intel

MCS®-51 UTILITIES USER'S GUIDE FOR 8080/8085-BASED DEVELOPMENT SYSTEMS

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Order Number: 121737-003

MCS®-51 UTILITIES USER'S GUIDE FOR 8080/8085-BASED DEVELOPMENT SYSTEMS

Order Number: 121737-003

Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

Intel 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.

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.

The following are trademarks of Intel Corporation and its affiliates and may only be used to identify Intel products:

BITBUS	i_m	iRMX	Plug-A-Bubble
COMMputer	iMMX	iSBC	PROMPT
CREDIT	Insite	iSBX	Promware
Data Pipeline	int_el	iSDM	QueX
Genius	int_elBOS	iSXM	QUEST
i	Intelevision	Library Manager	Ripplemode
$^{\text{a}}$	int_eligent Identifier	MCS	RMX/80
I^2ICE	int_eligent Programming	Megachassis	RUPI
ICE	Intellec	MICROMAINFRAME	Seamless
iCS	Intellink	MULTIBUS	SOLO
iDBP	iOSP	MULTICHANNEL	SYSTEM 2000
iDIS	iPDS	MULTIMODULE	UPI
iLBX			

REV.	REVISION HISTORY	DATE	APPD.
-001	Original issue.	9/81	
-002	Added IXREF and LIB51.	11/82	
-003	Added OVERLAY/NOOVERLAY controls.	11/83	M.S.

PREFACE

This manual describes the RL51 linker and locator and the LIB51 librarian for program modules produced by MCS-51 language translators such as ASM51 and PL/M-51.

The RL51 and the LIB51 program operate on an Intel development system with an 8080 or 8085 processor. The configuration must include 64K of RAM, a console, and at least one diskette or hard disk drive running under the ISIS-II operating environment.

NOTE

In this manual, the term MCS-51 refers to all members of the MCS-51 family of microcomputers and to the software development tools for the MCS-51 family.

Reader's Guide

The manual is organized into six chapters and five appendixes:

Chapter 1 discusses the advantages of modular programming and summarizes the process of modular programming with the MCS-51 development tools.

Chapter 2 reviews the mechanics of linkage and location for the RL51 program.

Chapter 3 gives the details on invoking the linker/locator.

Chapter 4 discusses the files and displays produced by the RL51 program, with examples.

Chapter 5 contains three examples of programs, with the link and locate steps for each program.

Chapter 6 describes the LIB51, the MCS-51 library manager and its usage.

Appendix A presents the syntax of the RL51 commands with brief definitions of the controls.

Appendix B lists the error messages and warnings displayed by RL51, with suggestions for corrective action.

Appendix C lists a summary of LIB51 commands.

Appendix D lists the error messages generated by LIB51, with suggestions for corrective action.

Appendix E contains hexadecimal-decimal conversion tables as a convenient reference.

Related Literature

To help you use this manual, you should be familiar with the contents of the following manuals:

- *MCS-51 Macro Assembler User's Guide*, order number 9800937
- *PL/M-51 User's Guide*, order number 121966
- *MCS-51 Family of Single-Chip Microcomputers User's Manual*, order number 121517
- *ISIS-II User's Guide*, order number 9800306

The following manuals may also be of interest:

- *MCS-51 Macro Assembly Language Pocket Reference*, order number 9800935
- *MCS-51 Utilities Pocket Reference*, order number 121817
- *ICE-51 In-Circuit Emulator Operating Instructions for ISIS-II Users*, order number 9801004

Notational Conventions

UPPERCASE	Characters shown in uppercase must be entered in the order shown. Enter the command words as shown, or use a system-defined abbreviation. You may enter the characters in uppercase or lowercase.
italic	Italic indicates a meta symbol that may be replaced with an item that fulfills the rules for that symbol. The actual symbol may be any of the following:
directory-name	Is that portion of a ***pathname*** that acts as a file locator by identifying the device and/or directory containing the ***filename***.
filename	Is a valid name for the part of a ***pathname*** that names a file.
system-id	Is a generic label placed on sample listings where an operating system-dependent name would actually be printed.
V*x.y*	Is a generic label placed on sample listings where the version number of the product that produced the listing would actually be printed.
[ ]	Brackets indicate optional arguments or parameters.
{ }	One and only one of the enclosed entries must be selected unless the field is also surrounded by brackets, in which case it is optional.
{ }...	At least one of the enclosed items must be selected unless the field is also surrounded by brackets, in which case it is optional. The items may be used in any order unless otherwise noted.
|	The vertical bar separates options within brackets [ ] or braces { }.
...	Ellipses indicate that the preceding argument or parameter may be repeated.
[,...]	The preceding item may be repeated, but each repetition must be separated by a comma.
punctuation	Punctuation other than ellipses, braces, and brackets must be entered as shown. For example, the punctuation shown in the following command must be entered: `SUBMIT PLM86(PROGA,SRC,'9 SEPT 81')`

`input lines`	In interactive examples, user input lines are printed in white on black to differentiate them from system output.
‹cr›	Indicates a carriage return.

CONTENTS

TABLES

FIGURES

CHAPTER 1
INTRODUCTION

The Advantages of Modular Programming

Many programs are too long or complex to write as a single unit. Programming becomes much simpler when the code is divided into small functional units. Modular programs are usually easier to code, debug, and change than monolithic programs.

The modular approach to programming is similar to the design of hardware that contains numerous circuits. The device or program is logically divided into "black boxes" with specific inputs and outputs. Once the interfaces between the units have been defined, detailed design of each unit can proceed separately.

Efficient Program Development

Programs can be developed more quickly with the modular approach because small subprograms are easier to understand, design, and test than large programs. With the module inputs and outputs defined, the programmer can supply the needed input and verify the correctness of the module by examining the output. The separate modules are then linked and located into one program module. Finally, the completed program is tested.

Multiple Use of Subprograms

Code written for one program is often useful in others. Modular programming allows these sections to be saved for future use. Because the code is relocatable, saved modules can be linked to any program that fulfills their input and output requirements. With monolithic programming, such sections of code are buried inside the program and are not so available for use by other programs.

If you put your frequently-used subprograms into a library, RL51 will take care to load only those you need. Thus, you can save RAM and ROM without having to keep track of what is needed and what is not.

Ease of Debugging and Modifying

Modular programs are generally easier to debug than monolithic programs. Because the modular interfaces are well-defined, problems can be isolated to specific modules. Once the faulty module has been identified, fixing the problem is considerably simpler. When a program must be modified, modular programming simplifies the job. You can link new or modified modules to the existing program with confidence that the rest of the program will not be changed.

MCS®-51 Modular Program Development Process

This section is a brief review of the program development process using an MCS-51 language translator (e.g., the relocatable MCS-51 assembler or PL/M-51 compiler), linker/locator, code converter programs, PROM programmer, and ICE™-51 in-circuit emulator. The process is shown in figure 1-1.

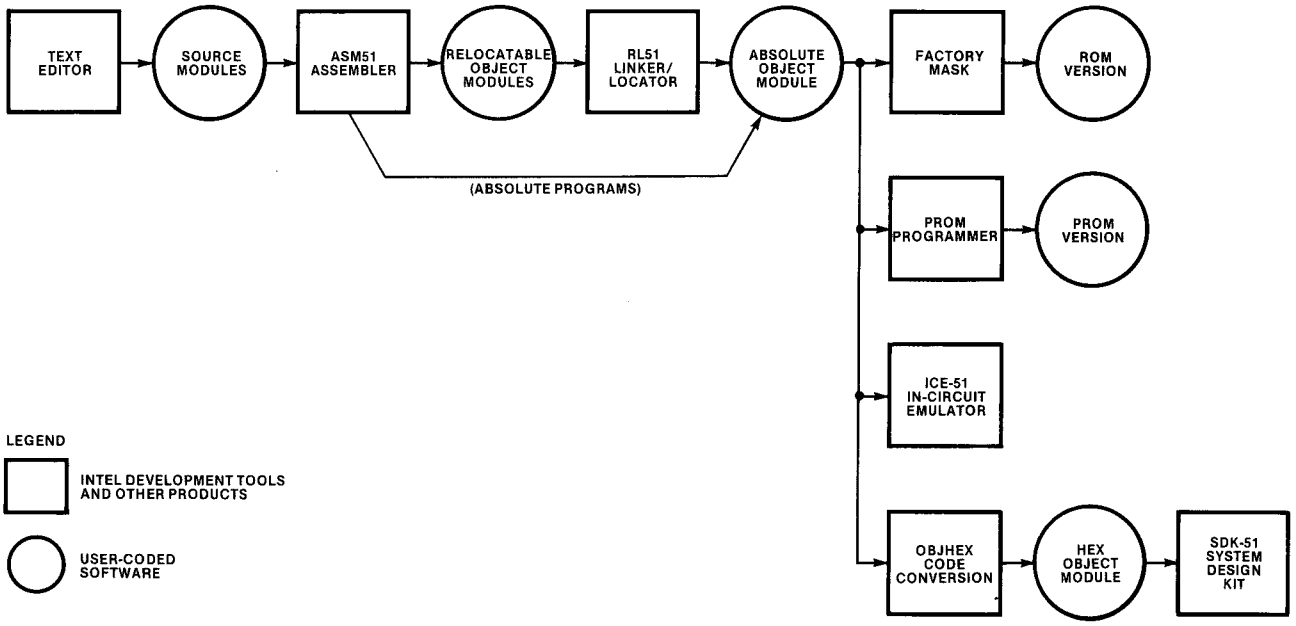


Figure 1-1. MCS®-51 Program Development Process 121737-1

Segments, Modules, Libraries, and Programs

In the initial design stages, the tasks to be performed by the program are defined and then partitioned into subprograms. Here are brief introductions to the kinds of subprograms used with the MCS-51 assembler and linker/locator.

A segment is a unit of code or data memory. A segment may be relocatable or absolute. A relocatable segment in a module can be a complete segment or can be a "partial" segment to be combined with other partial segments from other modules. A relocatable segment has a name, type, and other attributes that allow the linker to combine it with other partial segments, if required, and to correctly locate the segment. An absolute segment has no name and cannot be combined with other segments. See Chapter 2 for more detail on partial segments.

A module contains one or more segments or partial segments. A module has a name assigned by the user. The module definitions determine the scope of local symbols. An object file contains one or more modules. You can add modules to a file by transfering the new modules from their individual files to another file (e.g., COPY *file*1, *file*2 TO *file*3).

A library is a file that contains one or more modules. A library file is internally marked as a library, so RL51 can easily identify it as such. RL51 selects, out of the modules in the library, only those previously referenced. Libraries are created using the LIB51 utility, which is described in detail in Chapter 6.

A program consists of a single absolute module, merging all absolute and relocatable segments from all input modules. The name of the output module produced by RL51 can be defined by the user or allowed to default to the name of the first input module.

Entering and Editing Source Modules

After the design is completed, use the text editor on your system to code the modules into source files. The source modules are coded in assembly language or a high-level language such as PL/M-51. The editor may also be used to make corrections in the source code.

Assembly and Compilation

The assembler (ASM51) and compiler (PL/M-51) translate the source code into relocatable object code, producing an object file. The ASM51 object file is relocatable when at least one input segment is relocatable; otherwise the object file is an absolute file. The PL/M-51 object file is always relocatable. The assembler and compiler also produce a listing file showing the results of the translation. When the ASM51 or PL/M-51 invocation contains the DEBUG control, the object file also receives the symbol table and other debug information for use in symbolic debugging of the program.

Relocation and Linkage

After translation of all modules of the program, the linker/locator, RL51, processes the object module files. The RL51 program combines relocatable partial segments with the same name, then assigns absolute memory locations to all the relocatable segments. RL51 also resolves all references between modules, using the library files when they are necessary for this resolution. RL51 outputs an absolute object module file that contains the completed program, and a summary listing file showing the results of the link/locate process, including a memory map, symbol table, and, optionally, an inter-module cross-reference (IXREF) listing.

ROM and PROM Versions

The absolute object module produced by RL51 can be loaded into members of the MCS-51 family of microcomputers. For ROM versions of the microcomputer, the program is masked into ROM during the manufacturing process. For PROM versions and versions with no on-chip CODE memory, a PROM programmer is used to load the absolute module into program memory accessible to the microcomputer for execution. Refer to the *MCS-51 Family of Single Chip Microcomputers User's Manual* for details on the versions of microcomputers available.

ICE™-51 In-Circuit Emulator

The ICE-51 in-circuit emulator is used for software and hardware debugging and integration into the final product. The absolute object modules produced by RL51 can be loaded into the emulator's memory for execution. Refer to the ICE-51 manual listed in the preface for details.

SDK-51 System Design Kit

The SDK-51 system design kit for MCS-51 microcomputers is a useful tool for evaluation and simple program execution. However, the SDK-51 requires the hexadecimal object code format output by previous assemblers. For use with the SDK-51, the absolute object file must be processed by the OBJHEX code conversion program. Refer to the *ISIS-II User's Guide* for details on OBJHEX.

Keeping Track of Files

It is convenient to use the extensions of filenames to indicate the stage in the process represented by the contents of each file. Thus, source code files can use extensions like .SRC, .A51, or .P51 (indicating that the code is for input to ASM51 or PL/M-51). Object code files receive the extension .OBJ by default or the user can specify another extension. Executable files generally have no extension. Listing files can use .LST, the default extension given by the translator. RL51 uses .M51 for the default listing file extension (in order not to destroy the ASM51 listing file with the .LST extension).

Library files customarily have the extension .LIB.

Use caution with the extension .TMP, as many ISIS-II utilities (including RL51 and LIB51) create temporary files with this extension. These utilities will delete your file if it has the same name and extension as the temporary files they create.

CHAPTER 2
THE MECHANICS OF LINKAGE AND LOCATION WITH RL51

This chapter describes the operation of the RL51 program. Most of the process is transparent to the user; however, an understanding of the operation at the level presented here will help you to use the linking and locating controls in the RL51 invocation. More specific details on the allocating process appear in Chapter 3.

Major Functions

The RL51 program performs the following major functions:

1. Selects modules (including library processing)
2. Combines relocatable partial segments of the same name into a single segment
3. Allocates memory for the combined segments resulting from the previous step, and for all other complete relocatable segments from the input modules
4. Overlays data segments
5. Resolves external symbol references between the input modules
6. Binds relocatable addresses to absolute addresses
7. Produces an absolute object file
8. Produces a listing file consisting of a link summary, a symbol table, and an IXREF report
9. Detects and lists errors found in the input modules or in the RL51 command invocation

Functions 1, 2, 3, 5, and 6 are described in the remainder of this chapter. Functions 7, 8, and 9 are discussed in Chapter 4; the RL51 command invocation and overlaying of data segments are described in Chapter 3.

Selecting Modules

Input files are processed in the order they are specified in the invocation command.

The processing of an input file varies according to the content—that is, whether it is a library or non-library file. A non-library file may contain a concatenation of zero or more object modules. A library file contains zero or more object modules together with control information. A module in a non-library file is processed if it was explicitly listed in the module list, or if the module list was not specified at all (in other words, as if all modules were listed implicitly).

The processing of a library file is more complicated. If a module list was specified for the library file, then it is processed in the same manner as a non-library file. If a module list was not specified, then the library file is processed only if the previously processed modules contained at least one unresolved external. The library is scanned for modules containing public symbols that match as yet unresolved externals. Each such module is processed as if it were explicitly specified. The selection process continues until the modules in the library cannot satisfy any unresolved externals (including any externals encountered while processing modules from the library).

RL51 will report an error if the same module name is encountered more than once during the link process.

Take TRIGON.LIB as an (utterly fictitious) example. Assume it contains procedures called SINE, COSINE, TANGENT, ARCSINE, ARCCOS, ARCTAN, HYPERBOLIC_SINE, and HYPERBOLIC_COSINE.

When RL51 starts processing TRIGON.LIB, it has already made a first pass over all files that appear before it in the invocation line. If one of these contains a reference to the external SINE, and there is no public by that name, RL51 will assume that the procedure SINE from TRIGON.LIB is to be loaded. Otherwise, it will leave SINE alone for the moment.

If, while loading from TRIGON.LIB, RL51 encounters new externals that a module in the library can resolve, it will scan the library once more. Thus, there is no logical order among modules in a library; they are all equal. If TANGENT calls SINE and COSINE, and they are in the same library, in any order whatsoever, a reference to TANGENT will cause all three to be loaded.

Partial Segments

A segment is a unit of code memory or data memory. The portion of a segment defined in one module is called a partial segment. A partial segment has the following attributes (defined in the source module):

- Name. A relocatable segment has a name by which it is linked with other portions of the same segment from other modules. Absolute segments do not have names.
- Type. The type identifies the address space to which a segment belongs: CODE, XDATA, DATA, IDATA, or BIT.
- Relocatability. For relocatable segments only, this attribute describes any special constraints on relocation (PAGE, INPAGE, BLOCK, BITADDRESSABLE, or UNIT).
- Size. The size of the segment in bytes or bits, depending on the type.
- Base Address. The lowest address in the partial segment. For absolute segments, the base address is assigned at assembly time; for relocatable segments, it is assigned at location time.

Absolute segments are complete segments; they are taken as is into the output module. Relocatable segments are either defined by ASM51 users (using the SEGMENT directive in the source module) or automatically generated by the PL/M-51 compiler.

Refer to the *MCS-51 Macro Assembler User's Guide* for details on the assembler directives.

Combining Relocatable Segments

After processing the invocation command, RL51 performs a first pass over the input modules identified in the command. Pass 1 generates a segment table, a publics table, and an unresolved externals table. The segment table is discussed in this section; the other two tables are discussed later in this chapter.

The segment table contains the name, length, type, and relocation attribute of all combined segments from all modules. Combined segments are produced from the partial segments in the input modules according to the following rules:

- RL51 combines all partial segments with the same name into one relocatable segment. For example, if three input modules each have a partial relocatable segment named STACK, the segment table will have one segment named STACK that combines the length of the three partial segments.

- All the partial segments to be combined must be of the same type (CODE, DATA, IDATA, XDATA, or BIT). If any partial segments have the same names but different types, an error occurs.
- The length of the combined segment must not exceed the physical size of the memory type. Details on maximum size appear later in this chapter.
- The relocation attributes of all the partial segments to be combined must either be the same or UNIT-aligned combined with one other attribute. The combined segment receives the relocation attribute shared by the input partial segments, or, if the segments have attribute UNIT-aligned combined with one other attribute, the combined segment receives the more restrictive attribute.

  For example, if the three partial segments named STACK have relocation attributes UNIT, PAGE, and UNIT, the combined segment has attribute PAGE (i.e., page-aligned). Note that the relocation attribute is applied to the combined segment, not to each component segment. To continue the example, since the relocation is PAGE, the combined segment will start on a page boundary, but the component segments will be packed together without any gaps.

Allocating Memory for Segments

After the segment table is complete, RL51 can locate the segments within the memory spaces. Table 2-1 shows the address spaces used by MCS-51 processors, and the corresponding segment types.

The allocation process has a definite sequence; the exact order is presented in Chapter 3. As an overview, the process follows a general pattern of rules as follows:

1. Each of the types of memory space is allocated independent of the other spaces.
2. Within each space, absolute segments are allocated first, then segments specified within locating controls in the RL51 command, then other relocatable segments.
3. Because the on-chip data space represents three overlapping address spaces (DATA, IDATA, AND BIT), the general pattern in rule 2 is modified.
   a. Absolute BIT, DATA, and IDATA segments, and register banks are allocated first.
   b. Segments specified in PRECEDE and BIT controls are allocated next, then other relocatable BIT (and BIT-ADDRESSABLE) segments (following rule 2).
   c. DATA type segments are allocated next: segments in the DATA control first, then other relocatable DATA segments.
   d. IDATA type segments (except ?STACK) are allocated next; segments in the IDATA control first, then other relocatable IDATA segments.
   e. Segments specified in the STACK control are allocated, at as low an address as possible, provided that it is *above* all BIT, DATA, and IDATA segments allocated under (c) and (d).
   f. Last, the segment ?STACK, if it exists and is IDATA, and is not mentioned in an explicit location control, is now allocated, at as low an address as possible, provided that it is *above* all BIT, DATA, and IDATA segments allocated under (c) and (d) and (e).

In most cases, you do not need to use any explicit controls to obtain a satisfactory allocation of segments. RL51 tries to fit your segments into the designated memory spaces as best it can following the rules. As you can see, most of the complexity occurs in the on-chip data space.

Table 2-1. Address Spaces and Segment Types

Memory Space	Maximum Size	Addresses	Segment Type
Code	64K bytes	0000H - 0FFFFH	CODE
External data	64K bytes	0000H - 0FFFFH	XDATA
On-chip data (direct addressing)	128 bytes	00H - 7FH	DATA
On-chip data (indirect addressing)	256 bytes (see 1)	00H - 0FFH	IDATA
Bit space in on-chip data memory	128 bits (see 2)	00H - 7FH	BIT

1. The amount of indirectly addressable on-chip data memory is machine-dependent within the MCS-51 microcomputer family (see the discussion of RAMSIZE control in Chapter 3).
2. This bit space overlaps byte addresses 20H – 2FH in on-chip data memory.

Note: Addresses in the special function register memory (direct data addresses 80H – 0FFH, bit addresses 80H – 0FFH) cannot be relocated; they are always absolute. Thus, these addresses are not referenced in this table.

Rule (f) applies to PL/M-51. PL/M-51 produces for the stack an IDATA segment called ?STACK, whose size is 1. Although, by applying rule (f), RL51 makes the stack as big as possible, it is the user responsibility to ensure that the size of the stack is large enough (the segment map shows where the stack is located).

No rules for the allocation process can guarantee an optimal solution. If you are short of memory and RL51's first try is not satisfactory, you can place the segments in memory using the locating controls. Details on the locating controls are given in Chapter 3.

Overlaying Data Segments

On-chip RAM is a scarce resource on the MCS-51. To economize, the PL/M-51 compiler overlays data segments in the compiled module. RL-51 completes the work by overlaying the data segments across modules. This is accomplished by using the OVERLAY control. If RL-51 informed you about ignored segments due to lack of on-chip RAM, try this control. The use of OVERLAY is, in general, straightforward. However, for complex applications (for example, those with mixed ASM-51 and PL/M-51 modules), consult Chapter 3.

Resolving External References

An external reference points to a location in another module. The EXTERNAL declaration for symbols tells RL51 that the reference is to a location defined in another module. In the latter module, the symbol is declared PUBLIC so that external references to that symbol in other modules can be satisfied.

As it processes the input modules, RL51 builds a table of public symbols and unresolved external references. As each public symbol is added to the table, any external references to that symbol are deleted. After all segments have been located,

the public symbols are bound to absolute addresses. RL51 issues a warning for any unresolved externals that remain in the table.

External symbols and corresponding public symbols must be compatible. That is, both must be defined to address the same address space, or at least one must be defined as a typeless symbol (NUMBER); and if the symbol represents a PL/M-51 procedure name, then both must share the same register bank (i.e., must be declared within the PL/M-51 source modules with the same USING attribute).

Binding Relocatable Addresses

After allocating memory for the combined segments and binding the public symbols, RL51 makes a second pass (pass 2) through the input modules to build the listing file and fixup (i.e., bind to absolute addresses) any relocatable or external references. At this point, RL51 also processes debug records if requested, and performs fixups to any relocatable debug symbols that require processing to compute their absolute addresses.

CHAPTER 3
USING THE RL51 PROGRAM

Introduction

The RL51 program performs two functions for MCS-51 programs:

- The link function, combining a number of object modules specified in an input list into a single object module in an output file
- The locate function, assigning absolute addresses to any relocatable addresses in the input modules

This chapter explains how to enter commands, how to continue a long command onto more than one input line, how to enter comments in the invocation, and how to use abbreviations of the command words.

The chapter then presents a summary of the format of the RL51 invocation command, followed by details on the elements of the command with examples.

Command Entry, Continuation Lines, and Comments

The RL51 command is a standard ISIS-II file invocation. Terminate the command with the RETURN key. Note that the terminating carriage return is not shown in the command format notation.

Because of the many options available with the RL51 command, command lines can become very long. To break a command into several input lines, use the continuation character, an ampersand (&), before the RETURN to end intermediate lines of the command.

You can break a command between command words or other entries, but not in the middle of a word or parameter. The program begins a continuation line with a double asterisk (**) as a prompt.

The continuation feature is not shown in the format notation, but examples of continued commands occur in the discussions of command elements.

Any characters in a line following a semicolon character (;) or an ampersand are treated as a comment.

RL51 Command Format Summary

Here is a summary of the syntax of the RL51 invocation command. Refer to the Preface for an explanation of the command format notation.

The RL51 command has the overall format:

```
[:Fn:]RL51 input-list [TO output-file] [control-list]
```

where

input-list is a list of filenames separated by commas. The files named in *input-list* should contain the relocatable modules to be linked and located in the final absolute output module. For each file, you can additionally specify which modules are to be included.

output-file is the name of the file that is to receive the output module. If you omit this entry, the program will supply a default name based on the first filename in the input list.

control-list selects options for listing, linking, and locating the output. The listing controls specify what information is to be sent to the listing file, and the page width to be used. The linking controls specify the name of the output module, and determine what debug information is to be placed in the output file. The locating controls allow you to assign absolute addresses to relocatable segments, and to specify the order of relocatable segments within a given type of memory. The configuration control is used to describe the actual configuration the object is aimed to. The overlay control overlays data segments between modules.

The next several sections give details and examples of the elements of the RL51 command. Table 3-1 gives brief definitions of some of the terms used in the controls. A list of abbreviations for command words appears at the end of the chapter.

Table 3-1. Definitions of Common Terms

Term	Definition
name	Names can be from 1 to 40 characters in length and must be composed of letters A - Z, digits 0 - 9, or special characters (?, @, _). The first character must be a letter or a special character.
module-name	Same as name.
segment-name	Same as name.
pathname	A valid ISIS-II filename reference or device reference. See next two items for examples.
filename	A reference to a disk file. The format is [:F*n*:]*root* [.*ext*] Examples: PROG1, :F1:SAMPL1, TEST.HEX, :F2:SAMPLE.OBJ
device	A reference to a non-disk device. Examples: :LP:, :CO:, :TO:
value	A 16-bit unsigned integer. Examples: 1011B, 304Q, 4096D (or just 4096), 0C300H
address	Same as value.

Input List

The input list tells RL51 what files are to be processed. The files must be disk files containing relocatable object modules as described in Chapter 2.

The entry for each file in the list can include the following information:

- The drive number. If the drive number is omitted, drive 0 is assumed as the default.
- The filename. The filename is the name of the object file including an extension if one exists.
- A list of modules enclosed in parentheses. If a module list is provided, only the modules in the list are linked into the output file, and modules not in the list are ignored. If no module list is provided, the default for a non-library file is to link

all modules in the file into the output module. The default for a library file is to link only those modules that satisfy previously declared external symbols (see the exact process in Chapter 2 under "Selecting Modules").

If a module named in the module list is not present in the file, the system issues an error message but does not halt the link process.

Module names (specified explicitly or implicitly) must be unique throughout the entire application.

Examples

```
RL51 :F1:PROG.OBJ TO :F1:PROG.ABS
```

In this example, the input list has one file (PROG.OBJ on drive 1); RL51 links all the modules in this file into the output file (PROG.ABS). (This and other examples omit the drive number on the RL51 reference for clarity; i.e., the examples assume drive 0.)

```
RL51 :F1:SAMP1.OBJ, :F1:SAMP2.OBJ, :F1:SAMP3.OBJ &
**TO :F2:SAMP.ABS
```

In this example, the input list has three files. RL51 links all the modules in each of these files into the output file. (Note that the ** in the second line of the example is generated by the system in response to the continuation character & on the first line of the example).

```
RL51 :F1:PROG1.OBJ (MOD1, MOD3), :F1:PROG2.OBJ (MOD2) &
**TO :F1:PROG3.ABS
```

Here, the input list has two input files (PROG1.OBJ and PROG2.OBJ). From PROG1.OBJ, only the modules named MOD1 and MOD3 are to be linked into the output file; any other modules in file PROG1.OBJ are ignored by RL51. From PROG2.OBJ, only the module named MOD2 is to be linked.

```
RL51 :F1:PLMPRG.OBJ, :F1:UTIL51.LIB, :F1:I051.LIB, PLM51.LIB
```

The example introduces a typical linking using libraries. Here, PLMPRG is linked with two private libraries and with the mandatory library PLM51.LIB (which must be used if modules generated by PLM51 participate in the linkage).

```
RL51 :F1:EXAMPL.OBJ, COTRIG.LIB, TRIG.LIB, COTRIG.LIB
```

Interaction between libraries (i.e., libraries that reference each other) may sometimes require the same library to be mentioned twice in the input list.

In the preceding example, COTRIG.LIB contains the COTANGENT and COSINE trigonometric functions, TRIG.LIB contains the SINE and TANGENT function, and EXAMPL.OBJ references the COTANGENT function.

Since COTANGENT = 1/TANGENT, TRIG.LIB must be specified in order to resolve the reference to the TANGENT function. Also, since TANGENT = SINE/COSINE, COTRIG.LIB must be respecified in order to resolve the reference to the COSINE function.

Output File

The output filename is the name of the disk file that is to receive the absolute object module.

If the output file name is omitted, RL51 creates a filename for the output file by removing the extension from the first filename in the input list and using the drive and root name only. If this input file contains no extension, a fatal error occurs. For example, the command:

```
RL51 PROG1
```

is illegal since the output filename defaults to PROG1.

If there is already a file on the target drive with the name of the output file, that file is overwritten by the new output file.

Examples

```
RL51 :F1:PROG.OBJ TO :F1:PROG
```

This example specifies file PROG on drive 1 as the output file.

```
RL51 :F1:PROG.OBJ
```

This example uses the default output file generated by RL51. The effect is the same as the first example; the output file becomes :F1:PROG.

```
RL51 :F1:SAMPLE1.OBJ, :F1:SAMPLE2.OBJ TO :F2:SAMPL.ABS
```

In this example, the output file is on a different drive from the input files, and both the filename SAMPL and the extension .ABS are specified.

Controls

After the output filename, you can add a list of controls to select options for listing, linking, and locating the output. Use blanks (not commas) to separate controls in the list. The same control may not appear more than once in the list; if a duplicate control is encountered, a fatal error results and the program aborts. The next several sections explain the controls and give examples.

Listing Controls

The listing file output by RL51 can contain a link summary, a symbol table, an IXREF report, and a list of error messages. The link summary can contain a memory map of the linked segments.

The listing controls are the PRINT option, the PAGEWIDTH control, the MAP option, the SYMBOLS option, the PUBLICS option, the LINES option, and the IXREF option. These controls allow you to specify the file or device to receive the output listing, to omit the listing file altogether, to omit the map from the link summary, or to omit local symbols, public symbols, or line numbers from the symbol table. You may also specify if you wish to have the IXREF report generated, and the specific page width to be used.

NOTE

The information in the listing file is taken from the input object modules. If these are generated without the DEBUG option, the SYMBOLS, PUBLICS, and LINES information will not be available for listing.

PRINT/NOPRINT

The print options control the destination of the list file.

To direct the list file to a disk file, the print control format is

```
PRINT (([:Fn:]filename[.ext] )
```

Example

```
RL51 :F1:SAMPLE1.OBJ &
**PRINT (:F1:SAMPLE.LST)
```

To direct the list file to a device other than a disk file, the print control format is

```
PRINT (:device:)
```

where

device is an ISIS-II device code. Common devices are CO (console), LP (line printer), TO (terminal other than console), and VO (video terminal screen).

If you omit the print control, or if you enter the command word PRINT without a filename or device name, RL51 creates a disk file for the listing. The name of the default listing file has the same root as the output filename and has an extension of M51; the drive number is also the one used in the output filename.

Example

```
RL51 :F1:PROG.OBJ, :F1:PROG1.OBJ TO :F2:PROG2.ABS
```

Since this command does not specify a listing file destination, the system creates a default file named PROG2.M51 on drive 2.

The output listing filename may not be the same as the output filename or any of the filenames in the input list. If the listing file duplicates an input or output filename, a fatal error results. If the listing filename already exists on the target drive, the old file with that name is overwritten by the new listing file.

The NOPRINT option specifies that no output listing file is to be produced. NOPRINT overrides the MAP, SYMBOLS, PUBLICS, LINES and IXREF controls.

PAGEWIDTH

The PAGEWIDTH control specifies the maximum number of columns per line in the print output file. The control takes the form

```
PAGEWIDTH(width)
```

where

width is an unsigned number which specifies the maximum page width to be used.

The allowable range for *width* is 72 to 132. The default PAGEWIDTH is 78.

Listing Switches

The MAP, SYMBOLS, PUBLICS, LINES and IXREF controls select what portions of the listing files are to be generated. The default of any switch (with the exception of IXREF) is the positive form (MAP, SYMBOLS, PUBLICS, and LINES). Table 3-2 summarizes the listing switches.

IXREF/NOIXREF

This control specifies whether or not to produce the inter-module cross reference report. If IXREF is specified, the report is appended to the print file.

A selection list may be added to the positive form (only) of the IXREF control. A selection list causes RL51 to output or suppress output of various selected entries to the IXREF report. An entry consists of a symbol and a module where this symbol is referenced (either as public or as external). The general form of the IXREF control is

```
IXREF [(selection-item [,...])]
```

where

selection-item is either (NO)GENERATED or (NO)LIBRARIES. If IXREF is specified and any of the selection items are omitted, the missing selection item assumes its positive form. A selection item may appear at most once.

The selection-items are best explained by describing the effect of their negative form.

Table 3-2. Listing Switches

Switch	Effect
MAP	Output memory map to link summary
NOMAP	Suppress memory map
SYMBOLS	Output local symbols to symbol table
NOSYMBOLS	Suppress local symbols
PUBLICS	Output public symbols to symbol table
NOPUBLICS	Suppress public symbols
LINES	Output line numbers to symbol table (high-level language translators only)
NOLINES	Suppress line numbers
IXREF	Append intermodule cross-reference report to print file
NOIXREF	Suppress the intermodule cross-reference report

The NOGENERATED control causes RL51 to surpress output of entries whose symbol name begins with a question mark (?); such symbols are usually PL/M-51 generated symbols. The GENERATED form of the control causes RL51 to output such entries also.

The NOLIBRARIES control causes RL51 to surpress output of entries whose module resides within a library. The LIBRARIES form of the control causes RL51 to include all libraries in the IXREF report.

The selection list is used to control the number of entries collected for the IXREF report. This is needed when an excessive number of IXREF entries make it impossible for RL51 to generate the IXREF report.

Examples

Because the default for any listing switch (except IXREF) is the positive form, the main use of the switches is to suppress unwanted information. To suppress the entire symbol table, for example, the command would be

```
RL51 :F1:PROG.OBJ NOSYMBOLS NOPUBLICS NOLINES
```

As another example, to see only the public symbols (no map or other symbols or lines), the command is

```
RL51 :F1:PROG.OBJ PRINT (:F2:PROG.M51) NOMAP NOSB NOLI
```

Note the use of abbreviations (NOSB for NOSYMBOLS and NOLI for NOLINES) to save keystrokes. A complete list of abbreviated forms appears at the end of this chapter. Note that the blank separating PRINT from its parameters is optional; you could also use PRINT(:F2:PROG.M51).

In order to suppress generated symbols from the IXREF report, the command is

```
RL51 :F1:PROG.OBJ, :F1:PROCS.OBJ, :F1:PLM51.LIB IXREF(NOGN)
```

Using the NOGN (NOGENERATED) selection item prevents PL/M-51 run-time library procedures from being written to the IXREF report.

Linking Controls

The linking controls allow you to name the resultant output module and to specify which debug information is to be copied to the output module.

NOTE

In order to obtain the debug information (SYMBOLS, PUBLICS, or LINES), the DEBUG control must be included in the invocation line for the translator used to produce the input modules.

NAME

The NAME control allows you to name the output module. The format is

```
NAME (module-name)
```

If the NAME control is not used, the output module-name defaults to the name of the first input module processed.

Example

```
:F1:RL51 :F1:SAMPL1.OBJ, :F1:SAMPL2.OBJ TO :F1:SAMPLE.ABS &
**NAME(SAMPLE_PROGRAM)
```

In this example, the name SAMPLE_PROGRAM is assigned to the output module. Note that the blank between NAME and its parameter is optional and can be omitted as shown in the example.

Linking Switches

The DEBUGSYMBOLS, DEBUGPUBLICS, and DEBUGLINES controls select what kinds of debug information are to be included in the output file. The default of any switch is always the positive form (DEBUGSYMBOLS, DEBUGPUBLICS, and DEBUGLINES). Table 3-3 summarizes the linking switches.

Examples

Because the linking switches default to the positive form, you will usually use the negative forms to suppress unwanted debug information in the output file. For example, to cause the output file debug information to contain only the information for the public symbols, use

```
RL51 :F1:PROG1.OBJ NODEBUGSYMBOLS NODEBUGLINES
```

To have only the local symbols output to the absolute file, use

```
RL51 :F1:PROG1.OBJ NODP NODL
```

Note the use of abbreviations (NODP for NODEBUGPUBLICS and NODL for NODEBUGLINES). A complete list of abbreviations for command words appears at the end of this chapter.

Table 3-3. Linking Switches

Switch	Effect
DEBUGSYMBOLS	Copies local symbol information to output file
NODEBUGSYMBOLS	Suppresses local symbols
DEBUGPUBLICS	Copies public symbol information to output file
NODEBUGPUBLICS	Suppresses public symbols
DEBUGLINES	Copies line number information (high-level language translators only) to output file
NODEBUGLINES	Suppresses line numbers

Locating Controls

The locating controls allow you to assign absolute addresses to relocatable segments, to specify the ordering of relocatable segments of a given type in memory, and to force allocation of segments into a specific range of addresses.

Allocation Sequence

The system allocates memory in accordance with segment attributes and locating controls, using a fixed order of precedence. The precedence of the allocating operations (grouped by type of memory space) is as follows:

Internal Data Space:

- Absolute BIT, DATA, and IDATA segments, and register banks
- Segments specified in a PRECEDE control in the RL51 command
- Segments specified in a BIT control in the RL51 command
- DATA type segments with relocation equal to BIT-ADDRESSABLE
- Other relocatable bit segments
- Segments specified in a DATA control in the RL51 command
- DATA type segments with relocation equal to UNIT-aligned
- Segments specified in an IDATA control in the RL51 command
- Other relocatable IDATA segments, except ?STACK
- Segments specified in a STACK control in the RL51 command
- ?STACK, if it is IDATA and has not been specified in any other locate control

External Data Space:

- Absolute external data segments
- Segments specified in an XDATA control in the RL51 command
- Other relocatable external data segments

Code Space:

- Absolute code segments
- Segments specified in a CODE control in the RL51 command
- Other relocatable code segments

NOTE

In most cases, the allocation algorithm will produce a workable solution without requiring the user to enter any locating controls in the RL51 command. These controls are intended for the experienced user, in cases where running RL51 without them does not give a good enough result.

Format Summary

The locating controls have the format

control **(** *segment* **[, . . .])**

where

segment **:=** *segment-name* **[(** *base-address* **)]**

The segments specified in the locating controls are allocated in the order they appear; the first segment is assigned the lowest possible address, and succeeding segments receive higher and higher addresses.

The user has the option of specifying the base address of any or all segments. Segments with specified base addresses must appear in the list in ascending numerical order. Segments named in a locating control with a specific base address are allocated at that address irrespective of segment overlap or segment type contradiction, as long as ascending order is maintained. Base addresses are byte addresses except for the BIT locating control, where addresses are bit addresses in the bit space (0 to 127).

Table of Locating Controls

Table 3-4 lists the locating controls in order of precedence. The first column gives the name of the control. The second column describes the address space affected by the control. The third column gives the address range for segments within each control. The last column shows what types of segments are allowed for each control; for each valid type, the column also shows the allowable relocation attributes. (Refer to the *MCS-51 Macro Assembler User's Guide* and *PL/M-51 User's Guide* for details on segment types and relocation attributes.)

Notes On Locating Controls

The following notes refer to table 3-4.

1. Bit addresses for non-BIT segments in the BIT control must be on byte boundaries; that is, they must be divisible by eight. (BIT-type segments can be aligned on bit boundaries.)
2. The range of addresses for the IDATA control is dependent on the target machine. See the RAMSIZE control later in this chapter.
3. The STACK control specifies which segments are to be allocated uppermost in the IDATA space. The memory accessed starts after the highest on-chip RAM address occupied by any previously allocated segment and continues to the top of the IDATA space.

Table 3-4. Locating Controls

Control	Address Space	Address Range (Hex)	Segment Types (and Attributes)
PRECEDE	Register banks and bit-addressable space in on-chip data RAM	00H-2FH	DATA (UNIT-aligned); IDATA
BIT	Bit-addressable space in on-chip data RAM	00H - 7FH (see note 1)	BIT; DATA; IDATA
DATA	Directly-addressable on-chip data RAM	00H - 7FH	DATA (UNIT-aligned); IDATA
IDATA	Indirectly-addressable on-chip data RAM	00H - 0FFH (see note 2)	IDATA
STACK	Same as IDATA (see note 3)	Same as IDATA	Same as IDATA
XDATA	External data RAM	0 - 0FFFFH	XDATA
CODE	Code memory	0 - 0FFFFH	CODE

NOTE

This control has no other effect on any segments.

The IDATA ?STACK segment, if it exists, is placed higher than segments that were mentioned in the STACK control.

The STACK control provides a convenient way to handle the stack (usually for ASM51-based application, where ?STACK is not used).

First, assign the stack pointer (SP) to a relocatable segment; consider the following ASM51 example:

```
STACK_AREA SEGMENT IDATA        ; SEGMENT directive in source.
     DS         10H             ; Reserve 16 bytes for stack.

     ........                    ; Other CODE instructions.

     MOV        SP, #STACK_AREA-1    ; Initialize SP.
```

Then, at relocation time, specify the segment named STACK_AREA in a STACK locating control:

```
RL51 ... STACK (STACK_AREA)
```

where

ellipsis (...) represents the rest of the invocation line exclusive of the STACK control.

NOTE

If the application contains modules produced by PL/M-51, the ?STACK should be used as the stack segment.

Examples

Here are three brief examples of invocations with locating controls. See Chapter 5 for a more extended example.

```
RL51 :F1:PROG1.OBJ, :F1:PROG2.OBJ TO :F2:PROG.ABS &
**PRECEDE (MESSAGE1) XDATA (ARRAY1 (256), ARRAY2 (512))
```

In this example, the DATA (or IDATA) segment named MESSAGE1 will be allocated space in on-chip RAM in the lowest available location, overlapping the BIT space if necessary. The XDATA control specifies that the two arrays are to be located at specific addresses (e.g., for debugging).

```
RL51 :F1:TEST.OBJ STACK (STACK_AREA)
```

Here, the STACK control allocates the uppermost portion of IDATA space for the segment named STACK_AREA. The software definition of STACK_AREA might be as given in the previous section.

```
RL51 APROG.OBJ, BPROG.OBJ, PLM51.LIB CODE (MOD1 (4000H),
                                           MOD2, MOD3)
```

Here, the CODE control allocates space in code memory for segments MOD1, MOD2, and MOD3. MOD1 is aligned at location 4000H. MOD2 and MOD3 are assigned contiguous addresses after MOD1.

Configuration Controls

The configuration controls are used to describe the actual configurations that objects are aimed to.

This group contains the RAMSIZE control.

RAMSIZE

The RAMSIZE control format

```
RAMSIZE(value)
```

where

value is a number in the range 128 to 255.

RAMSIZE specifies the maximum amount of on-chip RAM that may be allocated for the user program. The default value for RAMSIZE is 128 (as is the case for the 8051). If the object is aimed at more than one configuration of the MCS-51 family, specify the MINIMUM of all on-chip RAM sizes among all machines you want to link.

The sole use of this control is to enable RL51 to check on-chip memory size constraints at RL-time and thus avoid confusion at ICE-time.

OVERLAY/NOOVERLAY Controls

The linker allows overlaying of on-chip RAM segments among modules, under the specification of the OVERLAY control. Two segments can be overlaid if all the following conditions exist:

- The segments have the same type (DATA, IDATA, BIT, or BITADDRESSABLE).
- The segments use the same register bank (determined by the USING attribute or the REGISTERBANK control).
- The segments are marked as overlayable. Currently, this is done only by the PL/M-51 compiler. ASM51 (V2.1 and lower) lacks this feature. Therefore, assembler segments are considered non-overlayable.
- The segments belong to disjoint modules. That is, no procedure in one module can directly or indirectly call a procedure from the other.

The default is NOOVERLAY. No overlaying of on-chip RAM segments is done by the linker.

The general form of the OVERLAY control is as follows:

```
OVERLAY [(overlay-unit [, ...])]
```

where

overlay-unit is *ov-module-name* calls *ov-module-name*.

ov-module-name is a legal RL51 module name or *, which stands for all the module names.

calls is >.

OVERLAY

If the OVERLAY control appears in the invocation line without arguments, the linker assumes that no intra-module calls exist except for those deducible from the PUBLIC-EXTERNAL declarations, and that overlaying of all overlayable segments is safe.

NOOVERLAY

The linker does not overlay data segments.

OVERLAY (A > B)

If the OVERLAY control appears in the invocation line with arguments, it indicates that there are invisible calls between modules. The notation A > B means that module A calls module B. In this case, the linker overlays all overlayable segments, except that segments from A are not overlaid by segments from B. Note that the added connection can prevent other segments from overlaying. For example, if the segment A was overlaid with the segment D, and B calls D (visibly by PUBLIC-EXTERNAL declarations), then the effect of A > B is that A and D will not be overlaid, since A can call D through B.

OVERLAY (A > *, * > B)

A module can be declared as non-overlayable in two ways. The argument A > * indicates that the module A calls all other modules. On the other hand, * > A means every module calls A. In either case, no segments from A will be overlaid. The effect of each form depends on the nature of A. For example, if the * > A form is used and A visibly calls all other modules, then every module can call (through A) each other module. In this case, the linker will not perform any overlays.

The overlaying of data segments in on-chip RAM has the following restrictions:

- The OVERLAY control cannot be invoked with the IXREF selection items NOGENERATED or NOLIBRARIES. RL51 generates an error if either one is specified.
- Combined segments and segments appearing in locating controls are not overlaid by the linker.

Following is an example in which two disjoint modules share the same on-chip RAM area:

```
mod1: DO;

      THREE_BEARS: PROCEDURE PUBLIC;
            DECLARE LITTLE_BEARS_BED BYTE;
            IF BOOLEAN (LITTLE_BEARS_BED) THEN
                  CALL MSG(.('SOMEONE''S BEEN IN MY BED!'),0);
            LITTLE_BEARS_BED = 0;
      END THREE_BEARS;

END mod1;
```

```
mod2: DO;

      GOLDILOCKS: PROCEDURE PUBLIC;
            DECLARE SPARE_BED BYTE;
            SPARE_BED = 1;
      END GOLDILOCKS;

END mod2;

main_story: DO;

      THREE_BEARS: PROCEDURE EXTERNAL; END;
      GOLDILOCKS: PROCEDURE EXTERNAL; END;

      CALL THREE_BEARS;
      CALL GOLDILOCKS;
      CALL THREE_BEARS;

END main_story;
```

In this example, the linker reserves the right to use the LITTLE_BEARS_BED as a SPARE_BED because the two procedures are never active simultaneously.

To perform overlaying, the linker must determine which procedures are active simultaneously. To do this, the linker assumes that all CALLs can be executed. For example, if procedure A calls procedure B, and B calls procedures C and D, then the linker can overlay RAM variables from C only with the RAM variables of D.

The linker, however, looks only at the PUBLIC-EXTERNAL declarations. It assumes that any reference to an EXTERNAL procedure will be executed, but ignores the possibility of hidden calls. The arguments to the OVERLAY control are therefore needed to specify those interconnections between modules that cannot otherwise be detected by the linker.

Such situations arise if the interconnection is done by a computed call to an external procedure whose address is not determined by a simple PUBLIC-EXTERNAL relationship. For example, module A imports from module B a public variable that contains the address of a local or public procedure in B. Module A then performs a computed call to the procedure in B. The rule can be stated as follows: The linker assumes a connection from module A to module B if there exists an external reference in A to a public procedure in B. In all other cases, hidden connections must be explicitly given as arguments to the OVERLAY control.

Following is an example of a computed call to an external procedure:

```
MOD1: DO;

      DECLARE I_O_CLEAR WORD EXTERNAL;
      . . .
      CALL I_O_CLEAR;
      . . .
END MOD1;
```

In another module, you have:

```
MOD2: DO;

      DECLARE I_O_CLEAR WORD PUBLIC;

      READER: PROCEDURE;

            I_O_ERROR: PROCEDURE;
            . . .
            END I_O_ERROR;

            I_O_SUCCESS: PROCEDURE;
            . . .
            END I_O_SUCCESS;

            IF ERR_CODE <> 0
                  THEN I_O_CLEAR = .I_O_ERROR;
                  ELSE I_O_CLEAR = .I_O_SUCCESS;
            . . .

      END READER;

END MOD2;
```

In the above procedure, MOD1 invokes a procedure defined in MOD2. To prevent the linker from overlaying on-chip RAM variables of MOD2 with on-chip RAM variables of MOD1, the following form of the OVERLAY control must be used:

```
OVERLAY (MOD1 > MOD2)
```

Overlaying can be a good way of economizing on-chip RAM space; however, overlaying may, in some cases, give worse results. For example, if most procedures call one another, the resulting segments will expand, making it more difficult for the linker to allocate a few large segments than many small ones.

The outcome of the overlaying process can be checked by inspecting the link map. All overlaid segments are indicated by **OVERLAP**. Warning (4), DATA SPACE MEMORY OVERLAP, is not generated for those segments.

Abbreviations for Command Words

Most of the command words in the RL51 command have short forms to save you keystrokes over the full spellings. Here is a list of the command words and their abbreviations.

Command Word	Abbreviation
BIT	BI
CODE	CO
DATA	DT
DEBUGLINES	DL
DEBUGPUBLICS	DP
DEBUGSYMBOLS	DS
GENERATED	GN
IDATA	ID
IXREF	IX

Command Word	Abbreviation
LIBRARIES	LB
LINES	LI
MAP	MA
NAME	NA
NODEBUGLINES	NODL
NODEBUGPUBLICS	NODP
NODEBUGSYMBOLS	NODS
NOGENERATED	NOGN
NOIXREF	NOIX
NOLIBRARIES	NOLB
NOLINES	NOLI
NOMAP	NOMA
NOOVERLAY	NOOL
NOPRINT	NOPR
NOPUBLICS	NOPL
NOSYMBOLS	NOSB
OVERLAY	OL
PAGEWIDTH	PW
PRECEDE	PC
PRINT	PR
PUBLICS	PL
RAMSIZE	RS

CHAPTER 4
RL51 OUTPUTS

The RL51 program produces three outputs: console displays, a listing file, and the absolute object module file. This chapter describes these outputs and gives examples. As discussed in Chapter 3, the listing controls in the RL51 command allow the user to suppress some information in the listing file, and the linking controls can suppress some information in the absolute object file.

Console Display

The console displays produced by RL51 consist of a sign-on message and any error messages that occur. The sign-on is as follows:

```
ISIS-II MCS-51 RELOCATOR AND LINKER Vx.y
```

where

x.y is the version number.

Listing File

RL51 produces a listing file unless it is suppressed in the RL51 invocation. The RL51 listing file contains:

- A summary of the link and locate process
- A symbol table, as specified in the RL51 invocation
- An inter-module cross-reference listing (IXREF)
- Error messages detected by RL51

Link Summary

A sample of a link summary is shown in figure 4-1. The summary includes the following kinds of information:

- A header echoing the RL51 invocation.
- Input modules included in the link process. Input modules are identified by module name and file name.
- A link map (unless suppressed by the NOMAP control). The map lists all allocated segments, giving the type, base address, and length of each segment. The map also identifies segment overlaps and gaps in the memory space.
- A list of segments that were ignored in the link process. If any segments were ignored, the reasons for doing so will be reported later as an error.
- A list of unresolved external symbols. An external symbol is unresolved when it is not matched by a public symbol in one of the input modules. Each occurrence of an unresolved external symbol in a module will be reported later as an error.
- A list of all symbols that were ignored in the locate process. A symbol is ignored when the same name appears as a public symbol in different modules, or has attributes that are incompatible with external references, or belongs to an ignored segment. Each occurrence of an ignored symbol in a module will be reported later as an error.

```
ISIS-II MCS-51 RELOCATOR AND LINKER, V1.0, INVOKED BY:
RL51 :F1:FILE1.EXT(MOD1,MOD2), :F1:FILE2.EXT TO OUTFIL.EXT &
NAME (EXAMPLE) MAP PRINT (:LP:)

INPUT MODULES INCLUDED
  FILE1.EXT(MOD1)
  FILE1.EXT(MOD2)
  FILE2.EXT(MOD3)

LINK MAP FOR OUTFIL.EXT(EXAMPLE)

             TYPE    BASE        LENGTH     RELOCATION   SEGMENT NAME
             ----    ----        ------     ----------   ------------
             REG     0000H       0008H                   "REG BANK 0"
             DATA    0008H       0010H      UNIT         DATA_SEG_1
             DATA    0014H       0008H      ABSOLUTE
**OVERLAP**  REG     0018H       0008H                   "REG BANK 3"
             BIT     0020H       0001H.6    UNIT         A_BIT_SEG
                     0021H.6     0000H.2                 ***GAP***
             DATA    0022H       0001H      BITADDR      DATA_SEG_2
             DATA    0023H       000BH      ABSOLUTE
             IDATA   002EH       0042H      UNIT         STACK_SEG
                     0070H       0010H                   ***GAP***

             XDATA   0000H       C000H      UNIT         DYNAMIC_MEM

             CODE    0000H       1389H      UNIT         PROC1
                     1389H       0477H                   ***GAP***
             CODE    1800H       07A5H      INBLOCK      PROC2

IGNORED SEGMENTS
  DYNAMIC POOL

UNRESOLVED EXTERNAL SYMBOLS
  INVERT

IGNORED SYMBOLS
  BIT256
```

Figure 4-1. Link Summary

NOTE

1. For bit addresses, the display format is *byte-address.bit-address* (example: 0020H.7 for bit 7 of byte 0020H). However, when bit 0 of a byte is referenced, only the byte address is displayed (the .0 is not displayed).
2. References to an unresolved external symbol, an external symbol referring to an ignored public symbol, or a reference to an ignored segment will produce additional error messages.

Symbol Table

The listing file contains a symbol table as specified by the SYMBOLS, PUBLICS, and LINES controls in the RL51 invocation. A sample symbol table is shown in figure 4-2.

```
SYMBOL TABLE FOR OUTFILE.EXT(EXAMPLE)

VALUE                   TYPE                    NAME
-----                   ----                    ----

-------                 MODULE                  MEMRY
D:0032H                 PUBLIC                  LOW_MEM_PTR
B:0020H                 PUBLIC                  INIT_FLAG
B:0020H.1               PUBLIC                  FULL_FLAG
D:0034H                 PUBLIC                  HIGH_MEM_PTR
X:0000H                 PUBLIC                  DYNAMIC_MEMORY
-------                 PROC                    ALLOCATE
D:0064H                 SYMBOL                  NUM_BYTES
D:0066H                 SYMBOL                  POOL_SELECTOR
D:0068H                 SYMBOL                  ALLOC_PTR
B:0020H.2               SYMBOL                  FLAG
C:0000H                 LINE#                   19
C:007H                  LINE#                   20
C:0010H                 LINE#                   21
C:0013H                 LINE#                   22
------                  DO
D:006AH                 SYMBOL                  I
C:0018H                 LINE#                   23
C:0021H                 LINE#                   24
C:0028H                 LINE#                   25
C:002FH                 LINE#                   26
C:0032H                 LINE#                   27
-------                 ENDDO
C:0037H                 LINE#                   28
C:0040H                 LINE#                   29
C:004FH                 LINE#                   30
C:0057H                 LINE#                   31
C:005FH                 LINE#                   32
C:0068H                 LINE#                   33
C:006FH                 LINE#                   34
C:0076H                 LINE#                   35
C:0082H                 LINE#                   36
C:008FH                 LINE#                   37
C:0094H                 LINE#                   38
-------                 ENDPROC                 ALLOCATE
-------                 ENDMOD                  MEMRY
```

Figure 4-2. Symbol Table

NOTE

The information in the listing file is taken from the input object modules. If these are generated without the DEBUG option, the SYMBOLS, PUBLICS, and LINES information will not be available for listing.

The symbol table contains scope definitions and information about the symbols and line numbers. Scope definition identifies the module, DO block or procedure that contains the symbol or line number. Note that when the table contains only public symbols (i.e., NOSYMBOLS and NOLINES controls are in effect), scope definition is by module only.

Each entry in the table consists of three parts, as follows:

- VALUE. The value is the absolute address of the symbol. The address is prefixed with a letter indicating the type of address space (C, code; D, internal data; I, indirect internal data; B, bit space; X, external data; N, typeless number). A byte address (or a bit address on a byte boundary) is shown as a four-digit hexadecimal number (example: 00E0H). A bit address (unless it is on a byte boundary) is shown as a byte address followed by a period and the bit offset (1 through 7) into the byte.
- TYPE. The type field identifies the entry as a local symbol (SYMBOL), a public symbol (PUBLIC), segment (SEGMENT), or a line number (LINE#).
- NAME. The name field gives the name of the symbol, or the number of the line.

For scope definition, a line is printed for the beginning and end of each block. The TYPE field shows the type of block (MODULE, DO, or PROC for PROCEDURE), and the end of each block (ENDMOD, ENDDO, ENDPROC). The NAME field shows the name of the block, if any.

NOTE

Line number information and scope definitions other than MODULE are applicable only to object files produced by high-level language translators (e.g., PL/M-51).

Inter-Module Cross-Reference Report (IXREF)

The listing file contains an IXREF report as specified by the IXREF control and its associated selection list in the RL51 invocation. A sample IXREF report is shown in figure 4-3.

The IXREF report consists of an alphabetically sorted list of symbols. Each such symbol begins a new line and represents a symbol that was declared as PUBLIC or EXTERNAL in at least one of the input modules. Each symbol is followed by its corresponding address space, followed by a semicolon. To the right of the semicolon starts a list of modules in which the symbol was declared PUBLIC or EXTERNAL. The first module name in the list is the one in which the symbol was declared PUBLIC. If a symbol is unresolved, or if a symbol is defined in a library and the NOLIBRARIES selection item is in effect, then the string ** UNRESOLVED ** appears in front of the modules list.

Error Messages

RL51 displays error messages on the console and copies them to the end of the listing file unless the listing file is suppressed.

RL51 error messages describe warnings, errors, and fatal errors. A warning is a detected condition that may or may not be what the user desired; a warning does not terminate the link/locate operation. An error does not terminate operation, but probably results in an output module that cannot be used. A fatal error terminates operation of RL51.

Refer to Appendix B for a list of the error messages and probable causes.

```
INTER-MODULE CROSS-REFERENCE LISTING
------------------------------------

NAME. . . . . . . . USAGE    MODULE NAMES
-----------------------------------------

?CHECK_EQ?BYTE. . . DATA;    CHKEQ   TESBAS
?CHECK_EQ_BITS?BIT. BIT;     CHKEQ
?P0008. . . . . . . CODE;    ?P0008  TESBAS
?P0015. . . . . . . CODE;    ?P0015  TESBAS
?P0016. . . . . . . CODE;    ?P0016  TESBAS
?PIHOR. . . . . . . CODE;    TESBAS  ?PIVOR
?PIVOR. . . . . . . CODE;    ?PIVOR  TESBAS
?PSWOR. . . . . . . NUMB;    TESBAS  ?PIVOR
CHECK_EQ. . . . . . CODE;    CHKEQ   TESBAS
CHECK_EQ_BITS . . . CODE;    CHKEQ
CHECK_EXIT. . . . . CODE;    CHKEQ
CHECK_INIT. . . . . CODE;    CHKEQ
PUB00 . . . . . . . CODE;    MODULE_0  MODULE_1  MODULE_2
PUB01 . . . . . . . CODE;    MODULE_0  MODULE_1  MODULE_2
PUB02 . . . . . . . CODE;    MODULE_0  MODULE_1  MODULE_2
PUB03 . . . . . . . CODE;    ** UNRESOLVED **  MODULE_1  MODULE_2
PUB04 . . . . . . . CODE;    MODULE_0  MODULE_1  MODULE_2
PUB05 . . . . . . . CODE;    MODULE_0  MODULE_1  MODULE_2
PUB06 . . . . . . . CODE;    ** UNRESOLVED **  MODULE_1  MODULE_2
PUB07 . . . . . . . CODE;    MODULE_0  MODULE_1  MODULE_2
PUB08 . . . . . . . CODE;    MODULE_0  MODULE_1  MODULE_2
PUB09 . . . . . . . CODE;    MODULE_0  MODULE_1  MODULE_2
PUB10 . . . . . . . CODE;    MODULE_0
PUB11 . . . . . . . XDATA;   MODULE_0
PUB12 . . . . . . . DATA;    MODULE_0
PUB13 . . . . . . . IDATA;   MODULE_0
PUB14 . . . . . . . BIT;     MODULE_0
PUB15 . . . . . . . NUMB;    MODULE_0
PUB16 . . . . . . . CODE;    MODULE_0
PUB17 . . . . . . . CODE;    MODULE_0
PUB18 . . . . . . . CODE;    MODULE_0
PUB19 . . . . . . . CODE;    MODULE_0
PUBX0 . . . . . . . CODE;    MODULE_1  MODULE_0
PUBX1 . . . . . . . CODE;    MODULE_1  MODULE_0
PUBY0 . . . . . . . CODE;    ** UNRESOLVED **  MODULE_1  MODULE_2
PUBZ0 . . . . . . . CODE;    ** UNRESOLVED **  MODULE_1
```

Figure 4-3. IXREF Listing

Absolute Object File

The linking and locating process combines one or more relocatable object files into one absolute object file. The absolute object file contains one module; the absolute module consists of

- A module header record that identifies the module.
- A set of intermixed content and debug records. The content records contain the program code. The debug records contain the location and scope of local symbols, public symbols, segment symbols, and line numbers, as specified by the DEBUG-SYMBOLS, DEBUGPUBLICS, and DEBUGLINES controls in the RL51 invocation.
- A module end record that verifies the module name.

CHAPTER 5
EXAMPLES OF PROGRAM DEVELOPMENT

This chapter shows three brief examples of program development using ASM51, PL/M-51, and RL51. The first example is the sample program discussed in the *ASM51 User's Guide*; the example shows how to assemble each of the three modules, then link and locate them into a single absolute object module with RL51. The second example is a short program that illustrates the use of the locating controls. The third example shows the use of RL51 with PL/M-51 modules, emphasizing the library process.

Using Multiple Modules

The first example is a program of three modules, named SAMPLE, CONSOLE_IO, and NUM_CONVERSION. The source for these modules is in three files, SAMP1.A51, SAMP2.A51, and SAMP3.A51, respectively. To assemble these modules, invoke the assembler as follows:

```
:F1:ASM51 :F1:SAMP1.A51 DEBUG

:F1:ASM51 :F1:SAMP2.A51 DEBUG

:F1:ASM51 :F1:SAMP3.A51 DEBUG
```

Note that this example assumes the three source files are on the same drive as the assembler and linker/locator, and that the output file will be sent to the same drive. The assembler invocations use the DEBUG control to have the symbol tables output to the object files for the three modules.

After assembly is complete, the system has created object files SAMP1.OBJ, SAMP2.OBJ, and SAMP3.OBJ, and listing files SAMP1.LST, SAMP2.LST and SAMP3.LST. The three listing files are shown in figures 5-1, 5-2, and 5-3.

To link and locate the three modules, enter the command

```
:F1:RL51 :F1:SAMP1.OBJ, :F1:SAMP2.OBJ, :F1:SAMP3.OBJ &

**TO :F1:SAMPLE &

**PRINT (:F1:SAMPLE.LST) SYMBOLS LINES PUBLICS .
```

After the RL51 program has executed, the system has placed the absolute object module in file SAMPLE, and an output file with information on the link and locate process in file SAMPLE.LST. The output file also contains symbol table information as requested by the SYMBOLS, LINES, and PUBLICS controls. The listing file is shown in figure 5-4.

```
MCS-51 MACRO ASSEMBLER SAMPLE

ISIS-II MCS-51 MACRO ASSEMBLER V2.0
OBJECT MODULE PLACED IN :F1:SAMP1.OBJ
ASSEMBLER INVOKED BY:  :F1:ASM51 :F1:SAMP1.A51 DEBUG
LOC  OBJ             LINE     SOURCE

                        1     NAME SAMPLE
                        2     ;
                        3        EXTRN code (put_crlf, put_string, put_data_str)
                        4        EXTRN code (get_num, binasc, ascbin)
                        5     ;
----                    6     CSEG
                        7     ; This is the initializing section. Execution
                        8     ; always starts at address 0 on power-up.
0000                    9     ORG  0
0000 758920            10     MOV  TMOD,#00100000B  ; Set timer to auto-reload
0003 758D03            11     MOV  TH1,#(-253)      ; Set timer for 110 BAUD
0006 7598DA            12     MOV  SCON,#11011010B  ; Prepare the Serial Port
0009 D28E              13     SETB TR1              ; Start clock
                       14     ;
                       15     ; This is the main program. It's an infinite loop,
                       16     ; where each iteration prompts the console for 2
                       17     ; input numbers and types out their sum.
                       18     START:
                       19     ; Type message explaining how to correct a typo
000B 900000    F       20     MOV  DPTR,#typo_msg
000E 120000    F       21     CALL put_string
0011 120000    F       22     CALL put_crlf
                       23     ; Get first number from console
0014 900000    F       24     MOV  DPTR,#num1_msg
0017 120000    F       25     CALL put_string
001A 120000    F       26     CALL put_crlf
001D 7800      F       27     MOV  R0,#num1
001F 120000    F       28     CALL get_num
0022 120000    F       29     CALL put_crlf
                       30     ; Get second number from console
0025 900000    F       31     MOV  DPTR,#num2_msg
0028 120000    F       32     CALL put_string
002B 120000    F       33     CALL put_crlf
002E 7800      F       34     MOV  R0,#num2
0030 120000    F       35     CALL get_num
0033 120000    F       36     CALL put_crlf
                       37     ; Convert the ASCII numbers to binary
0036 7900      F       38     MOV  R1,#num1
0038 120000    F       39     CALL ascbin
003B 7900      F       40     MOV  R1,#num2
003D 120000    F       41     CALL ascbin
                       42     ; Add the 2 numbers, and store the results in SUM
0040 E500      F       43     MOV  a,num1
0042 2500      F       44     ADD  a,num2
0044 F500      F       45     MOV  sum,a
                       46     ; Convert SUM from binary to ASCII
0046 7900      F       47     MOV  R1,#sum
0048 120000    F       48     CALL binasc
                       49     ; Output sum to console
004B 900000    F       50     MOV  DPTR,#sum_msg
004E 120000    F       51     CALL put_string
0051 7900      F       52     MOV  R1,#sum
0053 7A04              53     MOV  R2,#4
0055 120000    F       54     CALL put_data_str
0058 80B1              55     JMP  start
                       56     ;
----                   57        DSEG  at 8
0008                   58     STACK:  DS  8   ; At power-up the stack pointer is
                       59                     ; initialized to point here.
```

Figure 5-1. SAMP1 Listing File

```
                        60      ;
                        61      DATA_AREA     segment DATA
                        62      CONSTANT_AREA segment CODE
                        63      ;
----                    64      RSEG  data_area
0000                    65      NUM1: DS  4
0004                    66      NUM2: DS  4
0008                    67      SUM:  DS  4
                        68      ;
----                    69      RSEG  constant_area
0000 54595045           70      TYPO_MSG: DB 'TYPE  X TO RETYPE A NUMBER',00H
0004 205E5820
0008 544F2052
000C 45545950
0010 45204120
0014 4E554D42
0018 4552
001A 00
001B 54595045           71      NUM1_MSG: DB 'TYPE IN FIRST NUMBER: ',00H
001F 20494E20
0023 46495253
0027 54204E55
002B 4D424552
002F 3A20
0031 00
0032 54595045           72      NUM2_MSG: DB 'TYPE IN SECOND NUMBER: ',00H
0036 20494E20
003A 5345434F
003E 4E44204E
0042 554D4245
0046 523A20
0049 00
004A 54484520           73      SUM_MSG:  DB 'THE SUM IS ',00H
004E 53554D20
0052 495320
0055 00
                        74      ;
                        75      END

SYMBOL TABLE LISTING
------ ----- -------
N A M E         T Y P E   V A L U E       A T T R I B U T E S

ASCBIN. . . .   C ADDR    ----      EXT
BINASC. . . .   C ADDR    ----      EXT
CONSTANT_AREA   C SEG     0056H           REL=UNIT
DATA_AREA . .   D SEG     000CH           REL=UNIT
GET_NUM . . .   C ADDR    ----      EXT
NUM1_MSG. . .   C ADDR    001BH   P       SEG=CONSTANT_AREA
NUM1. . . . .   D ADDR    0000H   R       SEG=DATA_AREA
NUM2_MSG. . .   C ADDR    0032H   R       SEG=CONSTANT_AREA
NUM2. . . . .   D ADDR    0004H   R       SEG=DATA_AREA
PUT_CRLF. . .   C ADDR    ----      EXT
PUT_DATA_STR.   C ADDR    ----      EXT
PUT_STRING. .   C ADDR    ----      EXT
SAMPLE. . . .     ----    ----
SCON. . . . .   D ADDR    0098H   A
STACK . . . .   D ADDR    0008H   A
START . . . .   C ADDR    000BH   A
SUM_MSG . . .   C ADDR    004AH   R       SEG=CONSTANT_AREA
SUM . . . . .   D ADDR    0008H   R       SEG=DATA_AREA
TH1 . . . . .   D ADDR    008DH   A
TMOD. . . . .   D ADDR    0089H   A
TR1 . . . . .   B ADDR    0088H.6 A
TYPO_MSG. . .   C ADDR    0000H   R       SEG=CONSTANT_AREA
REGISTER BANK(S) USED: 0, TARGET MACHINE(S): 8051
ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Figure 5-1. SAMP1 Listing File (Cont'd.)

```
ISIS-II MCS-51 MACRO ASSEMBLER V2.0
OBJECT MODULE PLACED IN :F1:SAMP2.OBJ
ASSEMBLER INVOKED BY:  :F1:ASM51 :F1:SAMP2.A51 DEBUG


LOC  OBJ             LINE     SOURCE

                        1        NAME CONSOLE_IO
                        2     ;
                        3     IO_ROUTINES segment CODE
----                    4        RSEG  IO_ROUTINES
                        5     ; This is the console IO routine cluster.
                        6        PUBLIC put_crlf, put_string,put_data_str,get_num
                        7        USING  0
                        8     ;
                        9     ; This routine outputs a Carriage Return and
                       10     ; a Line Feed
                       11     PUT_CRLF:
  000D                 12     CR EQU  0DH                ; carriage return
  000A                 13     LF EQU  0AH                ; line feed
                       14     ;
0000 740D              15        MOV  A,#cr
0002 120000    F       16        CALL put_char
0005 740A              17        MOV  A,#lf
0007 120000    F       18        CALL put_char
000A 22                19        RET
                       20     ;
                       21     ; Routine outputs a null-terminated string located
                       22     ; in CODE memory, whose address is given in DPTR.
                       23     PUT_STRING:
000B E4                24        CLR  A
000C 93                25        MOVC A,@A+DPTR
000D 6006              26        JZ   exit
000F 120000    F       27        CALL put_char
0012 A3                28        INC  DPTR
0013 80F6              29        JMP  put_string
                       30     EXIT:
0015 22                31        RET
                       32     ;
                       33     ; Routine outputs a string located in DATA memory,
                       34     ; whose address is in R1 and its length in R2.
                       35     PUT_DATA_STR:
0016 E7                36        MOV  A,@R1
0017 120000    F       37        CALL put_char
001A 09                38        INC  R1
001B DAF9              39        DJNZ R2,put_data_str
001D 22                40        RET
                       41     ;
                       42     ; Routine outputs a single character to console.
                       43     ; The character is given in A.
                       44     PUT_CHAR:
001E 3099FD            45        JNB  TI,$
0021 C299              46        CLR  TI
0023 F599              47        MOV  SBUF,A
0025 22                48        RET
                       49     ;
                       50     ; Get a 4 character string from console
                       51     ; and stores it at the address given in R0.
                       52     ; If a  X is received, routine starts over again.
                       53     GET_NUM:
0026 7A04              54        MOV  R2,#4     ; set up string length as 4
0028 A900              55        MOV  R1,AR0    ; R0 value may be needed for
restart
                       56     GET_LOOP:
002A 120000    F       57        CALL get_char
                       58     ; Next 4 instr's handle  X- the routine starts
```

Figure 5-2. SAMP2 Listing File

```
                        59      ; over if received
002D C2E7               60            CLR  ACC.7            ; clear the parity bit
002F B41805             61            CJNE A,#18H,GO_ON     ; if not  X- go on
0032 120000    F        62            CALL put_crlf
0035 80EF               63            JMP  get_num
                        64      GO_ON:
0037 F7                 65            MOV  @R1,A
0038 09                 66            INC  R1
0039 DAEF               67            DJNZ R2,get_loop
003B 22                 68            RET
                        69      ;
                        70      ; Get a single character from console.
                        71      ; The character is returned in A.
                        72      GET_CHAR:
003C 3098FD             73            JNB  RI,$
003F C298               74            CLR  RI
0041 E599               75            MOV  A,SBUF
0043 22                 76            RET
                        77      ;
                        78      END


SYMBOL TABLE LISTING
------ ----- -------


N A M E         T Y P E   V A L U E       A T T R I B U T E S

ACC. . . . .  D ADDR    00E0H   A
AR0. . . . .  D ADDR    0000H   A
CONSOLE_IO .    ----    ----
CR . . . . .    NUMB    000DH   A
EXIT . . . .  C ADDR    0015H   R         SEG=IO_ROUTINES
GET_CHAR . .  C ADDR    003CH   R         SEG=IO_ROUTINES
GET_LOOP . .  C ADDR    002AH   R         SEG=IO_ROUTINES
GET_NUM. . .  C ADDR    0026H   R PUB     SEG=IO_ROUTINES
GO_ON. . . .  C ADDR    0037H   R         SEG=IO_ROUTINES
IO_ROUTINES.  C SEG     0044H             REL=UNIT
LF . . . . .    NUMB    000AH   A
PUT_CHAR . .  C ADDR    001EH   R         SEG=IO_ROUTINES
PUT_CRLF . .  C ADDR    0000H   R PUB     SEG=IO_ROUTINES
PUT_DATA_STR  C ADDR    0016H   R PUB     SEG=IO_ROUTINES
PUT_STRING .  C ADDR    000BH   R PUB     SEG=IO_ROUTINES
RI . . . . .  B ADDR    0098H.0 A
SBUF . . . .  D ADDR    0099H   A
TI . . . . .  B ADDR    0098H.1 A

REGISTER BANK(S) USED: 0, TARGET MACHINE(S): 8051

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Figure 5-2. SAMP2 Listing File (Cont'd.)

```
MCS-51 MACRO ASSEMBLER    NUM_CONVERSION


ISIS-II MCS-51 MACRO ASSEMBLER V2.0
OBJECT MODULE PLACED IN :F1:SAMP3.OBJ
ASSEMBLER INVOKED BY:  :F1:ASM51 :F1:SAMP3.A51 DEBUG

LOC  OBJ             LINE     SOURCE

                        1        NAME NUM_CONVERSION
                        2     ;
                        3     NUM_ROUTINES segment CODE
----                    4        RSEG  NUM_ROUTINES
                        5     ; This module converts from ASCII to binary
                        6     ; and back. The binary numbers are signed one-byte
                        7     ; integers, i.e. range is -128 to +127. Their
                        8     ; ASCII representation is always 4 char's long-
                        9     ; i.e. a sign followed by 3 digits.
                       10        PUBLIC ascbin, binasc
                       11        USING  0
  0030                 12     ZERO   EQU  '0'
  002B                 13     PLUS   EQU  '+'
  002D                 14     MINUS  EQU  '-'
                       15     ;
                       16     ; This routine converts ASCII to binary.
                       17     ; INPUT-  a 4 char string pointed at by R1. The
                       18     ;    number range must be -128 to +127, and the
                       19     ;    string must have 3 digits preceded by a sign.
                       20     ; OUTPUT- a signed one-byte integer, located where
                       21     ;    the input string started (pointed at by R1).
                       22     ASCBIN:
0000 A801              23        MOV  R0,AR1  ; R1 original value needed later
                       24     ; Compute first digit value, and store it in TEMP
  REG                  25     TEMP EQU R3
0002 08                26        INC  R0
0003 E6                27        MOV  A,@R0
0004 C3                28        CLR  C
0005 9430              29        SUBB A,#zero
0007 75F064            30        MOV  B,#100
000A A4                31        MUL  AB
000B FB                32        MOV  TEMP,A
                       33     ; Compute the second digit value
000C 08                34        INC  R0
000D E6                35        MOV  A,@R0
000E 9430              36        SUBB A,#zero
0010 75F00A            37        MOV  B,#10
0013 A4                38        MUL  AB
                       39     ; Add the value of the second digit to num.
0014 2B                40        ADD  A,TEMP
0015 FB                41        MOV  TEMP,A
                       42     ; Get third digit and its value to total
0016 08                43        INC  R0
0017 E6                44        MOV  A,@R0
0018 C3                45        CLR  C
0019 9430              46        SUBB A,#zero
001B 2B                47        ADD  A,TEMP
001C FB                48        MOV  TEMP,A
                       49     ; Test the sign and complement the number if the
                       50     ; sign is a minus
001D E7                51        MOV  A,@R1
001E B42D04            52        CJNE A,#minus,pos    ;Skip the next 4 instr's
                       53                             ;if the number is positive
0021 EB                54        MOV  A,TEMP
0022 F4                55        CPL  A
0023 04                56        INC  A
0024 FB                57        MOV  TEMP,A
                       58     ;
                       59     ; Epilogue- store the result and exit
                       60     POS:
```

Figure 5-3. SAMP3 Listing File

```
0025 EB                 61          MOV  A,TEMP
0026 F7                 62          MOV  @R1,A
0027 22                 63          RET
                        64      ;
                        65      ; This routine converts binary to ASCII.
                        66      ; INPUT-a signed 1-byte integer, pointed at by R1
                        67      ; OUTPUT- a 4 character string, located where the
                        68      ;    input number was (pointed at by R1).
                        69      BINASC:
  00E7                  70      SIGN bit ACC.7
                        71      ; Get the number, find its sign and store its sign
0028 E7                 72          MOV  A,@R1
0029 772B               73          MOV  @R1,#plus      ; Store a plus sign (over-
                        74                              ;written by minus if needed)
002B 30E704             75          JNB  sign,go_on2        ;Test the sign bit
                        76      ; Next 3 instructions handle negative numbers
002E 772D               77          MOV  @R1,#minus         ;Store a minus sign
0030 14                 78          DEC A
0031 F4                 79          CPL A
                        80      ; Factor out the first digit
                        81      GO_ON2:
0032 09                 82          INC  R1
0033 75F064             83          MOV  B,#100
0036 84                 84          DIV  AB
0037 2430               85          ADD  A,#zero
0039 F7                 86          MOV  @R1,A              ;store the first digit
                        87      ; Factor out the second digit
003A 09                 88          INC  R1
003B E5F0               89          MOV  A,B
003D 75F00A             90          MOV  B,#10
0040 84                 91          DIV  AB
0041 2430               92          ADD  A,#zero
0043 F7                 93          MOV  @R1,A              ;store the second digit
                        94      ; Store the third digit
0044 09                 95          INC  R1
0045 E5F0               96          MOV  A,B
0047 2430               97          ADD  A,#zero
0049 F7                 98          MOV  @R1,A              ;store the third digit
                        99      ; note that we return without restoring R1
004A 22                100          RET
                       101      ;
                       102      END


SYMBOL TABLE LISTING
------ ----- -------

N A M E          T Y P E   V A L U E       A T T R I B U T E S

ACC. . . . . .   D ADDR    00E0H    A
AR1. . . . . .   D ADDR    0001H    A
ASCBIN . . . .   C ADDR    0000H    R PUB    SEG=NUM_ROUTINES
B. . . . . . .   D ADDR    00F0H    A
BINASC . . . .   C ADDR    0028H    R PUB    SEG=NUM_ROUTINES
GO_ON2 . . . .   C ADDR    0032H    R        SEG=NUM_ROUTINES
MINUS. . . . .     NUMB    002DH    A
NUM_CONVERSION     ----    ----
NUM_ROUTINES .   C SEG     004BH             REL=UNIT
PLUS . . . . .     NUMB    002BH    A
POS. . . . . .   C ADDR    0025H    R        SEG=NUM_ROUTINES
SIGN . . . . .   B ADDR    00E0H.7  A
TEMP . . . . .     REG     R3
ZERO . . . . .     NUMB    0030H    A

REGISTER BANK(S) USED: 0, TARGET MACHINE(S): 8051

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Figure 5-3. SAMP3 Listing File (Cont'd.)

```
ISIS-II RL51                                                    PAGE    1


ISIS-II MCS-51 RELOCATOR AND LINKER, V2.0, INVOKED BY:
 :F1:RL51 :F1:SAMP1.OBJ,:F1:SAMP2.OBJ,:F1:SAMP3.OBJ &
**TO :F1:SAMPLE &
**PRINT (:F1:SAMPLE.LST) SYMBOLS LINES PUBLICS IXREF

INPUT MODULES INCLUDED
  :F1:SAMP1.OBJ(SAMPLE)
  :F1:SAMP2.OBJ(CONSOLE_IO)
  :F1:SAMP3.OBJ(NUM_CONVERSION)

LINK MAP FOR :F1:SAMPLE(SAMPLE)


            TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
            ----    ----      ------    ----------   ------------

            REG     0000H     0008H                  "REG BANK 0"
            DATA    0008H     0008H     ABSOLUTE
            DATA    0010H     000CH     UNIT         DATA_AREA

            CODE    0000H     005AH     ABSOLUTE
            CODE    005AH     0056H     UNIT         CONSTANT_AREA
            CODE    00B0H     004BH     UNIT         NUM_ROUTINES
            CODE    00FBH     0044H     UNIT         IO_ROUTINES


SYMBOL TABLE FOR :F1:SAMPLE(SAMPLE)

VALUE           TYPE            NAME
-----           ----            ----

-------         MODULE          SAMPLE
C:005AH         SEGMENT         CONSTANT_AREA
D:0010H         SEGMENT         DATA_AREA
C:0075H         SYMBOL          NUM1_MSG
D:0010H         SYMBOL          NUM1
C:008CH         SYMBOL          NUM2_MSG
D:0014H         SYMBOL          NUM2
D:0098H         SYMBOL          SCON
D:0008H         SYMBOL          STACK
C:000BH         SYMBOL          START
C:00A4H         SYMBOL          SUM_MSG
D:0018H         SYMBOL          SUM
D:008DH         SYMBOL          TH1
D:0089H         SYMBOL          TMOD
B:0088H.6       SYMBOL          TR1
C:005AH         SYMBOL          TYPO_MSG
-------         ENDMOD          SAMPLE
-------         MODULE          CONSOLE_IO
D:00E0H         SYMBOL          ACC
D:0000H         SYMBOL          AR0
N:000DH         SYMBOL          CR
C:0110H         SYMBOL          EXIT
C:0137H         SYMBOL          GET_CHAR

ISIS-II RL51                                                    PAGE    2

C:0125H         SYMBOL          GET_LOOP
C:0121H         PUBLIC          GET_NUM
C:0132H         SYMBOL          GO_ON
C:00FBH         SEGMENT         IO_ROUTINES
N:000AH         SYMBOL          LF
C:0119H         SYMBOL          PUT_CHAR
```

Figure 5-4. RL51 Output File

```
C:00FBH         PUBLIC          PUT_CRLF
C:0111H         PUBLIC          PUT_DATA_STR
C:0106H         PUBLIC          PUT_STRING
B:0098H         SYMBOL          RI
D:0099H         SYMBOL          SBUF
B:0098H.1       SYMBOL          TI
-------         ENDMOD          CONSOLE_IO

-------         MODULE          NUM_CONVERSION
D:00E0H         SYMBOL          ACC
D:0001H         SYMBOL          AR1
C:00B0H         PUBLIC          ASCBIN
D:00F0H         SYMBOL          B
C:00D8H         PUBLIC          BINASC
C:00E2H         SYMBOL          GO_ON2
N:002DH         SYMBOL          MINUS
C:00B0H         SEGMENT         NUM_ROUTINES
N:002BH         SYMBOL          PLUS
C:00D5H         SYMBOL          POS
B:00E0H.7       SYMBOL          SIGN
N:0030H         SYMBOL          ZERO
-------         ENDMOD          NUM_CONVERSION

ISIS-II RL51                                                    PAGE   3

INTER-MODULE CROSS-REFERENCE LISTING
-----------------------------------

NAME. . . . . USAGE   MODULE NAMES
---------------------------------

ASCBIN. . . . CODE;   NUM_CONVERSION  SAMPLE
BINASC. . . . CODE;   NUM_CONVERSION  SAMPLE
GET_NUM . . . CODE;   CONSOLE_IO  SAMPLE
PUT_CRLF. . . CODE;   CONSOLE_IO  SAMPLE
PUT_DATA_STR. CODE;   CONSOLE_IO  SAMPLE
PUT_STRING. . CODE;   CONSOLE_IO  SAMPLE
```

Figure 5-4. RL51 Output File (Cont'd.)

Using the Locating Controls

The second example shows how to use the PRECEDE control to specify an order for data segments, in this case because the RL51 algorithm for locating segments results in a segment being left out.

The program is named TEST01. After assembly, the listing of TEST01.OBJ is as shown in figure 5-5. The program's code sequence is irrelevant to the example. The two DATA segments, SEG1 and SEG2, and the BIT segment, BIT3, are the points of interest for this example.

SEG1 is 21H bytes long; SEG2, 50H bytes long; SEG3, one bit long. The assembler listing also shows working register bank 0 (8 bytes long, absolutely located at addresses 00H through 07H).

All these segments are to be located in the on-chip data RAM of an 8051. For the 8051, the directly-addressable on-chip data RAM is 80H bytes long (addresses 00H through 7FH); addresses 20H through 2FH are bit-addressable. The working registers may occupy the first 20H bytes of the space. To see what RL51 does with this program, enter the command

```
RL51 :F1:TEST01.OBJ
```

```
MCS-51 MACRO ASSEMBLER    TEST01


ISIS-II MCS-51 MACRO ASSEMBLER V2.0
OBJECT MODULE PLACED IN :F1:TEST01.OBJ
ASSEMBLER INVOKED BY:  :F1:ASM51 :F1:TEST01.SRC PRINT (:CO:) &
**OBJECT (:F1:TEST01.OBJ)


LOC  OBJ             LINE     SOURCE

                        1     ; This test shows the use of the
                        2     ;   PRECEDE locating control.
                        3     ;   One bit causes failure of the
                        4     ;   RL51 allocation algorithm,
                        5     ;   but the PRECEDE control fixes it.
                        6     ;
                        7             NAME test01
                        8     ;
                        9     prog    SEGMENT   CODE
                       10     seg1    SEGMENT   DATA
                       11     seg2    SEGMENT   DATA
                       12     seg3    SEGMENT   BIT
                       13     ;
----                   14             RSEG      prog
                       15     ;
                       16     ;   Code segment.
                       17     ;
0000 434F4445          18             DB        'CODE IS IRRELEVANT'
0004 20495320
0008 49525245
000C 4C455641
0010 4E54
                       19     ;
----                   20             RSEG      seg1
0000                   21             DS        021H
                       22     ;
----                   23             RSEG      seg2
0000                   24             DS        050H
                       25     ;
----                   26             RSEG      seg3
0000                   27             DBIT      001H
                       28     ;
                       29             END
```

Figure 5-5. TEST01 Assembly Listing File

The RL51 listing file is shown in figure 5-6. ERROR 107 informs us that the locate attempt for SEG1 would overflow the data space; SEG1 was ignored (not located) for this reason. The link map shows the following assignments for the remaining segments:

Addresses	Segment
00H - 07H	Register Bank 0
08H - 1FH	GAP
20H	SEG3 (one bit at bit location 0)
20H.1 - 20H.7	GAP
21H - 71H	SEG2 (50H bytes)

After these segments have been located, there is not enough room for SEG1 (21H bytes). However, there would be enough room if SEG1 were located before the BIT segment. To obtain this result, the command is

```
RL51 :F1:TEST01.OBJ PRECEDE(SEG1)
```

```
ISIS-II RL51

ISIS-II MCS-51 RELOCATOR AND LINKER, V2.0, INVOKED BY:
 RL51 :F1:TEST01.OBJ


INPUT MODULES INCLUDED
  :F1:TEST01.OBJ(TEST01)


LINK MAP FOR :F1:TEST01(TEST01)

            TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
            ----    ----      ------    ----------   ------------

            REG     0000H     0008H                  "REG BANK 0"
                    0008H     0018H                  *** GAP ***
            BIT     0020H     0000H.1   UNIT         SEG3
                    0020H.1   0000H.7                *** GAP ***
            DATA    0021H     0050H     UNIT         SEG2

            CODE    0000H     0012H     UNIT         PROG


IGNORED SEGMENTS
  SEG1

- ERROR 107:  ADDRESS SPACE OVERFLOW
  SPACE:  DATA
  SEGMENT:  SEG1
```

Figure 5-6. RL51 Listing File Without PRECEDE

The RL51 listing file for this example is shown in figure 5-7. The PRECEDE control caused the link mapping to be as follows:

Addresses	Segment
00H - 07H	Register Bank 0
08H - 28H	SEG1 (21H bytes)
29H	SEG3 (one bit at bit location 0)
29H.1 - 29H.7	GAP
2AH - 7AH	SEG2 (50H bytes)

Refer to Chapter 2 for details on RL51's allocating algorithm.

```
ISIS-II RL51

ISIS-II MCS-51 RELOCATOR AND LINKER, X021, INVOKED BY:
 RL51 :F1:TEST01.OBJ  PRECEDE(SEG1)

INPUT MODULES INCLUDED
  :F1:TEST01.OBJ(TEST01)

LINK MAP FOR :F1:TEST01(TEST01)

            TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
            ----    ----      ------    ----------   ------------

            REG     0000H     0008H                  "REG BANK 0"
            DATA    0008H     0021H     UNIT         SEG1
            BIT     0029H     0000H.1   UNIT         SEG3
                    0029H.1   0000H.7                *** GAP ***
            DATA    002AH     0050H     UNIT         SEG2

            CODE    0000H     0012H     UNIT         PROG
```

Figure 5-7. RL51 Listing File with PRECEDE

Using RL51 with PL/M-51 Modules

The third example shows how to use RL51 with object modules produced by PL/M-51. The example shows the use of PLM51.LIB and demonstrates PL/M-51 generated segments and the PL/M-51 to ASM51 linkage.

The entire application introduces a way to halt ICE-51 at run time. The procedure CHECK_EQUAL in the PL/M-51 module CHK_EQ checks if an arithmetic expression is true. If yes, it calls the HALT_ICE assembler routine, which causes ICE51 to stop the program that is currently running. The code of the program is irrelevant; the example merely intends to show the program development process.

The PLM51 main module CHK_EQ is compiled by

```
PLM51 :F1:CHKEQ.P51 DEBUG PW(90)
```

The output of the compilation is shown in figure 5-8.

The ASM51 module HLTICE is assembled by

```
ASM51 :F1:HLTICE.A51 DEBUG PW(90)
```

The output of the compilation is shown in figure 5-9.

RL51 is invoked by the following command:

```
RL51 :F1:CHKEQ.OBJ,:F1:HLTICE.OBJ,PLM51.LIB IXREF PW(72)
```

RL51 links the two pre-translated input modules, along with the mandatory library PLM51.LIB. PLM51.LIB must be linked whenever a PL/M-51 module participates in the linkage. The result of the linkage is shown in figure 5-10.

```
PL/M-51 COMPILER     == ICE51 - Check/Halt ==                          PAGE   1


ISIS-II PL/M-51 V1.0
COMPILER INVOKED BY:  plm51 :f1:cnkeq.p51 debug pw(90)


                $title ('== ICE51 - Check/Halt ==')
                /*****************************************/
                /* Check_equal:                          */
                /*      Check if comparison yields the   */
                /*      expected result. If not, call    */
                /*      assembler routine to return to ICE. */
                /*****************************************/

                chk_eq:
   1    1          DO;

   2    1       halt_ice:
                   PROCEDURE (val1, val2, eq_switch) EXTERNAL;
   3    2             DECLARE (val1, val2) WORD, eq_switch BIT; END;

   5    1       check_equal:
                   PROCEDURE (val1, val2, eq_switch) PUBLIC;
   6    2             DECLARE
                        (val1, val2) WORD,
                         eq_switch    BIT;

   7    2             IF ((val1 <> val2) <> eq_switch)
                         THEN CALL halt_ice (val1, val2, eq_switch);
   9    1          END check_equal;

                /* dummy main program */

  10    1          DECLARE
                      pi WORD CONSTANT (3),
                      si WORD;

  11    1          CALL check_equal(pi*si, 27/si, 1);

  12    1       END chk_eq;


MODULE INFORMATION:                (STATIC+OVERLAYABLE)
    CODE SIZE                      = 0057H       87D
    CONSTANT SIZE                  = 0002H        2D
    DIRECT VARIABLE SIZE           =   02H+04H    2D+  4D
    INDIRECT VARIABLE SIZE         =   00H+00H    0D+  0D
    BIT SIZE                       =   00H+01H    0D+  1D
    BIT-ADDRESSABLE SIZE           =   00H+00H    0D+  0D
    AUXILIARY VARIABLE SIZE        = 0000H        0D
    MAXIMUM STACK SIZE             = 0006H        6D
    REGISTER-BANK(S) USED:           0
    34 LINES READ
    0 PROGRAM ERROR(S)
END OF PL/M-51 COMPILATION
```

Figure 5-8. PL/M-51 Listing File of CHK EQ

```
MCS-51 MACRO ASSEMBLER    '== ICE51 - Halt =='                                  PAGE    1
ISIS-II MCS-51 MACRO ASSEMBLER V2.0
OBJECT MODULE PLACED IN :F1:HLTICE.OBJ
ASSEMBLER INVOKED BY:  asm51 :f1:hltice.a51 debug pw(90)

LOC  OBJ             LINE     SOURCE

                        1     $title ('== ICE51 - Halt ==')
                        2     ;*******************************************
                        3     ;* Halt_Ice:                               *
                        4     ;*     Store word parameters in R45, R67,  *
                        5     ;*     Bit in C and execute A5 instruction *
                        6     ;*     to return to ICE.                   *
                        7     ;*******************************************
                        8     ;
                        9           NAME    halt_ice
                       10           PUBLIC  halt_ice, ?halt_ice?bit, ?halt_ice?byte
                       11     bits  SEGMENT  BIT
                       12     bytes SEGMENT  DATA
                       13     prog  SEGMENT  CODE
                       14     ;
----                   15           RSEG    bits
                       16     ?halt_ice?bit:
                       17     bit_par:
0000                   18           DBIT 1
                       19     ;
----                   20           RSEG    bytes
                       21     ?halt_ice?byte:
                       22     first_par:
0000                   23           DS  2
                       24     second_par:
0002                   25           DS  2
                       26     ;
----                   27           RSEG    prog
                       28     halt_ice:
0000 AC00      F       29           MOV   R4,first_par        ; move 1st par to place
0002 AD00      F       30           MOV   R5,first_par+1
0004 AE00      F       31           MOV   R6,second_par       ; move 2nd par to place
0006 AF00      F       32           MOV   R7,second_par+1
0008 A200      F       33           MOV   C,bit_par           ; move bit par to place
                       34     ;
000A A5                35           DB    0A5H                ; illegal op-code.
000B 00                36           NOP                       ; Will stop ICE-51 if
000C 00                37           NOP                       ; you type "GO TIL OPC IS A5"
000D 00                38           NOP
000E 22                39           RET                       ; you can continue after stop
                       40
                       41           END

MCS-51 MACRO ASSEMBLER    '== ICE51 - Halt =='                                  PAGE    2
SYMBOL TABLE LISTING
------ ----- -------
N A M E          T Y P E   V A L U E       A T T R I B U T E S

?HALT_ICE?BIT.   B ADDR    0000H.0 R PUB   SEG=BITS
?HALT_ICE?BYTE   D ADDR    0000H   R PUB   SEG=BYTES
BIT_PAR. . . .   B ADDR    0000H.0 R       SEG=BITS
BITS . . . . .   B SEG     0001H           REL=UNIT
BYTES. . . . .   D SEG     0004H           REL=UNIT
FIRST_PAR. . .   D ADDR    0000H   R       SEG=BYTES
HALT_ICE . . .   C ADDR    0000H   R PUB   SEG=PROG
PROG . . . . .   C SEG     000FH           REL=UNIT
SECOND_PAR . .   D ADDR    0002H   R       SEG=BYTES

REGISTER BANK(S) USED: 0, TARGET MACHINE(S): 8051

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Figure 5-9. ASM51 Listing File of HLTICE

```
I ISIS-II RL51                                                    PAGE   1


ISIS-II MCS-51 RELOCATOR AND LINKER, V2.0, INVOKED BY:
 RL51 :F1:CHKEQ.OBJ,:F1:HLTICE.OBJ,PLM51.LIB IXREF PW(72)


INPUT MODULES INCLUDED
  :F1:CHKEQ.OBJ(CHK_EQ)
  :F1:HLTICE.OBJ(HALT_ICE)
  :F0:PLM51.LIB(?P0034)
  :F0:PLM51.LIB(?P0038)
  :F0:PLM51.LIB(?PIV0R)


LINK MAP FOR :F1:CHKEQ(CHK_EQ)

            TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
            ----    ----      ------    ----------   ------------

            REG     0000H     0008H                  "REG BANK 0"
            DATA    0008H     0004H     UNIT         BYTES
            DATA    000CH     0004H     UNIT         ?CHK_EQ?DT?0
            DATA    0010H     0002H     UNIT         ?CHK_EQ?DT
                    0012H     000EH                  *** GAP ***
            BIT     0020H     0000H.1   UNIT         ?CHK_EQ?BI?0
            BIT     0020H.1   0000H.1   UNIT         BITS
                    0020H.2   0000H.6                *** GAP ***
            IDATA   0021H     0001H     UNIT         ?STACK

            CODE    0000H     0003H     ABSOLUTE
            CODE    0003H     0057H     INBLOCK      ?CHK_EQ?PR
            CODE    005AH     0016H     UNIT         ?P0038S
            CODE    0070H     000FH     UNIT         PROG
            CODE    007FH     000CH     UNIT         ?P0034S
            CODE    008BH     0009H     UNIT         ?PIV0RS
            CODE    0094H     0002H     UNIT         ?CHK_EQ?CO

SYMBOL TABLE FOR :F1:CHKEQ(CHK_EQ)

VALUE           TYPE            NAME
-----           ----            ----

-------         MODULE          CHK_EQ
C:0003H         PUBLIC          CHECK_EQUAL
C:0033H         SYMBOL          CHK_EQ
-------         PROC            CHECK_EQUAL
D:000CH         SYMBOL          VAL1
D:000EH         SYMBOL          VAL2
B:0020H         SYMBOL          EQ_SWITCH
-------         ENDPROC         CHECK_EQUAL
C:0094H         SYMBOL          PI
D:0010H         SYMBOL          SI
C:0033H         LINE#           1
C:0003H         LINE#           5
C:0003H         LINE#           7
C:001FH         LINE#           8
C:0032H         LINE#           9
```

Figure 5-10. RL51 Listing File of CHK_EQ

```
ISIS-II RL51                                                    PAGE    2


C:0033H         LINE#           11
C:005AH         LINE#           12
-------         ENDMOD          CHK_EQ

-------         MODULE          HALT_ICE
B:0020H.1       PUBLIC          ?HALT_ICE?BIT
D:0008H         PUBLIC          ?HALT_ICE?BYTE
B:0020H.1       SYMBOL          BIT_PAR
B:0020H.1       SEGMENT         BITS
D:0008H         SEGMENT         BYTES
D:0008H         SYMBOL          FIRST_PAR
C:0070H         PUBLIC          HALT_ICE
C:0070H         SEGMENT         PROG
D:000AH         SYMBOL          SECOND_PAR
-------         ENDMOD          HALT_ICE


ISIS-II RL51                                                    PAGE    3

INTER-MODULE CROSS-REFERENCE LISTING
------------------------------------


NAME . . . . . USAGE   MODULE NAMES
---------------------------------------

?CHECK_EQUAL?BIT . BIT;    CHK_EQ
?CHECK_EQUAL?BYTE. DATA;   CHK_EQ
?HALT_ICE?BIT. . . BIT;    HALT_ICE  CHK_EQ
?HALT_ICE?BYTE . . DATA;   HALT_ICE  CHK_EQ
?P0034 . . . . . CODE;     ?P0034  CHK_EQ
?P0038 . . . . . CODE;     ?P0038  CHK_EQ
?PIH0R . . . . . CODE;     CHK_EQ  ?PIV0R
?PIV0R . . . . . CODE;     ?PIV0R  CHK_EQ
?PSW0R . . . . . NUMB;     CHK_EQ  ?PIV0R
CHECK_EQUAL. . . . CODE;   CHK_EQ
HALT_ICE . . . . . CODE;   HALT_ICE  CHK_EQ
```

Figure 5-10. RL51 Listing File of CHK_EQ (Cont'd.)

The result of a linkage process that includes PL/M-51 modules deserves an explanation. The following paragraphs describe the modules, segments, and symbols that appear in the output listing of such a linkage. The explanation refers to the actual example (figure 5-10).

In addition to the two input modules CHK_EQ and HALT_ICE, RL51 pulled some modules from PLM51.LIB. The two modules ?P0034 and ?P0038 contain common PL/M-51 run-time routines and were pulled to resolve calls to those routines in the CHK_EQ module. The module ?PIV0R contains the initialization routine (set the stack pointer, set PSW), and is pulled whenever a linkage process encounters a main module written in PL/M-51.

The segments BYTES, BITS, and PROG are the user segments as defined in the ASM51 HALT_ICE module. The code segments ?P0034S, ?P0038S and ?PIV0RS are the code segments of the previously explained run-time routines.

All segments whose names are of the form ?CHK_EQ?*any* are segments generated by PL/M-51 as result of compiling module CHK_EQ. The prefix ?CHK_EQ? indicates that the segment belongs to the CHK_EQ module. The suffix indicates the segment type; e.g, PR stands for the PRogram CODE segment, CO for the COnstant CODE segment, DT for DATA segment, and BI for BIT segment.

On-chip segment names may be followed by a register bank number (0-3). This number indicates the register bank that must be in effect while data in this segment is accessed.

The ?STACK segment was discussed before. Note that this segment is not supplied by the user, but is pulled automatically from PLM51.LIB because the main module is written in PL/M-51. The absolute segment at 0000H-0002H contains the reset vector, which consists of a JUMP to the initialization routine contained in the ?PIV0RS segment.

Most of PL/M-51-generated relocatable segments have the UNIT relocation type. A frequent exception is the program code segment (?CHK_EQ?PR), which is INBLOCK whenever a module is compiled under ROM (MEDIUM), which is the default used by the compiler. Another (less frequent) exception is the BITADDRESSABLE DATA segment generated when bit structures are declared within the PL/M-51 source program.

User symbols appear in the symbol table and the IXREF report. Symbols whose names are equal to segments and modules defined previously represent entry points in the appropriate modules/segments pulled from PLM51.LIB (e.g., the symbol ?P0034 is a code address in the module ?P0034).

Symbols in the format ?*procedure*?BYTE or ?*procedure*?BIT (e.g., ?HALT_ICE?BYTE) are DATA and BIT addresses used for passing parameters to the appropriate external procedures (as implied by the name). BYTE and WORD parameters are placed at DATA address starting at, for example, ?HALT_ICE?BYTE. BIT parameters are placed at BIT address starting at ?HALT_ICE?BIT (see also the *PL/M-51 User-s Guide* about PL/M-51 linkage to ASM51).

CHAPTER 6
LIB51 LIBRARIAN

Introduction

LIB51 is used to create and manipulate library files. Library files are specially formatted collections of object modules, any element of which may be retrieved during the linkage process in order to resolve program references to external names.

LIB51 also allows you to alter the contents of these files by adding new modules or deleting old ones.

LIB51 user interface is fully compatible with LIB (the ISIS-II Librarian for the 8080 and 8085 environment).

LIB51 Input

Input to LIB51 consists of modules originally generated by one of the 8051 language translators.

The Invocation Line

The LIB51 program is called by the command

```
LIB51 <cr>
```

LIB51 identifies itself with a sign-on message, followed by an asterisk prompt, shown as follows

```
ISIS-II MCS-51 LIBRARIAN Vx.y
```

where

x.y is the version number.

While in LIB51, you will receive the asterisk prompt after each command is completed. Following the asterisk prompt, you may enter any of the following LIB51 subcommands:

ADD
CREATE
DELETE
EXIT
LIST

The Command Line

If the command line is longer than a line on your particular console (up to the maximum of 122 characters allowed), you may continue it on the next line by entering an ampersand (&) as the last non-blank character on the line before executing the carriage return. LIB51 responds to this with a double asterisk prompt (**) to let you know that it is ready for the continuation of the command line.

CAUTION

LIB51 uses a temporary file named LIB.TMP on the library file disk. If you have a file with this name, it will be destroyed.

Error Messages

Appendix D lists the error messages generated by LIB51.

LIB51 Subcommands

Each of the LIB51 subcommands is described on the following pages. The syntax and definition of each subcommand is given, along with an example of its use.

ADD

The ADD command adds modules to a specified library.

Syntax

ADD *filename*[(*modname*,...)][,...] TO *library*

Definition

This command inserts modules into the library. The modules may be elements of another library, or they may be in object files.

filename is the name of the file containing at least one object module. *modname* is the name of a module within *filename; modname* may be specified only if *filename* is a library file.

If *modname*s are given, only the specified modules within *filename* are copied into *library*; otherwise, all modules are copied. You may enter as many *filename*s or *modname*s as you wish. *library* is the name of an existing library file, to which the specified modules will be added.

Example

```
-ADD NIXON.OBJ, NOVEL.LIB(HISTORICAL,LESSONS) TO SCHOOL.LIB<cr>
```

CREATE

The CREATE command creates a new empty library.

Syntax

```
CREATE filename
```

Definition

This command creates a new library file called *filename*. If another file exists with that name, an error message is generated and the user is prompted for a new subcommand.

DELETE

The DELETE command deletes modules from a specified library.

Syntax

```
DELETE library( modname, . . . )
```

Definition

The DELETE subcommand permits you to remove modules for which you have no need, from the specified library. DELETE removes the module specified and updates the library directory. *library* is the library from which the deletion is to be made, and *modname* is the name of the module to be removed. You may specify as many *modname*s as you wish.

Example

```
*DELETE SCHOOL.LIB (ATLAS, YEAR_1932)<cr>
```

LIST

The LIST command shows the current content of a specified library(ies).

Syntax

```
LIST library[(modname,...)] [,...] [TO listfile] [PUBLICS]
```

Definition

The LIST command enables you to examine the contents of the specified library. A listing of the contents of the library may be printed; you can send this list to a file to be printed later or you may print the list directly, depending upon *listfile*. If *listfile* is omitted, the listing is sent to the console output.

library is the library for which you need the list of modules, *modname* is the name of the module desired, *listfile* is a file or an output device on which the list of modules is to be printed, and PUBLICS optionally calls for a listing of the public symbols in each listed module.

Example

The following examples illustrate the use of the LIST subcommand:

```
*LIST SCHOOL.LIB (REFERENCE, ONE DAY)<cr>

*LIST REF.LIB (ALMANACS, DICTIONARIES, ENCYCLOPEDIAS, MAPS)&
**TO :F1:REMOVE.NOT PUBLICS<cr>
```

See figure 6-1 for an example of the LIST output format.

```
PRINTED_MATTER
   PUBLISHER
   LIBRARY_OF_CONGRESS_NUMBER
DICTIONARIES
   ABRIDGED
   UNABRIDGED
ENCYCLOPEDIAS
   VOLUMES
   SETS
   PUBLISHER
MAPS
   STATE
   COUNTRY
   RELIEF
   GLOBES
```

Figure 6-1. LIST Command Output

EXIT

The EXIT terminates the LIB51 program.

Syntax

```
EXIT
```

APPENDIX A
SUMMARY OF RL51 CONTROLS

Table of Basic Definitions

Table A-1 gives definitions of basic terms used in the command format summary.

Table A-1. Definitions of Common Terms

Term	Definition
name	Names can be from 1 to 40 characters long and must be composed of letters A - Z, digits 0 - 9, or special characters (?, @, _). The first character must be a letter or a special character.
module-name	Same as name.
segment-name	Same as name.
pathname	A valid ISIS-II filename reference or device reference. See next two items for examples.
filename	A reference to a disk file. The format is [:F*n*:]*root* [.*ext*] Examples: PROG1, :F1:SAMPL1, TEST.HEX, :F2:SAMPLE.OBJ
device	A reference to a non-disk device. Examples: :LP:, :CO:, :TO:
value	A 16-bit unsigned integer. Examples: 1011B, 304Q, 4096D (or just 4096), 0C300H
address	Same as value.

RL51 Command Format Summary

Here is a summary of the syntax of the RL51 invocation command. Refer to the Preface for an explanation of the command format notation.

The RL51 command has the overall format

`[:Fn:]RL51` *input-list* `[TO` *output-file*`]` `[`*control-list*`]`

where

n := ; drive number

input-list := *input-file* `[`*module-list*`]` `[,...]`

input-file := *filename* ; see table A-1

module-list := `(`*module-name* `[,...])`

module-name := ; see table A-1

output-file := *filename* ; see table A-1

control-list := *control* ...

```
control : = { listing-control
            | linking-control
            | locating-control
            | configuration-control
            | overlay-control }

listing-control : = { print
                    | pagewidth
                    | map
                    | symbols
                    | publics
                    | lines
                    | ixref }

print : = { PRINT [(pathname)]
          | NOPRINT }

pathname : = ; see table A-1

pagewidth : = PAGEWIDTH (value)

value : = see table A-1

map : = { MAP
        | NOMAP }

symbols : = { SYMBOLS
            | NOSYMBOLS }

publics : = { PUBLICS
            | NOPUBLICS }

lines : = { LINES
          | NOLINES }

ixref : = { IXREF [selection-list]
          | NOIXREF }

selection-list : = (selection-item [,...])

selection-item : = { generated
                   | libraries }

generated : = { GENERATED
              | NOGENERATED }

libraries : = { LIBRARIES
              | NOLIBRARIES }

linking-control : = { NAME (module-name)
                    | debugsymbols
                    | debuglines
                    | debugpublics }

debugsymbols : = { DEBUGSYMBOLS
                 | NODEBUGSYMBOLS }

debuglines : = { DEBUGLINES
               | NODEBUGLINES }

debugpublics : = { DEBUGPUBLICS
                 | NODEBUGPUBLICS }

locating-controls : = { PRECEDE
                      | DATA
                      | BIT
                      | IDATA
                      | STACK
                      | XDATA
                      | CODE } (segment [,...])

segment : = segment-name [(address)]
```

segment-name : = ; see table A-1
address : = ; see table A-1

configuration-control : = *ramsize*

ramsize : = RAMSIZE (*value*)

value : = ; see table A-1

overlay-control : = { OVERLAY / NOOVERLAY } [(*overlay-unit* [, . . .])]

overlay-unit : = *ov-module-name calls ov-module-name*

ov-module-name : = { * / *module-name* }

module-name : = ; see table A-1

calls : = >

Tables of Listing, Linking, Locating, and OverlayingControls

Tables A-2 through A-6 describe the RL51 controls. Table A-7 gives abbreviations for the controls.

Notes On Locating Controls

The following notes refer to table A-4.

1. Bit addresses for non-BIT segments in the BIT control must be on byte boundaries; that is, they must be divisible by eight. (BIT-type segments can be aligned on bit boundaries.)
2. The range of addresses for the IDATA control is dependent on the target machine. The 8051 has 128 bytes (addresses 00H — 7FH). See the RAMSIZE control in this context.
3. The STACK control specifies which segments are to be allocated uppermost in the IDATA space. The memory accessed starts after the highest on-chip RAM address occupied by any previously allocated segment, and continues to the top of the IDATA space.

NOTE

This control has no other effect on any segments.

The IDATA ?STACK segment, if it exists, is placed higher than segments that were mentioned in the STACK control.

Table A-2. Listing Controls

Control	Effect
PRINT [(*pathname*)]	Sends the listing file to the file or device specified by pathname.
NOPRINT	Suppresses the listing file; overrides any of the following listing controls.
PAGEWIDTH (*value*)	Specifies the maximum page width to be used.
MAP	Outputs memory map to link summary.
NOMAP	Suppresses memory map.
SYMBOLS	Outputs local symbols to symbol table.
NOSYMBOLS	Suppresses local symbols.

Table A-2. Listing Controls (Cont'd.)

Control	Effect
PUBLICS	Outputs public symbols to symbol table.
NOPUBLICS	Suppresses public symbols.
LINES	Outputs line numbers to symbol table (high-level language translators only).
NOLINES	Suppresses line numbers.
IXREF [(*selection-list*)]	Appends intermodule cross-reference report to print file.
NOIXREF	Suppresses the intermodule cross-reference report.

NOTE: The default for any control (except IXREF) is the positive form (PRINT, MAP, SYMBOLS, PUBLICS, and LINES).

Table A-3. Linking Controls

Control	Effect
NAME (*module-name*)	Specifies the name of the output module. If the NAME control is omitted, the output module name defaults to the name of the first input module processed.
DEBUGSYMBOLS	Copies local symbol information to output file.
NODEBUGSYMBOLS	Suppresses local symbols.
DEBUGPUBLICS	Copies public symbol information to output file.
NODEBUGPUBLICS	Suppresses public symbols.
DEBUGLINES	Copies line number information (high-level language translators only) to output file.
NODEBUGLINES	Suppresses line numbers.

NOTE: For all linking controls except NAME, the default is the positive form (DEBUGSYMBOLS, DEBUGPUBLICS, and DEBUGLINES).

Table A-4. Locating Controls

Control	Address Space	Address Range (Hex)	Segment Types (and Attributes)
PRECEDE	Register banks and bit-addressable space in on-chip data RAM	00H-2FH	DATA (UNIT-aligned); IDATA
BIT	Bit-addressable space in on-chip data RAM	00H - 7FH (see note 1)	BIT; DATA; IDATA
DATA	Directly-addressable on-chip data RAM	00H - 7FH	DATA (UNIT-aligned); IDATA
IDATA	Indirectly-addressable on-chip data RAM	00H - 0FFH (see note 2)	IDATA
STACK	Same as IDATA (see note 3)	Same as IDATA	Same as IDATA
XDATA	External data RAM	0 - 0FFFFH	XDATA
CODE	Code memory	0 - 0FFFFH	CODE

Table A-5. Configuration Controls

Control	Effect
RAMSIZE (*value*)	Specifies the amount of on-chip RAM the object is aimed to.

Table A-6. Overlay Controls

Control	Effect
OVERLAY (*overlay-units*)	Overlays data segments, based on the information in the module declarations and in the overlay units.
NOOVERLAY	Suppresses the overlaying of data segments.

Table A-7. Abbreviations for Command Words

Command Word	Abbreviation
BIT	BI
CODE	CO
DATA	DT
DEBUGLINES	DL
DEBUGPUBLICS	DP
DEBUGSYMBOLS	DS
GENERATED	GN
IDATA	ID
IXREF	IX
LIBRARIES	LB
LINES	LI
MAP	MA
NAME	NA
NODEBUGLINES	NODL
NODEBUGPUBLICS	NODP
NODEBUGSYMBOLS	NODS
NOGENERATED	NOGN
NOIXREF	NOIX
NOLIBRARIES	NOLB
NOLINES	NOLI
NOMAP	NOMA
NOOVERLAY	NOOL
NOPRINT	NOPR
NOPUBLICS	NOPL
NOSYMBOLS	NOSB
OVERLAY	OL
PAGEWIDTH	PW
PRECEDE	PC
PRINT	PR
PUBLICS	PL
RAMSIZE	RS
STACK	ST
SYMBOLS	ST
TO	TO
XDATA	XD

APPENDIX B
RL51 ERROR MESSAGES

RL51 error messages describe warnings, errors, and fatal errors. A warning is a detected condition that may or may not be what the user desired; a warning does not terminate the link/locate operation. An error does not terminate operation, but probably results in an output module that cannot be used. A fatal error terminates operation of RL51.

This appendix lists the warning, error, and fatal error messages in that order. The text of each message is in UPPER CASE. A brief explanation of the probable cause for the error condition accompanies each error message.

Warnings

WARNING 1: UNRESOLVED EXTERNAL SYMBOL
SYMBOL: *external-name*
MODULE: *file-name(module-name)*

The specified external symbol, requested in the specified module, has no matching public symbol in any of the input modules.

WARNING 2: REFERENCE MADE TO UNRESOLVED EXTERNAL
SYMBOL: *external-name*
MODULE: *file-name(module-name)*
REFERENCE: *code-address*

The specified unresolved external is referenced in the specified module at the specified code address.

WARNING 3: ASSIGNED ADDRESS NOT COMPATIBLE WITH ALIGNMENT
SEGMENT: *segment-name*

The address specified for the segment in a locating control is not compatible with the segment's alignment. The segment is placed at the specified address, violating its alignment.

WARNING 4: DATA SPACE MEMORY OVERLAP
FROM: *byte.bit address*
TO: *byte.bit address*

The data space in the given range is occupied by two or more segments.

WARNING 5: CODE SPACE MEMORY OVERLAP
FROM: *byte address*
TO: *byte address*

The code space in a given range is occupied by two or more segments.

WARNING 6: XDATA SPACE MEMORY OVERLAP
FROM: *byte address*
TO: *byte address*

The xdata space in the given range is occupied by two or more segments.

```
WARNING 7: MODULE NAME NOT UNIQUE
MODULE:    file-name(module-name)
```

The specified name was used as the module name for more than one module. The specified module is not processed.

```
WARNING 8: MODULE NAME EXPLICITLY REQUESTED FROM
           ANOTHER FILE
MODULE:    file-name(module-name)
```

The specified module was requested, explicitly, to be processed from another file that has not yet been processed. The specified module is not processed.

```
WARNING 9: EMPTY ABSOLUTE SEGMENT
MODULE:    file-name(module-name)
```

The specified module contains an empty absolute segment. This segment is not allocated. The base address of this segment may be overlapped without any additional message.

Errors

```
ERROR 101: SEGMENT COMBINATION ERROR
SEGMENT:   segment-name
MODULE:    file-name(module-name)
```

The attributes of the specified partial segment, in the specified module, contradict those of previous (unspecified) occurrences of partial segments with the same name. The segment is ignored.

```
ERROR 102: EXTERNALS ATTRIBUTE MISMATCH
SYMBOL:    external-name
MODULE:    file-name(module-name)
```

The attributes of the specified external symbol, in the specified module, contradict those of previous (unspecified) occurrences of public symbol with the same name. The specified symbol is ignored.

```
ERROR 103: EXTERNAL ATTRIBUTES DO NOT MATCH PUBLIC
SYMBOL:    symbol-name
MODULE:    file-name(module-name)
```

The attributes of the specified external (public) symbol, in the specified module, contradict those of previous (unspecified) occurrences of public (external) symbol with the same name. The specified symbol is ignored.

```
ERROR 104: MULTIPLE PUBLIC DEFINITIONS
SYMBOL:    symbol-name
MODULE:    file-name(module-name)
```

The specified public symbol, in the specified module, has already been defined in a previously (unspecified) processed module. The specified symbol is ignored.

ERROR 105: PUBLIC REFERS TO IGNORED SEGMENT
SYMBOL: *public-name*
SEGMENT: *segment-name*

The specified public symbol is defined referencing the specified ignored segment. The specified public symbol is ignored.

ERROR 106: SEGMENT OVERFLOW
SEGMENT: *segment-name*

The specified segment, after combination, is larger than the maximum segment size allowed for the segment according to its type or to the given locating control. The specified segment is ignored.

ERROR 107: ADDRESS SPACE OVERFLOW
SPACE: *space-name*
SEGMENT: *segment name*

RL51 was unable to allocate the specified relocatable segment, according to the segment relocation type, in the specified address space. The specified segment is ignored.

ERROR 108: SEGMENT IN LOCATING CONTROL CANNOT BE ALLOCATED
SEGMENT: *segment name*

RL51 was unable to allocate the specified relocatable segment that appears in the locating control, according to the requirements imposed by the locating control and according to the segment relocation type. The specified segment is ignored.

ERROR 109: EMPTY RELOCATABLE SEGMENT
SEGMENT: *segment-name*

The specified segment, after combination has zero size. The specified segment is ignored.

ERROR 110: CANNOT FIND SEGMENT
SEGMENT: *segment-name*

The specified segment name occurred in the command tail but is not the name of any segment defined within the input files. The specified segment is ignored.

ERROR 111: SPECIFIED BIT ADDRESS NOT ON BYTE BOUNDARY
SEGMENT: *segment-name*

The specified segment was requested in a BIT locating control. The segment is not a BIT segment, and the requested address is not on byte boundary. The specified segment is ignored.

ERROR 112: SEGMENT TYPE NOT LEGAL FOR COMMAND
SEGMENT: *segment-name*

The specified segment is not one of the types that are legal for the locating control for which it is specified. The specified segment is ignored.

ERROR 113: RESERVED.

ERROR 114: SEGMENT DOES NOT FIT
SEGMENT: *segment-name, base, length*

The specified segment cannot be located at the base specified by the locating control. Starting at that base address there is insufficient memory for a segment of its length. The specified segment is ignored.

ERROR 115: INPAGE SEGMENT IS GREATER THAN 256 BYTES
SEGMENT: *segment-name*

The specified INPAGE segment is greater than one page. The specified segment is ignored.

ERROR 116: INBLOCK SEGMENT IS GREATER THAN 2047 BYTES
SEGMENT: *segment-name*

The specified INBLOCK segment is greater than one block. The specified segment is ignored.

ERROR 117: BIT ADDRESSABLE SEGMENT IS GREATER THAN 16 BYTES
SEGMENT: *segment-name*

The specified BIT-ADDRESSABLE segment is greater than the BIT space. The specified segment is ignored.

ERROR 118: REFERENCE MADE TO ERRONEOUS EXTERNAL
SYMBOL: *external-name*
MODULE: *file-name(module-name)*
REFERENCE: *code-address*

The specified, ignored external symbol is referenced in the specified module at the specified code address.

ERROR 119: REFERENCE MADE TO ERRONEOUS SEGMENT
SEGMENT: *segment-name*
MODULE: *file-name(module-name)*
REFERENCE: *code-address*

A symbol, which is defined using the specified, but ignored, segment, is referenced in the specified module at the specified code address.

ERROR 120: CONTENT BELONGS TO ERRONEOUS SEGMENT
SEGMENT: *segment-name*
MODULE: *file-name(module-name)*

A content record, which belongs to the specified, but ignored, segment, has been encountered. The content record is not relocated.

ERROR 121: IMPROPER FIXUP
MODULE: *file-name(module-name)*
SEGMENT: *segment-name*
OFFSET: *pseg-offset*

An error occurred in the evaluation of a fixup. An example of this error is when the value of the fixup expression does not meet the requirements of the type of the referenced location.

```
ERROR 122: CANNOT FIND MODULE
MODULE:    file-name(module-name)
```

The specified module name, which was explicitly requested from the specified file (in the command tail), was not found in that file.

```
ERROR 123: ABSOLUTE IDATA SEGMENT DOES NOT FIT
MODULE:    file-name(module-name)
FROM:      data-address
TO:        data-address
```

The specified module contains an absolute IDATA segment that occupies non-existent internal RAM space in the target machine. The segment is ignored. Notice, however, that the module may contain erroneous references to this segment, which are not reported.

```
ERROR 124: RESERVED
```

```
ERROR 125: MORE ERRORS ENCOUNTERED, NOT REPORTED
```

Non-fatal errors encountered henceforth will not be reported.

```
ERROR 126: OVERLAY MODULE NOT FOUND
MODULE:    file-name(module-name)
```

The specified module name explicitly mentioned in the overlay control was not found.

```
ERROR 127: OVERLAY DATA ADDRESS SPACE OVERFLOW
SPACE:     on-chipRAM space
```

RL-51 was unable to allocate an overlaid segment of the specified address space. Try to link with the NOOVERLAY control.

Fatal Errors

```
FATAL ERROR 201: INVALID COMMAND LINE SYNTAX
partial command
```

A syntax error was detected in the command. The command is repeated up to and including the point of error.

```
FATAL ERROR 202: INVALID COMMAND LINE; TOKEN TOO LONG
partial command
```

The command line contains a token that is too long. The command is repeated up to and including the point of error.

```
FATAL ERROR 203: EXPECTED ITEM MISSING
partial command
```

An expected item in the command line, such as an input file name or a file name following the TO is missing. The command is repeated up to and including the point of error.

```
FATAL ERROR 204: INVALID KEY WORD
partial command
```

An invalid keyword was found in the command. The command is repeated up to and including the point of error.

```
FATAL ERROR 205: NUMERIC CONSTANT TOO LARGE
partial command
```

A numeric constant greater than 0FFFFH was found in the command. The command is repeated up to and including the point of error.

```
FATAL ERROR 206: INVALID CONSTANT
partial command
```

An illegally constructed context was found. A common example of this error is entering a hexadecimal number with a letter first. The command is repeated up to and including the point of error.

```
FATAL ERROR 207: INVALID NAME
partial command
```

An illegally constructed name was found. Names can be from 1 through 40 characters long and must be composed of the letters A-Z, the digits 0-9, or special characters ("?", "@", "_"). The first character must be a letter or a special character. The command is repeated up to and including the point of error.

```
FATAL ERROR 208: INVALID FILE NAME
partial command
```

The file-name specified in the command is not a valid ISIS-II file name. The command is repeated up to and including the point or error.

```
FATAL ERROR 209: FILE USED IN CONFLICTING CONTEXTS
FILE:           file-name
```

The specified file is used in more than one context, for example, using the same file for both input and output. (This may be caused by specifying for the first input file a file that has no extension, and not specifying an output file.)

```
FATAL ERROR 210: I/O ERROR, INPUT FILE; ISIS-II ERROR#
FILE:           file-name
```

An ISIS-II I/O error was detected in accessing an input file. The text of the message includes a description of the specific I/O error that occurred. See the *ISIS-II User's Guide* for a list of possible I/O errors.

```
FATAL ERROR 211: I/O ERROR, OUTPUT FILE; ISIS-II ERROR#
FILE:           file-name
```

An ISIS-II I/O error was detected in accessing the output file. The text of the message includes a description of the specific I/O error that occurred. See the *ISIS-II User's Guide* for a list of possible I/O errors.

```
FATAL ERROR 212: I/O ERROR, LISTING FILE; ISIS-II
                 ERROR#
FILE:           file-name
```

An ISIS-II I/O error was detected in accessing the listing file. The text of the message includes a description of the specific I/O error that occurred. See the *ISIS-II User's Guide* for a list of possible I/O errors.

```
FATAL ERROR 213: I/O ERROR, TEMPORARY FILE; ISIS-II
                 ERROR#
```
FILE: *file-name*

An ISIS-II I/O error was detected in accessing a temporary file. The text of the message includes a description of the specific I/O error that occurred. See the *ISIS-II User's Guide* for a list of possible I/O errors.

```
FATAL ERROR 214: INPUT PHASE ERROR
```
MODULE: *file-name(module-name)*

This error occurs when RL51 encounters different data during pass two than it read during pass one.

```
FATAL ERROR 215: CHECK SUM ERROR
```
MODULE: *file-name(module-name)*

A bad check sum was detected in the input module. This indicates a bad input module or a read error.

```
FATAL ERROR 216: INSUFFICIENT MEMORY
```

The memory available for execution of RL51 has been used up. This is usually caused by too many external /public symbols or segments in the input files or by too many errors.

```
FATAL ERROR 217: NO MODULE TO BE PROCESSED
```

After scanning all the input files, no module was selected to be processed. This is usually caused by an empty input file(s) or incorrect module names in the input list.

```
FATAL ERROR 218: NOT AN OBJECT FILE
```
FILE: *file-name*

The file named in the message, judging by its first byte of data, is not a valid object file.

```
FATAL ERROR 219: NOT AN 8051 OBJECT FILE
```
FILE: *file-name*

The translator-ID field in the module header record indicates that the specified module is not an 8051 object module.

```
FATAL ERROR 220: INVALID INPUT MODULE
```
MODULE: *file-name(module-name)*

The specified input module was found to be invalid. Possible causes are incorrect record order, incorrect record type, illegal field, illegal relation between fields, or a missing required record. This error could be the result of a translator record.

```
FATAL ERROR 221: MODULE SPECIFIED MORE THAN ONCE
```
partial command

The input list in the invocation line contains the same module name more than once. The command is repeated up to and including the point of error.

```
FATAL ERROR 222: SEGMENT SPECIFIED MORE THAN ONCE
```
partial command

The locating controls in the invocation line contain the same segment name more than once. The command is repeated up to and including the point of error.

```
FATAL ERROR 223: NOT A DISK FILE
```
partial command

The file specified in the input list or as an output file is not a valid ISIS-II disk file name. The command is repeated up to and including the point of error.

```
FATAL ERROR 224: DUPLICATE KEYWORD
```
partial command

The same keyword appears in the command more than once. The command is repeated up to and including the point of error.

```
FATAL ERROR 225: SEGMENT ADDRESSES ARE NOT IN
                 ASCENDING ORDER
```
partial command

The addresses of the segments within one locating control are not in ascending order. The command is repeated up to and including the point of error.

```
FATAL ERROR 226: SEGMENT ADDRESS INVALID FOR CONTROL
```
partial command

The address requested for a segment is not valid for the given locating control. The command is repeated up to and including the point of error.

```
FATAL ERROR 227: PAGEWIDTH PARAMETER OUT OF RANGE
```
partial command

The PAGEWIDTH parameter given is out of the acceptable range.

```
FATAL ERROR 228: RAMSIZE PARAMETER OUT OF RANGE
```
partial command

The RAMSIZE parameter given is out of acceptable range.

```
FATAL ERROR 229: I/O ERROR, OVERLAY FILE;
                 ISIS-II ERROR#
```
FILE: *file-name*

An ISIS-II I/O error was detected in accessing an overlay file. The text of the message includes a description of the specific I/O error that occurred. See the *ISIS-II User's Guide* for a list of possible I/O errors. (This error occurs only if IXREF was requested. Its occurrence does not invalidate the output object file.)

```
FATAL ERROR 230: INCOMPATIBLE OVERLAY VERSION
```
FILE: *file-name*

The overlay file, although loaded successfully by ISIS-II, has a version number that is not the one expected by RL51. The possible cause is that the RL51 program and the loaded overlay are not of the same version. (This error occurs only if IXREF or OVERLAY was requested. If only IXREF was requested, the output object file is valid.)

FATAL ERROR 231: TOO MANY IXREF ENTRIES

The number of IXREF entries (entry is a pair consisting of modules and symbol reference) is too large to be processed. The IXREF listing step is not performed. The NOLIBRARIES and NOGENERATED controls may be used in order to decrease this number and overcome the error. (This error occurs only if IXREF was requested. Its occurrence does not invalidate the output object file.)

FATAL ERROR 232: OVERLAY CONTROL CONFLICTS XREF SELECTOR ITEMS

The overlay control should not appear with the IXREF selector items NOLIBRARIES or NOGENERATED.

FATAL ERROR 233: ILLEGAL USE OF * IN OVERLAY CONTROL

The use of * > * with the OVERLAY control is illegal.

FATAL ERROR 240: INTERNAL PROCESS ERROR

RL51 has detected that it has made a processing error. This error indicates a bug within RL51.

APPENDIX C
LIB51 COMMAND SUMMARY

The following is a summary of the commands used by the LIB51 librarian.

Command	Action
CREATE	Creates a new library file
ADD	Inserts new modules into a library
DELETE	Removes modules from a library
LIST	Lists the contents of a library
EXIT	Returns control to ISIS

APPENDIX D
LIB51 ERROR MESSAGES

All LIB51 error messages are non-fatal because LIB51 is an interactive program. The command that caused the error will be aborted, but LIB51 will not be interrupted.

Command Errors

Errors caused by improper command entry are followed by a partial copy of the incorrect command, with a number sign (#) in the vicinity of the error, as shown below:

```
ERROR MESSAGE
partial command#
```

The following are the LIB51 command error messages:

```
INSUFFICIENT MEMORY
```

LIB51 cannot execute the command given because it requires more memory than is available in the Intellec system.

```
INVALID MODULE NAME
```

A module listed in the command is incorrectly specified. Module names must conform to the format given in Chapter 2.

```
INVALID SYNTAX
```

Check the command for one of the following:

- Misspelled keywords
- Ampersand followed by a non-blank character
- ADD: TO *filename* not followed by a ‹ cr ›
- DELETE: *libname* (*modname*) not followed by a ‹ cr ›
- DELETE: *modname* not specified
- CREATE: *filename* not followed by a ‹ cr ›
- LIST: TO *filename* not followed by PUBLICS or a ‹ cr ›

```
LEFT PARENTHESIS EXPECTED
```

There is a missing left parenthesis in the command.

```
RIGHT PARENTHESIS EXPECTED
```

There is a missing right parenthesis in the command.

```
MODULE NAME TOO LONG
```

The specified module name exceeds 40 characters.

```
"TO" EXPECTED
```

The TO filename is omitted in the ADD command.

UNRECOGNIZED COMMAND

An illegal or misspelled command was entered. The only legal commands are ADD, CREATE, DELETE, LIST, and EXIT.

File or Module Errors

The following errors indicate that there is some problem with the file or module specified. There is no partial copy of the command given with these error messages.

FILE ALREADY EXISTS

The file specified in the CREATE command already exists. Choose a new name for the library.

filename, DUPLICATE SYMBOL IN INPUT

You have attempted to add a file that contains a PUBLIC symbol already within the library.

filename, NOT LIBRARY

The specified file is not a library.

filename(*modname*): NOT FOUND

You have attempted to delete a module that does not exist. Check for misspelling of the filename or module name.

modname—ATTEMPT TO ADD DUPLICATE MODULE

The specified module name already appears within the library.

symbol—ALREADY IN LIBRARY

You have attempted to add a module that contains a PUBLIC symbol that is already in the library.

filename, CHECKSUM ERROR (See ISIS-II error 208.)

filename, OBJECT RECORD TOO SHORT (See ISIS-II error 217.)

filename, ILLEGAL RECORD FORMAT (See ISIS-II error 218.)

LIB51 cannot process the specified file because it is not a legal object file. Possible cause is a file damage or translator error.

APPENDIX E
HEXADECIMAL-DECIMAL CONVERSION TABLE

Table E-1 is for hexadecimal to decimal and decimal to hexadecimal conversion. To find the decimal equivalent of a hexadecimal number, locate the hexadecimal number in the correct position and note the decimal equivalent. Add the decimal numbers.

To find the hexadecimal equivalent of a decimal number, locate the next lower decimal number in the table and note the hexadecimal number and its position. Subtract the decimal number shown in the table from the starting number. Find the difference in the table. Continue this process until there is no difference.

Table E-1. Hexadecimal-Decimal Conversion Table

Most Significant Byte				Least Significant Byte			
Digit 4		Digit 3		Digit 2		Digit 1	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0
1	4 096	1	256	1	16	1	1
2	8 192	2	512	2	32	2	2
3	12 288	3	768	3	48	3	3
4	16 384	4	1 024	4	64	4	4
5	20 480	5	1 280	5	80	5	5
6	24 576	6	1 536	6	96	6	6
7	28 672	7	1 792	7	112	7	7
8	32 768	8	2 048	8	128	8	8
9	36 864	9	2 304	9	144	9	9
A	40 960	A	2 560	A	160	A	10
B	45 056	B	2 816	B	176	B	11
C	49 152	C	3 072	C	192	C	12
D	53 248	D	3 328	D	208	D	13
E	57 344	E	3 548	E	224	E	14
F	61 440	F	3 840	F	240	F	15

INDEX

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Intel Literature Department (see page ii of this manual).

1. Please describe any errors you found in this publication (include page number).

2. Does the publication cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of publication for your needs? Is it at the right level? What other types of publications are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating).

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

(COUNTRY)

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS . . .

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

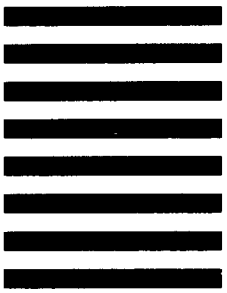
If you are in the United States, use the preprinted address provided on this form to return your comments. No postage is required. If you are not in the United States, return your comments to the Intel sales office in your country. For your convenience, a list of international sales offices is provided on the back cover of this document.

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

Intel Corporation
DTO Technical Publications, HF2-38
5200 N.E. Elam Young Parkway
Hillsboro, Oregon 97124-9987



INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) **987-8080**

Printed in U.S.A.

DEVELOPMENT SYSTEMS

3894/250/0188/OSPS/NR/DTO