OUT OF RANGE
NEAR: *token*

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.

FATAL ERROR 237: REGOVERLAY PARAMETER TOO COMPLEX

The specified parameter is too complex (e.g., too long) to be processed by RL96. Simplify your parameter.

FATAL ERROR 242: INVALID REGISTER RANGE

The address range specified in the *registers* control does not conform to the component's register space. See the *16-Bit Embedded Controllers Handbook*, listed in Chapter 1, for memory space information.

FATAL ERROR 243: WINDOWSIZE 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).

FATAL ERROR 244: INVALID WINDOWSIZE 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.

5.1.3 Error Messages

The following is a complete list of all error messages generated by RL96:

ERROR 101: CODE MEMORY OVERLAP
FROM: *byte address*
TO: *byte address*

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.

ERROR 102: ERRONEOUS INPUT MODULE
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. RL96 marks an output object module as erroneous. Be sure that all input modules are processed without any errors.

ERROR 103: MEMORY OVERFLOW
SEGMENT: *segment type*
MODULE: *filename(module_name)*

RL96 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.

ERROR 105: INCOMPATIBLE STACK SEGMENTS
MODULE: *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. RL96 takes no action on the indicated stack segment.

ERROR 110: NO ROOM FOR RELOCATABLE SEGMENT IN THE
SPECIFIED RANGE
SEGMENT: *segment type*
MODULE: *filename(module_name)*
FROM: *byte address*
TO: *byte address*

This message indicates that RL96 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.

ERROR 111: INSTRUCTION OPERAND DOESN'T MEET
ALIGNMENT REQUIREMENT

MODULE: *filename(module_name)*

OFFSET: *offset or base*

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.

ERROR 112: INSTRUCTION OPERAND OUT OF RANGE IN
A JUMP/CALL INSTRUCTION

MODULE: *filename(module_name)*

OFFSET: *offset or base*

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.

ERROR 113: INSTRUCTION OPERAND OUT OF RANGE

MODULE: *filename(module_name)*

OFFSET: *offset or base*

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 iC-96 programs, or `fast(all)` for PL/M-96 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 either of these options: `registers(all)` or `fast(all)`.

ERROR 115: MODULE NOT COMPILED FOR WINDOWS
MODULE: `filename(module_name)`

For iC-96 programs, one of the input modules was not compiled with the windows control. For ASM96 programs, no reference to `?wsr` was found. See Chapter 5 of the *iC-96 Compiler User's Guide for DOS Systems* for an example of an ASM96 module written for vertical windows.

ERROR 116: TOO MANY GLOBAL REGISTERS

During incremental links, RL96 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.

5.1.4 Warnings

The following is a complete list of all warning messages generated by RL96:

WARNING 001: SYMBOL ATTRIBUTE MISMATCH
SYMBOL: `symbol name`
MODULE: `filename(module_name)`
MODULE: `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.

WARNING 002: UNRESOLVED EXTERNAL SYMBOL
SYMBOL: *external name*
MODULE: *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 004.

WARNING 003: MULTIPLE PUBLIC DEFINITION
SYMBOL: *public name*
MODULE: *filename(module_name)*
MODULE: *filename(module_name)*

The specified symbol was declared as public in the first specified module and also in the second module. RL96 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.

WARNING 004: REFERENCE MADE TO UNRESOLVED EXTERNAL
SYMBOL: *external name*
MODULE: *filename(module_name)*
OFFSET: *offset or base*

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.

WARNING 005: MODULE NAME NOT UNIQUE
MODULE: *filename(module_name)*

Another module with the same name has been processed. RL96 does not process the specified module of the specified file. Recheck all your module names.

WARNING 006: MORE THAN ONE MAIN MODULE
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`, RL96 issues this error for the last three modules. The output object module is marked as `main`.

WARNING 007: SPECIFIED STACK SIZE TOO SMALL

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 009).

WARNING 008: ILLEGAL SPECIFIED STACK SIZE. SHOULD BE EVEN.

The stack size must be an even number because stack operations are performed on word items. RL96 adds 1 to the specified `stacksize` (modulo 2^{16}).

WARNING 009: STACK ALREADY LOCATED. ''STACKSIZE'' IGNORED.

The `stacksize` control was specified but the stack segment of the input modules is already absolute (i.e., located). RL96 takes no action on the `stacksize` control.

WARNING 010: DATA MEMORY OVERLAP
FROM: *byte address*
TO: *byte address*

The memory in the given (inclusive) range is occupied by two or more absolute segments. None of the overlapping segments is a code segment.

WARNING 012: 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 0FFFFEH. Stack size is set to 0FFFFEH.

WARNING 013: STACK ALREADY LOCATED. ''NOABSSTACK''
IGNORED.

The noabsstack control was specified but the stack segment of the input modules is already absolute (i.e., located). RL96 takes no action on the noabsstack control.

WARNING 014: NO MAIN MODULE

None of the input modules is a main module. Consequently, the output object module also is not marked as main. For iC-96 applications, mark one module as main using ASM96 or include the cstart.obj in your RL96 invocation.

WARNING 016: SYMBOL DEFINED OUT OF SEGMENT

SYMBOL: *symbol name*
MODULE: *filename(module name)*
SEGMENT: *segment type*
OFFSET: *offset or base*

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 ASM96, or an at clause, in PL/M-96.

WARNING 017: ABSOLUTE SEGMENT DOES NOT FIT

SEGMENT: *segment type*
MODULE: *filename(module_name)*
FROM: *byte address*
TO: *byte address*

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.

WARNING 020: TYPE DEFINITION TOO COMPLEX

MODULE: *filename(module_name)*

The specified module contains a type definition that is too complex to be processed by RL96. 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.

WARNING Ø21: A DIRECT CALL BETWEEN TWO OVERLAID MODULES
SYMBOL: *symbol name*
MODULE: *filename(module_name)*
MODULE: *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.

NOTE

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 Section 3.2.

WARNING Ø22: 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 (ØEØH). In this case, RL96 could not generate vertical windows and uses no register space above ØFFH. RL96 then locates the overlay segments in the remaining unallocated register space up to ØFFH. If all of your overlay segments do not fit under ØFFH, RL96 generates a memory overflow error. To resolve this problem, reduce the number of global register variables.

WARNING Ø23: WINDOW SIZE SPECIFIED TOO LARGE - IGNORED

RL96 cannot generate a window with the window size you specified in the `window size` control. The linker uses the biggest possible window size.

5.2 OH Error Messages

OH 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: OH <infile> [TO outfile]

This message means that the invocation syntax is incorrect. OH expected the `to` keyword but found something else. Reinvoke OH 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 OH has detected an invalid format. Be sure the input object file is available and that it is an absolutely located file.

5.3 LIB96 Error Messages

The following is a list of LIB96 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

LIB96 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.

Glossary

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.
base address	A starting address from which an absolute address can be calculated by combination with an offset.
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.
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.
error	An exception that does not immediately terminate the program's operation but can cause an invalid object module.
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.
fatal error	An unrecoverable error detected by the executing program.
fixup	Instructions placed in the object file that allow RL96 to fix undetermined calls in the code image.
incremental linking	Linking modules in small subgroups before linking the subgroups together.
keyword	A character string that has special meaning to the program. See reserved word.
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.
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.
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.
OH	MCS [®] -96 object code to hexadecimal conversion utility.
opcode fetch	Reading an instruction from memory.

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.
quasi-absolute file	An object file that contains a relocatable stack segment.
RAM	Random access memory
register	A high-speed storage location on a processor chip.
register file	MCS-96 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.
RL96 linker	MCS-96 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.
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 MCS-96 component used for hardware control.
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.
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.
unresolved external	A symbol that is not matched by a public symbol in one of the input modules.
variable	A quantity that can assume any of a set of values, or a symbol that refers to a value.
work files	Files created and deleted by the development tool during translation.

Index

& (ampersand), 2-36, 4-2
80C196KC processor, 2-17
80C196KR processor, 2-17

A-B

a command. *See* add command
About this manual, 1-5
Absolute object file, 1-4
 Creating an, 2-32, 2-42
Absolute segment, 2-1, 2-8
absstack control, 2-8 thru 2-10, 2-42
add command, 4-4
Alignment, 2-1, 2-24
as control. *See* absstack control
Attribute field, 2-27, 2-28
Audience, 1-5
Base address, 2-1, 2-8, 2-24
BASED attribute, 2-27

C

c command. *See* create command
Calling graph, 2-59
Calls relationship. *See* regoverlay control
Case statements, 2-12
CCB. *See* Chip configuration byte
CCR. *See* Chip configuration register
Code segment, 1-2, 2-8, 5-8, 5-9
 Type, 2-27, 2-28, 2-44
Controls
 /f, 2-40
 absstack, 2-42
 inst, 2-43
 ixref, 2-44
 Linking, 2-38
 list, 2-45
 Listing, 2-38
 Locating, 2-38
 name, 2-47

 pagewidth, 2-49
 print, 2-50
 purge, 2-51
 ram, 2-53
 register, 2-56
 regoverlay, 2-58
 RL96, 2-19
 rom, 2-63
 stacksize, 2-66
 typecheck, 2-68
 window size, 2-69

Conventions, 1-6. *See also* Notational conventions

create command, 4-6
Creating libraries, 1-3
Cross-reference, 2-44
 Listing, 1-3
Customer comments, 1-6
Customer service hotline, 1-6

D

d command. *See* delete command
Data segment types, 2-27, 2-28, 2-44
 array pointer, 2-27, 2-28, 2-44
 byte, 2-28, 2-44
 entry, 2-27, 2-28, 2-44
 integer, 2-27, 2-28, 2-44
 long, 2-27, 2-28, 2-44
 longint, 2-27, 2-28, 2-44
 null, 2-27, 2-28, 2-44
 real, 2-27, 2-28, 2-44
 shortint, 2-27, 2-28, 2-44
 structure, 2-27, 2-28, 2-44
 word, 2-27, 2-28, 2-44
debug control, 2-31
Debugging, 2-31
 With ICE-196PC, 2-66

- Definition records, 2-2
 - External, 2-2
 - Public, 2-2
- delete command, 4-7
- Directive File, appending, 2-40
- DOS batch files, 2-33
- DOS command files, 2-35
- Dynamic memory allocation, 2-9, 2-42
- Dynamic segment type, 2-27, 2-28

E

- e command. *See* exit command
- Error messages, 2-31, 5-2, 5-16
 - Fatal, 5-1
 - LIB96, 5-16
 - Location, 2-31
 - OH, 5-16
 - RL96, 5-8
 - Warnings, 5-1
- Executable files. *See* Absolute object file
- exit command, 4-8
- External references, 1-3
- extract command, 4-9

F-H

- Fatal error messages
 - Fix-up evaluation, 2-7
 - RL96, 5-3
- First fit/decreasing size algorithm, 2-9
- Global register variables, 2-19
- Global symbols
 - External, 2-3
 - Public, 2-3
- h command. *See* help command
- help command, 4-10
- .hex file, 3-2

I

- ICE-196KB/HX, 2-11
- in control. *See* inst control
- Incremental linking, 2-2, 2-10
- inst control, 2-43
- INST pin, 2-11
 - Behavior, 2-11
 - Chip configuration byte, 2-11

- Chip configuration register, 2-11
- Interrupt vector table, 2-11
- Opcode fetch, 2-11
- Program constants and variables, 2-12
- Vector tables, 2-12

- Hardware development guidelines, 2-15
- Overlapping memory scheme, 2-14
- RL96 invocation example, 2-16
- Software development guideline, 2-16

- Installation, vii

- Instruction pin. *See* INST pin

- Intermodule cross-reference listing, 2-28, 2-44.

- See also* ixref control

- Invocation examples, 2-32

- Invocation line

- LIB96, 4-1

- OH, 3-1

- RL96, 2-19

- ix control. *See* ixref control

- ixref control, 2-44

J-L

- l command. *See* list command

- li control. *See* list control

- LIB96

- Character set, 4-2

- Command entry, 4-2

- Commands, 4-3

- Error messages, 5-16

- Invocation line, 4-1

- Interactive mode, 1-3, 4-1

- Noninteractive mode, 1-3, 4-1

- Library manager, 1-3, 4-1

- Library files, 1-3, 2-20

- Line numbers, 2-46, 2-51

- Line number information.

- Including. *See* list control

- Purging. *See* purge control

- Link summary information, 2-23

- Linkage, 1-2

- list command, 4-11

- list control, 2-24, 2-26, 2-45

- ln. *See* Line number information

- Local register variables, 2-19

Local symbols, 2-46, 2-51

.lst extension, 1-5

M

.m96file, 2-23

Main module, 2-1, 2-20

Map file, creating a, 2-50

Matching segment type, 2-3, 2-68

Matching symbol type, 2-3, 2-68

Memory allocation, 2-8

Memory segment, 2-8

Code, 2-8

Data, 2-8

Overlay, 2-8

Register, 2-8

?MEMORY__SIZE symbol, 2-9, 5-1

MEMORY symbol, 2-9, 5-1

N

na control. *See* name control

name control, 2-47

noabsstack control, 2-7, 2-42, 2-52

noas control. *See* noabsstack control

noin control. *See* noinst control

noinst control, 2-43

noix control. *See* noixref control

noixref control, 2-44

noli control. *See* nolist control

nolist control, 2-45

noov control. *See* noregoverlay control

nopr control. *See* noprint control

noprint control, 2-50

nopu control. *See* nopurge control

nopurge control, 2-31, 2-51

noregoverlay control, 2-58

notc control. *See* notypecheck control

notypecheck control, 2-3, 2-68

null

Segment type, 2-27, 2-44

Symbol type, 2-5, 2-27, 2-44

O

Object library file, 2-20, 2-21

OH object-to-hexadecimal converter, 1-5, 3-1

Error messages, 5-16

Invocation line, 3-1

Output file, 3-2

Record type, 3-2

Opcode fetch, 2-11

Order of allocation, 2-8

Ordinary object file, 2-19, 2-20, 4-4

Output object module, naming, 2-47

ov control. *See* regoverlay control

Overlapping memory scheme, 2-14

Overlapping ROM and RAM, 2-11, 2-43

Overlay segments, 2-9, 2-19, 2-58

Type, 2-27, 2-28, 2-44

P

pagewidth control, 2-49

Performing fix-ups, 2-7

pl. *See* Public symbols

pr control. *See* print control

print control, 2-50

Print file, 2-45

PROM programmer, 1-4, 2-31

Loading, 1-4

pu control. *See* purge control

Public symbols, 2-2, 2-10, 2-51

Including. *See* list control

Purging. *See* purge control

Publications, related, 1-5

Publiconly object file, 1-3, 2-20, 2-22

purge control, 2-31, 2-51

Conserving space, 2-32

pw control. *See* pagewidth control

Q-R

Quasi-absolute object file, 2-2, 2-32, 2-42

Creating an, 2-32

r command. *See* replace command

ra control. *See* ram control

ram control, 2-8, 2-53

RAM memory, 1-2, 2-6, 2-8, 2-10, 5-5

Designating, 2-53

Reg. *See* Register segment

- Register
 - 8-bit direct addressing mode, 2-17
 - 16-bit direct addressing mode, 2-17
 - Allocation, 2-17
 - Overlay segment, 2-18
 - Overlaying, 2-9
 - Range, 2-56
 - Segment, 1-2, 2-8, 2-18, 2-19, 2-56
 - Type, 2-27, 2-28, 2-44
- registers control, 2-8, 2-19, 2-56
- regoverlay control, 2-8, 2-9, 2-58, 5-15
- Relocatable segment, 2-1, 2-9
- Relocatable symbol, 2-7
- Relocation, 1-2
- replace command, 4-12
- Resolving external references, 2-2
- rg control. *See* registers control
- RL96 controls, 2-20, 2-37
 - Linking, 2-37
 - Listing, 2-37
 - Locating, 2-37
- RL96 messages, 5-1
 - Error messages, 5-8
 - Fatal, 5-3
 - Warning messages, 5-11
- RL96 input modules
 - Object library file, 2-20, 2-21
 - Conditional processing, 2-21
 - Syntax, 2-21
 - Ordinary object file, 2-20
 - Publiconly object file, 2-20, 2-22
 - Selecting, 2-20
- RL96 invocation line, 2-19
 - Input list, 2-19
 - Object library file, 2-21
 - Ordinary object file, 2-20
 - Publiconly object file, 2-22
- RL96 linker and locator, 1-2, 2-1
 - Major functions, 2-1
- RL96 outputs
 - Console display, 5-2
 - Object file, 2-23
 - Print file, 2-23
- ro control. *See* rom control
- rom control, 2-8, 2-63
- ROM memory, 1-2, 2-8, 5-5, 5-6
 - Designating, 2-63
- ROM/RAM overlapping, 2-43
- S
 - sb. *See* Symbol information
 - Segments
 - Including. *See* list control
 - Map, 1-2, 2-24
 - Option, 2-24
 - Placement, 2-1
 - Purging. *See* purge control
 - Type matching, 2-3, 2-68
 - Types, 2-27, 2-28, 2-44
 - sm. *See* Segments
 - Software development process, 1-1
 - Software installation, vii
 - ss control. *See* stacksize control
 - st. *See* stack segment
 - Stack segment, 2-1, 2-20. *See also* absstack control; stacksize control
 - Size, 2-66
 - Type, 2-27, 2-28, 2-44
 - stacksize control, 2-66, 5-13
 - Static storage type, 2-19
 - switch statements, 2-12
 - Symbol base, 2-27
 - Symbol information
 - Including. *See* list control
 - Purging. *See* purge control
 - Symbol references, 2-2
 - Symbol table, 1-3, 2-26
 - Value, 2-26
 - Symbol type matching, 2-4
 - Between ASM96 and iC-96, 2-5
 - Symbol types, 2-6, 2-44. *See also* Data segment types
- T
 - tc control. *See* typecheck control
 - Trademarks, 1-7
 - Type checking, 2-3, 2-68
 - Type matching
 - More restriction, 2-6
 - Symbol, 2-4

- Type mismatch, 2-7
 - Output of RL96, 2-7
 - Warning, 2-7
- typecheck control, 2-3, 2-68

U

- Unallocated segments, 2-9
- Unresolved external symbols, 2-3
- Unresolved references, 1-3
- Utilities
 - LIB96, 4-1
 - OH, 3-1
 - RL96, 2-19

V

- Value field, 2-27
- Vector tables, 2-12
- Vertical windows, 2-17, 2-56, 2-69
 - Register allocation scheme, 2-19
 - Size, 2-56, 2-69
 - Using, 2-17
- VWindows. *See* Vertical windows

W-Z

- Warning messages, 2-67
 - RL96, 5-11
- Windows
 - Horizontal, 2-18
 - Mapping, 2-18
 - Vertical, 2-18
- windows control, 2-19
- window size control, 2-19, 2-69
- ws control. *See* window size control
- WSR management code, 2-19
- ?wsr variable, 2-19
- x command. *See* extract command



READER RESPONSE CARD

We'd Like Your Opinion

Please use this form to help us evaluate the effectiveness of this manual and improve the quality of future versions.

To order publications, contact the Intel Literature Distribution Center (see page ii of this manual).

Fill in the blanks below with a rating of 1 through 10, where 1 is poor and 10 is excellent:

- ___ Readability
- ___ Technical depth
- ___ Technical accuracy
- ___ Usefulness of material for your needs
- ___ Comprehensibility of material
- ___ OVERALL QUALITY OF THIS MANUAL

If you have any comments, please include them here:

What suggestions would you have for improving this manual:

If you would like us to call you for more specific suggestions about this book, please additionally fill in your phone number below.

Name

Phone Number (

)

Address

Thanks for taking the time to fill out this form.



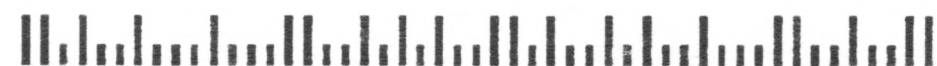
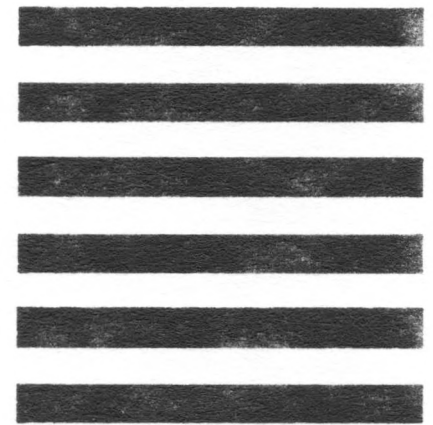
NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 HILLSBORO OR

POSTAGE WILL BE PAID BY ADDRESSEE

**DTO TECHNICAL PUBLICATIONS JF1-59
INTEL CORPORATION
5200 NE ELAM YOUNG PARKWAY
HILLSBORO OR 97124-9978**



Please fold here and close the card with tape. Do not staple.

WE'D LIKE YOUR COMMENTS....

This document is one of a series describing Intel products. Your comments on the other side of this form will help us produce better manuals. Each reply will be reviewed. All comments and suggestions become the property of Intel Corporation.

If you are in the United States and are sending only this card, postage is prepaid.

If you are sending additional material or if you are outside the United States, please insert this card and any enclosures in an envelope. Send the envelope to the above address, adding "United States of America" if you are outside the United States.

Thanks for your comments.

Intel International Sales Offices

AUSTRALIA

Intel Australia Pty. Ltd.
Unit 13
Allambie Grove Business Park
25 Frenchs Forest Road East
Frenchs Forest, NSW 2086
Sydney

Intel Australia Pty Ltd.
711 High Street
1st Floor
East Kw. Vic., 3102
Melbourne

BELGIUM

Intel Corporation S.A.
11, av. Winston Churchill
1180 Brussels

BRAZIL

Intel Semicondutores do Brazil LTDA
Av. Paulista, 1159-CJS 404/405
01311 - Sao Paulo - S.P.

CANADA

Intel Semiconductor of Canada, Ltd.
4585 Canada Way, Suite 202
Burnaby V5G 4L6
British Columbia

Intel Semiconductor of Canada, Ltd.
2650 Queensview Drive
Suite 250
Ottawa K2B 8H6
Ontario

Intel Semiconductor of Canada, Ltd.
190 Attwell Drive
Suite 500
Rexdale M9W 6H8
Ontario

Intel Semiconductor of Canada, Ltd.
1 Rue Holiday, Suite 115
Tour East
Pt. Claire H9R 5N3
Quebec

CHINA/HONG KONG

Intel PRC Corporation
15/F, Office 1, Citic Bldg.
Jian Guo Men Wai Street
Beijing, PRC

Intel Semiconductor Ltd.
10/F East Tower
Bond Center
Queensway, Central
Hong Kong

FINLAND

Intel Finland OY
Ruosilantie 2
00390 Helsinki 39

FRANCE

Intel Corporation S.A.R.L.
1, Rue Edison-BP 303
78054 St. Quentin-en-Yvelines
Cedex

GERMANY

Intel GmbH
Dornacher Strasse 1
8016 Feldkirchen bei Muenchen

Intel GmbH
Wandersmannstr, 64
D-6200 Wiesbaden

Intel GmbH
Verkaufsburo Stuttgart
Zettachring 10A

INDIA

Intel Asia Electronics, Inc.
4/2, Samrah Plaza
St. Mark's Road
Bangalore 560001

IRELAND

Intel Ireland Ltd.
Collinstown Industrial Park
Leixlip County Kildare

Intel International Sales Offices (Continued)

ISRAEL

Intel Semiconductor Ltd.
Atidim Industrial Park-Neve Sharet
P.O. Box 43202
Tel-Aviv 61430

ITALY

Intel Corporation Italia S.p.A.
Milanofiori Palazzo E
20090 Assago
Milano

JAPAN

Intel Japan K.K.
5-6 Tokodai, Tsukuba-shi
Ibaraki, 300-26

Intel Japan K.K.
Bldg. Kumagaya
2-69 Hon-cho
Kumagaya-shi, Saitama 360

Intel Japan K.K.
Hachioji ON Bldg.
4-7-14 Myojin-machi
Hachioji-shi, Tokyo 192

Intel Japan K.K.
Kawa-asa Bldg.
2-11-5 Shin-yokohama
Kohoku-ku, Yokohama-shi
Kanagawa, 222

Intel Japan K.K.
Ryokuchi-Eki Bldg.
2-4-1 Terauchi
Toyonaka-shi, Osaka 560

Intel Japan K.K.
Shinmaru Bldg.
1-5-1 Marunouchi
Chiyoda-ku, Tokyo 100

Intel Japan K.K.
Green Bldg.
1-16-20 Nishiki
Naka-ku, Nagoya-shi
Aichi 460

KOREA

Intel Korea, Ltd.
16th Floor, Life Bldg.
61 Yoido-Dong, Youngdeungpo-Ku
Seoul 150-010

NETHERLANDS

Intel Semiconductor Nederland B.V.
Alexander Poort Building
Marten Meesweg 93
Postbus 84130
3068 AV Rotterdam

SINGAPORE

Intel Singapore Technology, Ltd.
101 Thomson Road #08-03/06
United Square
Singapore 1130

SPAIN

Intel Iberia
Calle Zurbaran, 28
28010 Madrid 04

SWEDEN

Intel Sweden A.B.
Dalvagen 24
171136 Solna

SWITZERLAND

Intel Semiconductor A.G.
Zuerichstrasse
8185 Winkel-Rueti bei Zuerich

TAIWAN

Intel Technology Far East Ltd.
Taiwan Branch Office
8th Floor, No. 205
Bank Tower Bldg.
Tung Hua N. Road
Taipei

UNITED KINGDOM

Intel Corporation (U.K.) Ltd.
Pipers Way
Swindon, Wiltshire SN3 1RJ

Service Information

For service or assistance with Intel products, call:

- **1-800-843-4481** - in the United States and Canada
- Your local Intel sales office - in Europe or Japan
- Your Intel distributor - in any other area

Intel stands behind its products with a world-wide service and support organization. If you have problems, need assistance, or have a question, Intel can provide:

- On-site or carry-in service for hardware products
- Phone support for all Intel products
- On-site consulting for designing with Intel products or using Intel products in your designs
- Customer training workshops
- Updates to software products

To get more information on these services or to take advantage of them, call the number listed above.

All Intel products have a minimum warranty of 90 days, and all warranties include one or more of the services listed above. Talk with your Intel salesperson or call 1-800-843-4481 to determine the warranty services available for this product and how to register for them.

Intel is committed to continuing service for all its products.



INTEL CORPORATION, 2200 Mission College Drive, Santa Clara, California 95052-8119 (408) 987-8080

Printed in U.S.A.

DEVELOPMENT TOOLS AND SOFTWARE