8086 FAMILY UTILITIES USER'S GUIDE

for 8080/8085-Based Development Systems

Manual Order Number: 9800639-04 Rev. E

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

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

The information in this document is subject to change without notice.

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 may be used only to identify Intel products:

BXP	Intellec	Multibus
CREDIT	iSBC	Multimodule
i	iSBX	PROMPT
ICE	Library Manager	Promware
iCS	MCS	RMX
Insite	Megachassis	UPI
Intel	Micromap	μScope
Intelevision		

and the combination of ICE, iCS, iSBC, iSBX, MCS, or RMX and a numerical suffix.

A620/582/ 6K DD

PREFACE

This manual describes and shows how to use the LINK86, LOC86, LIB86, and OH86 commands that support 8086 program development. These commands run under Version 3.4 and later versions of ISIS-II.

This manual is directed to the programmer who is developing programs for the 8086 with PL/M-86 or ASM86 or any other language translator that generates object code compatible with the tools described in this manual.

This manual is divided into six chapters and four appendices:

- "Introduction," which provides an overview to the tools covered by the manual and the 8086 itself.
- "LINK86 Command," which describes and shows examples of the use of the LINK86 command.
- "LOC86 Command," which describes and shows examples of the use of the LOC86 command.
- "How LINK86 and LOC86 Handle Modules," which describes how LINK86 and LOC86 prepare output modules.
- "LIB86 Command," which describes and shows examples of the use of the LIB86 command to create and maintain program libraries.
- "OH86 Command," which describes and shows examples of the command to convert absolute object code to the hexadecimal format.
- "Appendix A: Error Messages," which list all the error and warning messages issued by the commands, as well as their probable causes.
- "Appendix B: Hexadecimal-Decimal Conversion," which is a table to help you convert between the two number systems.
- "Appendix C: PL/M-86 Models of Computation," which describes the structure of PL/M-86 output modules.
- "Appendix D: MCS-86 Absolute Object File Formats".

Related Publications

The following manuals may be helpful in various aspects of your work:

- *ISIS-II User's Guide*, order number 9800306, which describes the disk-based operating system for the Intellec development systems.
- *PL/M-86 Programming Manual*, order number 9800466, which describes the PL/M-86 programming language.
- *ISIS-II PL/M-86 Compiler Operator's Manual*, order number 9800478, which describes how to compile programs written in PL/M-86.
- *MCS-86™ Macro Assembly Language Reference Manual*, order number 9800640, which describes the 8086 assembly language.
- *MCS-86™ Macro Assembler Operating Instructions for ISIS-II Users*, order number 9800641, which describes how to assemble programs written in the 8086 assembly language.
- *8089 Macro Assembler User's Manual*, order number 9800638, which describes how to assemble programs written in 8089 Assembly language.

Notational Conventions

The following conventions are used in displaying command and control syntax in this manual:

BOLDFACE	Information in boldface and capitals must be entered as shown in the syntax statements. Although this information is shown in uppercase, it can be entered in uppercase or lowercase.
italic	Information in italics and lowercase represents variable information that you must supply when entering the commands.
[ ]	Brackets indicate parameters or controls that are optional.
{ }	Braces indicate a choice. One and only one of the items enclosed in braces can be chosen. None can be chosen only if the items are also enclosed in brackets.
|	The vertical bar indicates a choice. It is usually used within braces to separate choices.
...	The ellipsis indicates that multiple items can be entered.

CONTENTS

CONTENTS (Cont'd.)

TABLES

ILLUSTRATIONS

CHAPTER 1
INTRODUCTION

Program Development

Program development is a process of varying complexity. The complexity depends on the language used to develop code, the complexity of the end product, and the tools chosen to tie the parts together.

Figure 1-1 shows the development process and the tools available for development of an MCS-86 based product.

This manual gives you information necessary to use four parts of the MCS-86 Software Development Package. The MCS-86 Software Development Package allows you to develop the program for your 8086-based product on an 8080-based development system.

In particular, these tools are designed to run on an Intellec Microcomputer Development System or an Intellec Series II Microcomputer Development System. In either system, the tools run under control of ISIS-II, Version 3.4 or later.

The tools described in this manual are:

- LINK86, which is a linkage tool.
- LOC86, which is the relocation tool.
- LIB86, which is the librarian function for 8086 object modules.
- OH86, which converts 8086 absolute object information to the hexadecimal format.

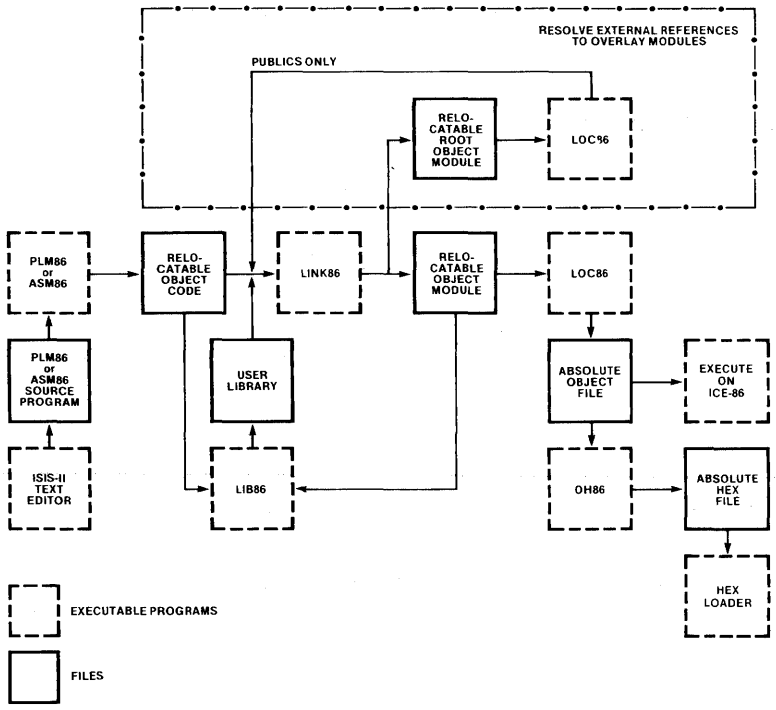


Figure 1-1. The MCS-86™ Development Process 639-1

Mechanics of Linkage and Relocation

LINK86 and LOC86 combine object modules and convert relative addresses to absolute addresses. They can combine any module in 8086 object module format. These modules can be produced by the 8086 assembler, 8088 assembler, 8089 assembler, or PL/M-86.

The segment is the fundamental unit with which the linkage and relocation functions work. First, segments are combined, then absolute addresses are assigned to segments. Addresses that are referenced relative to the beginning of a segment are translated to references to absolute addresses. This operation binds the segments to a memory location; the segments cannot be executed or accessed unless they reside at the assigned addresses.

The object modules to be combined are specified in a list in the LINK86 command. The sequence in which the segments in the input modules are combined and absolute addresses assigned to segments is determined by the order in which the modules are listed, the structure of the modules, and the controls supplied with the command. The details of module combination are described in Chapters 2 and 3.

The following information from object modules is used during linkage and relocation:

- Addresses given as offsets into segments which must be translated into absolute memory addresses.
- Segment definitions which identify contiguous pieces of information, usually code or data.
- Group definitions which identify segments that must be kept within a 64K byte range of memory.
- Class definitions which identify segments that share common attributes and should be kept together.
- Public and external symbols by which address references between different object modules can be resolved.

Relative Addressing

The relative addresses of instructions and data in program modules are assigned by the source translator. The addresses are relative to the beginning of the segment in which they reside. The relative address is actually the number of bytes from the beginning of the segment.

LINK86 combines one or more input modules to form one output module. The combining is done by segments. The order in which the command combines segments is described in Chapter 4.

After all the input segments are combined, LOC86 assigns absolute memory addresses to all relative addresses. The resulting output module can only be executed when its segments are loaded at the absolute addresses assigned by the command.

External References and Public Symbols

An address field that refers to a location in a different object module is called an external reference. An external reference differs from a relative address because the translator that generates the module knows nothing about the location of the referenced symbol. You must declare these references as external when coding a program. This tells the translator, and subsequently the relocation and linkage (R&L) commands, that the target of the reference is in a different module.

A module that contains external references is called an unsatisfied module. To satisfy the module, a module with a public symbol that matches the external symbol must be found. Associated with a public symbol in a module is an address that allows other modules, with the appropriate external reference, to reference the module with the public symbol. You must declare these symbols as public when coding the program. This tells the source translator and the R&L commands that other modules can reference the symbol.

If there are external references that are not satisfied by public symbols, warning messages are issued and the resulting module remains unsatisfied.

Use of Libraries

Libraries aid in the job of building programs. The library manager program, LIB86, creates and maintains files containing object modules.

LINK86 treats library files in a special manner. If you specify a library file as input to LINK86, it searches the library for modules that satisfy unresolved external references in the input modules it has *already* read. This means that libraries should be specified after the input modules that contain external references. If a module included from the library has an external reference, the library is searched again to try to satisfy the reference. This process continues until all external references are satisfied or until no public symbols are found in the library that match an unsatisfied external reference.

When LINK86 searches a library, it *normally* includes only library modules in the output that satisfy external references. If no external references are satisfied by a library, no modules from the library are included in the output module. However, LINK86 provides the means to unconditionally include a library module even if there is no external reference to it. Figure 1-2 shows LINK86 handling of a library file.

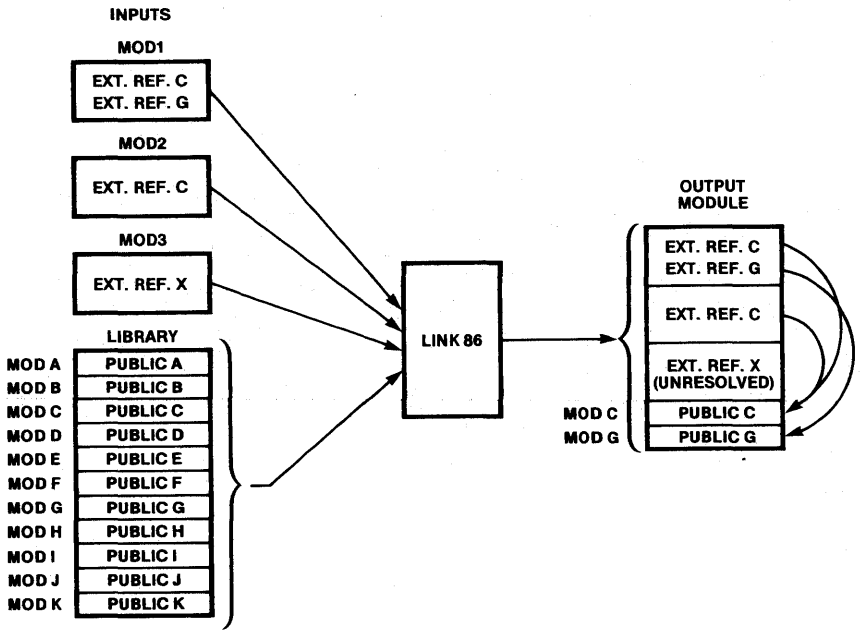


Figure 1-2. Library Linkage by LINK86 639-2

The LINK86/LOC86 Process

LINK86 and LOC86 execute on Intellec Development Systems. Execution is initiated by ISIS-II commands. Along with a list of input modules (LINK86) or a single input module (LOC86), the commands may contain controls that affect their output. Controls are not required for LINK86 and LOC86 execution. There are execution defaults for module combination, address assignment, and output information. The controls give you capability of changing the default algorithms and the types of output.

The inputs are object modules in disk files. The input modules can contain relative addresses, absolute addresses, external references, and public symbols. The input modules must be in the 8086 object module format such as is generated by the PL/M-86 compiler, MCS-86 assembler, and LINK86 and LOC86 themselves.

LINK86 combines segments from the input modules. LOC86 orders the segments and assigns absolute addresses according to the controls specified with the command and/or the default algorithms. Both commands output the module when processing is completed along with any error messages and diagnostic information. Figure 1-3 shows the LINK86/LOC86 process.

An 8086 Overview

To fully use the R&L commands you must have an understanding of how 8086 programs are structured and how memory is used.

Memory

The 8086 can address up to a maximum of a megabyte of memory. In decimal a megabyte is 1,048,576 bytes. Memory addresses are always shown in hexadecimal. A megabyte of memory has the addresses: 0H through 0FFFFFH.

Not all 8086-based systems will have a full megabyte of memory. Many systems will have gaps in the memory that is available. The different portions of memory will probably be implemented with different types of memory chips. The system monitor or supervisor is usually stored in ROM or PROM chips. Because it is not modified by execution it can be a permanent part of the system. This prevents the need to load

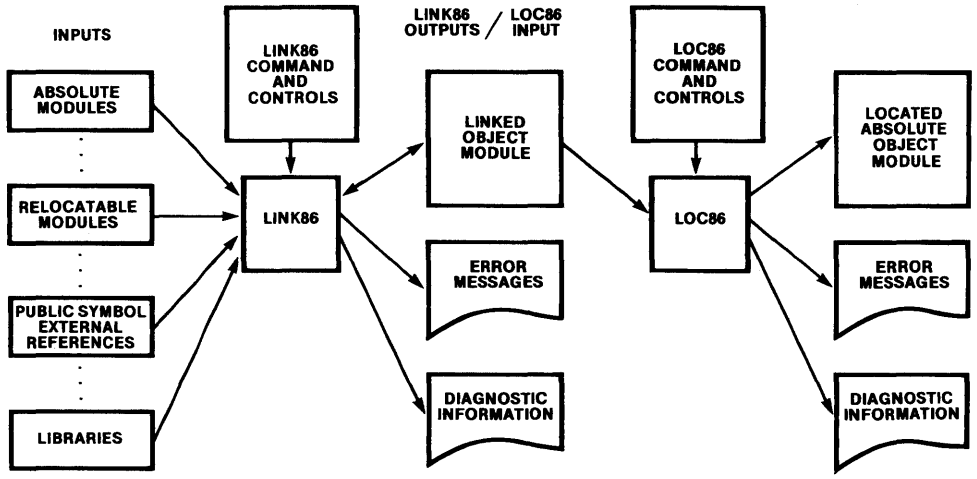


Figure 1-3. The LINK86/LOC86 Process 639-3

it each time the system is turned on. The data that is referenced often is kept in high-speed RAM because it is modified frequently. It may be practical to keep data that is referenced less often in slower-speed memory. The size and composition of a system's memory is totally dependent on the application the system serves.

Linkage and relocation is designed to handle the linking and locating of your program, no matter how your 8086-based system memory is implemented. It provides very flexible segment placement within any given memory configuration.

8086 Addressing Techniques

The 8086 addresses memory with a 20-bit address that is constructed from a segment address and a 16-bit offset from that segment address. This means that with a single segment address, 64K bytes of memory is directly addressable by only changing the offset.

A hardware segment address is a 20-bit address. But the segment address is constrained such that the segment is placed on a boundary that is a multiple of 16 (10H). The segment address can be set to any hexadecimal address ending in 0:

```
    0H
  010H
  020H
    .
    .
    .
0FFFF0H
```

Because the low four bits of the 20-bit segment address are always zero, the segment address can be represented with only 16 bits.

The segment address is kept in one of four 16-bit segment registers. Because there are four segment registers, the 8086 can, at any moment, access 256K (4 × 64K) bytes of memory. The full megabyte of memory is accessed by changing the values in the segment registers. Figure 1-4 shows the 8086 addressing concept.

Segments

Programs are comprised of pieces called segments, which are the fundamental units of linkage and relocation. The basic divisions have functional purposes related to the hardware configuration of memory. The portions of programs that are to be kept in ROM or PROM can be put in separate segments from the portions that will be kept in RAM.

The 8086 Assembler allows the programmer to name the segments of the program being developed. The PL/M86 compiler generates predefined names for segments (see Appendix C).

A segment is a contiguous area of memory that is defined at translation time (assemble or compile). When defined, a segment does not necessarily have a fixed address or size. A fixed address is assigned to a segment during the locate function. The size can be changed by combining segments and by a control that specifies a specific size. Some translations may produce absolute object information, with absolute addresses and a specific segment size.

LINK86 combines all segments with the same complete (segment and class) name and combination type (memory, stack, etc.) from all input modules. The ordering of segments is done on the basis of these combined segments. The manner in which

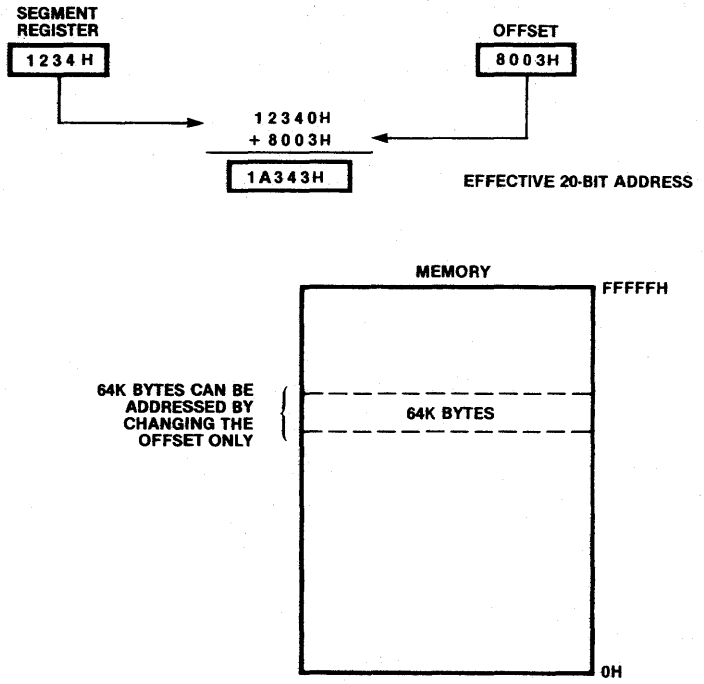


Figure 1-4. 8086 Addressing 639-4

segments are combined depends on the alignment of the segments (which is described in the next topic) and a combining attribute associated with the segment.

When we refer to combining segments, we are talking about how the segments will be loaded in memory, *not* how they will be stored in the output module. The segments in the LOC86 output module contain addresses that determine where they will be loaded in memory. The segments reside in the output module in the same order as they were in the input modules. Figure 1-5 shows the physical relationships between the input modules, output module, and loaded program.

Segment Alignment

A segment can have one (and in the case of the inpage attribute, two) of five alignment attributes:

- Byte, which means a segment can be located at any address.
- Word, which means a segment can only be located at an address that is a multiple of two, starting from address 0H.
- Paragraph, which means a segment can only be located at an address that is a multiple of 16, starting from address 0.
- Page, which means a segment can only be located at an address that is a multiple of 256, starting from address 0.
- Inpage, which means a segment can be located at whichever of the preceding attributes apply plus must be located so that it does not cross a page boundary.

Figure 1-6 shows the segment alignment boundaries.

Any alignment attribute except byte can result in a gap between combined segments. For example, when two page aligned segments are combined there will always be a gap unless the first happens to be an exact multiple of 256 bytes in length.

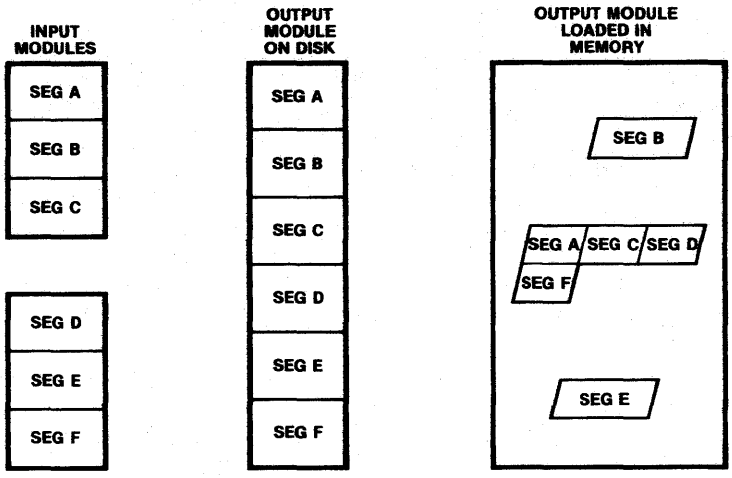


Figure 1-5. Segment Physical Relationships 639-5

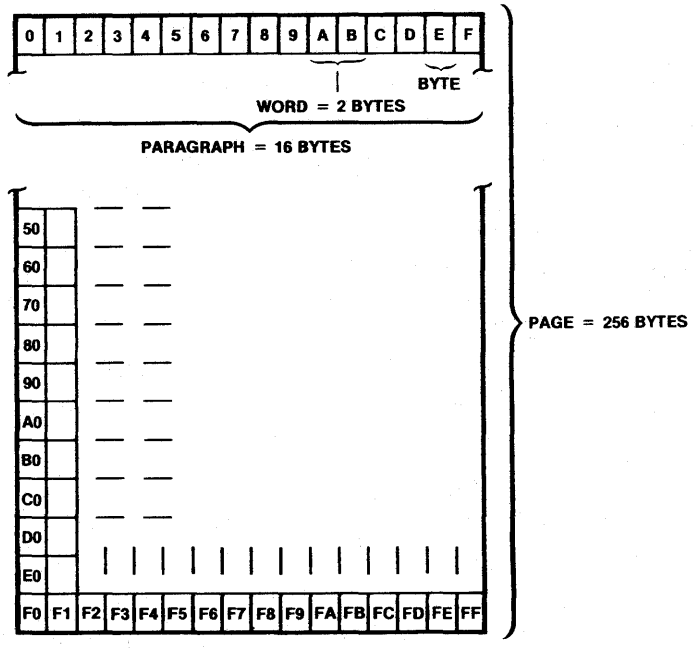


Figure 1-6. Segment Alignment Boundaries 639-6

Segment Combining

Segments containing data and code are combined end to end. There may be a gap between the segments if the alignment characteristics require it. The relative addresses in the segments are adjusted for the new longer segment.

There are two special cases of segment combination: stack segments and memory segments. PL/M86 defines these segments with the names STACK and MEMORY. With ASM86 you must define them by adding the STACK or MEMORY parameter to the SEGMENT directive.

When stack segments are combined, they are overlaid but their lengths are added together.

When memory segments are combined, they are overlaid with their low addresses at a common address. The length of the combined memory segment is the length of the largest segment that was combined. No relative address adjusting is necessary. Normally the memory segment is located above (at a higher memory address) the rest of the program segments if no controls are used to override this.

To make sure that stack segments are combined correctly, you should always give them the same segment name in each module. The same is true of memory segments. If you are going to link assembly language routines to PL/M-86 routines you should give them the names STACK and MEMORY to be compatible with PL/M-86.

Segment Locating

Segments are located in the order in which they are encountered in the input modules. If classes (described in the next section) are defined, the segments from a class are located together. The locating algorithm can be changed by using the locating controls that are described with the commands in Chapter 3 . Detailed descriptions of how segments are located with and without controls are in Chapter 4.

One variation to the sequential locating of segments is how the MEMORY segment is located. When the first segment with the memory attribute is encountered, it is placed last in the list of segments. This means that after all other segments are located, the MEMORY segment will be assigned the highest address in the output module.

NOTE

The MEMORY segment may not be located at the top of the module if its name or class name appears in any LOC86 control (other than SEGSIZE) or it has the absolute attribute.

Classes

A class is a collection of segments. When segments are defined in assembly language, a class name can be specified. The segments generated by PL/M-86 are generated with predefined class names. (See Appendix C for a description of PL/M-86 generated segments and classes.) Any number of segments can be given the same class name. Class names can extend beyond module boundaries; the same class name can be used in different modules that are to be combined.

The primary purpose of classes is to collect together (in an arbitrary order) segments that share a common attribute and to manipulate this collection at locate-time by specifying only the class name.

All segments with the same class name are located together in the memory address space of the output module. (You can override class collection by specifying the location of segments with the LOC86 ORDER control or LOC86 ADDRESSES control.)

Classes give you a second means of collecting like segments in the output module. The first is giving segments the same name. If you are developing several modules that are to be combined, you may want to give the segment containing executable code the name CODE in each module. If there are several differently named segments within a module that contain executable code, you may want to give these segments the class name of CODE which causes them to be located together but not combined. (The same name can be used for segments and classes.)

Groups

A group is also a collection of segments. Groups define addressing range limitations in 8086 object modules. A group specifies a collection of segments that must be located within a 64K byte range. This means that the entire group of segments can be addressed with offsets from a single segment register. Or, to put it another way, the segment register need not be changed when addressing any segment in a group. This permits efficient addressing within the module.

Group addressing always begins at an address that is a multiple of 16 (i.e., a paragraph boundary). R&L does not manipulate segments of a group to make sure they fall within a 64K byte range. However, if they do not fit in the range, a warning message is issued.

The segments included in a group do not have to be contiguous in the output module. The only requirement is that all the segments defined in the group must totally fall within 64K bytes of the beginning address of the group.

CHAPTER 2
LINK86 COMMAND

The LINK86 command combines separately translated 8086 modules into a single module. These modules can be produced by the 8086 assembler, 8088 assembler, 8089 assembler, or PL/M-86. LINK86 performs the following:

- Combines segments with the same complete name.
- Resolves external and public references between the input modules.
- Selects modules from specified libraries to resolve external references.
- Provides the facility to rename groups.
- Optionally purges public symbol, local symbol, line number, and comment definitions from the output module.
- Generates a link map that summarizes the link process.
- Generates error messages for abnormal conditions encountered during processing.
- Combines groups with the same name.

In combining modules, LINK86 only resolves symbolic addressing between parts of a program. It is up to the developer to make sure that the parts work together. LINK86 cannot check anything in this area.

The command signs on with the message:

```
ISIS-II MCS-86 LINKER, Vx.y
```

to the current output console device. This message is also placed in the file designated for the link map, if it is different than the console device.

The LINK86 command combines the input modules into a single output module. This output module is the major output of the command. Secondary outputs consist of the link map and informational, warning, and error messages.

Messages go to the current console output device and to the file designated for the link map, if different than the console device.

The format of the LINK86 command is:

LINK86 *inputlist* [TO *outputfile*][*controls*][;*comment*]

where:

LINK86 is the command name.

inputlist is a list of items that contain the modules or symbols to be linked. Multiple items are separated with commas (,). The items in the *inputlist* must be in either of the following form:

filename[(*modname*[,...])]

or

PUBLICSONLY(*filename*[,...])

filename specifies a single file that contains one or more object modules or a library (created by the LIB86 command) of object modules. If the specified file does not contain a library and *modname* is not entered, all modules are unconditionally included in the output module.

If the file contains individual modules or a library and *modnames* are specified, only the modules specified are included in the output module. If the file contains a library and *modnames* are not specified, only those modules that satisfy an external reference from preceding object modules or from the library itself are included in the output module. For example, if the following *inputlist* is specified:

```
:F1:IRON.OBJ,STEEL.LIB,:F2:ORE.OBJ
```

and:

- :F1:IRON.OBJ has a single module that has an external reference to a public symbol in the module named CARB in STEEL.LIB.
- the module CARB has an external reference to the module STAIN which is also in STEEL.LIB.
- :F2:ORE.OBJ has a single module that has an external reference to the module SMELT in STEEL.LIB.

the output module will contain:

```
:F1:IRON.OBJ
CARB
STAIN
:F2:ORE.OBJ
```

and the reference to SMELT in ORE.OBJ will be unsatisfied because the library that contains SMELT was specified before ORE.OBJ. To satisfy all external references, the *inputlist* should have been specified in the following order:

```
:F1:IRON.OBJ,:F2:ORE.OBJ,STEEL.LIB
```

PUBLICSONLY indicates that the absolute public symbols in the specified files will be used to resolve external references only. The modules will not be included in the output module.

TO *outputfile* is an optional parameter that specifies the name of the file that is to contain the linked module. If **TO** *outputfile* is omitted, LINK86 uses the device and root portion of the first *filename* specified in the *inputlist*. The extension portion of the name (if any) is dropped and LNK is substituted. The *filename* specified in **TO** *outputfile* or defaulted must not be referenced in the *inputlist*. If a file by that name already exists on the specified device, it will be overwritten. The output module file must exist on a randomly accessible device such as a disk drive.

controls specifies one or more of the optional LINK86 command controls that modify the operation of the command. There are three categories of controls: input list, diagnostic controls and output module controls. The following is a table of the controls (the input list control may not appear in *controls*):

Table 2-1. LINK86 Controls

Control	Abbrevi.	Default
Input List Controls		
PUBLICSONLY(*filename*[, ...])	PO	Not applicable.
Diagnostic Controls		
MAP	MA	MAP
NOMAP	NOMA	
PRINT[(*filename*)]	PR	PRINT
NOPRINT	NOPR	
Output Module Controls		
COMMENTS	CM	COMMENTS
NOCOMMENTS	NOCM	
LINES	LI	LINES
NOLINES	NOLI	
NAME(*modname*)	NA	Name of first module in input.
PUBLICS	PL	PUBLICS
NOPUBLICS	NOPL	
PURGE	PU	NOPURGE
NOPURGE	NOPU	
RENAMEGROUPS(*groupname* TO *groupname*[, ...])	RG	Not applicable.
SYMBOLS	SB	SYMBOLS
NOSYMBOLS	NOSB	
TYPE	TY	TYPE
NOTYPE	NOTY	

;*comment* is an optional field that can be used for any purpose. The comment must always be preceded by a semicolon or an ampersand. (The ampersand is the continuation character and its use for continuation and commenting is described in the next section.) The semicolon can occur anywhere a space or delimiter can occur. It cannot be entered in the middle of a keyword or value. The comment extends to the end of the line. This facility lets you put comments on the permanent record of the link, if you use a hard copy PRINT file. A line with a ;*comment* cannot be continued. However, a comment can be added to a line that is being continued (see the following section).

If the same control appears more than once in a control list (e.g., "PRINT(:LP:)" and "PRINT(FOO)", or "LINES" and "NOLINES"), then the rightmost control is the only one with effect.

Continuation Lines

A LINK86 command with several input modules and controls might not fit on a single terminal input line (which may not exceed 122 characters). You can continue a LINK86 command to the next line by entering an ampersand (&) before the carriage return. Anything entered between the ampersand and the carriage return is treated as a comment. The system prompts for the next line with two asterisks (**). An input line may not be broken within a keyword or value. Lines can be broken anywhere a parenthesis, comma, or space is normally used. Note that the line break does not replace the parenthesis or comma; it can however, replace a space. Any number of continuation lines are allowed. The following is an example of line continuation:

```
LINK86 :F1:IRON.OBJ,ORE.OBJ,STEEL.LIB & <CR>
** TO :F1:METAL.LNK PRINT(:LP:) MAP & <CR>
** NOCOMMENTS<CR>
```

LINK86 Command Controls

There are three categories of LINK86 controls, those that affect the input list, the diagnostic output information and those that affect the output module.

Input List Control

The input list control gives you the capability to create overlay files. You can resolve external references without including the input module that contain the public symbols in the output module.

PUBLICSONLY Control

The PUBLICSONLY control specifies that only the absolute public symbols located will be used from the specified input modules. The input modules will not be included in the output file. To speed up linkage, you may create a file containing only absolute public symbols by specifying only the input control in the input list.

The format of the PUBLICSONLY control is:

PUBLICSONLY(*filename*[, ...])

filename specifies the located file whose public symbols will be linked to the external references in the inputlist, but the module itself will not be included in the output.

filename cannot be a library file.

PUBLICSONLY can be abbreviated PO.

PUBLICSONLY Control Example. To include the absolute public symbols of two overlay files (:F1:OVL.2 and :F1:OVL.3) in a temporary file.

```
LINK86 PUBLICSONLY(:F1:OVL.2,:F1:OVL.3) TO :F1:PSYMB.TMP
```

For a more complete discussion of overlays, see Chapter 4.

Diagnostic Controls

The diagnostic controls give you the capability of controlling where the diagnostic output is to be directed.

MAP Control

The MAP control specifies whether a link map is to be included in the diagnostic output file. The link map consists of six parts:

- a copy of the command tail that invoked the LINK86 command.
- the warning messages issued by LINK86 (if any).
- a header that contains the filename and module name of the output module.
- a list of the segments in the output module along with their length, class, and address (if they are absolute).
- a list of the input modules.
- a list of unresolved external names (if any).

The LINK86 command does not detect, and thus, the link map does not show:

- gaps between segments.
- conflicts between absolute segments.

This last piece of information can be obtained from the LOC86 command.

The following is an example of a link map:

```
ISIS-II MCS-86 LINKER, Vx.y, INVOKED BY:
LINK86 :F3:SHIP.OBJ,PROG.LIB(TRIG),PROG.LIB TO :F1:SHIP.WRK &
NAME(CELESTIGATION) MAP PRINT(:LP:) NOPUBLICS&
EXCEPT(SIN,COS)

warning messages may appear here

LINK MAP FOR SHIP.WRK(CELESTIGATION)

LOGICAL SEGMENTS INCLUDED:

          LENGTH   ADDRESS   SEGMENT              CLASS
          2345H    ------    CODE                 CODE
            FAH    ------    CONST                CONST
            ABH    ------    DATA                 DATA
            32H    ------    STACK                STACK
             0H    ------    MEMORY               MEMORY
           145H    32000H    ABSOLUTE@CODE        ACODE

INPUT MODULES INCLUDED:
    SHIP.OBJ (NAVAL_PACKAGE)
    PROG.LIB (TRIG)
    PROG.LIB (EXP1)
    PROG.LIB (EXP2)
```

```
UNRESOLVED EXTERNAL NAMES:
    COSIGN IN SHIP.OBJ (NAVAL__PACKAGE)
    COSIGN IN PROG.LIB (EXP2)
    SIGN IN PROB.LIB (EXP2)
```

warning messages may appear here

The format of the MAP control is:

MAP | NOMAP

MAP specifies that a link map is to be produced. This is the default if no control is specified. **MAP** can be abbreviated **MA**. Note that no map is produced if the NOPRINT control (described next) is coded.

NOMAP specifies that a link map is not to be produced. **NOMAP** can be abbreviated **NOMA**.

MAP Control Examples. To explicitly specify that a link map be produced:

```
MAP
```

To specify that no link map is to be produced:

```
NOMAP
```

PRINT Control

The PRINT control specifies where the diagnostic output information is to be sent. If no control is specified, the diagnostic output goes to a disk file with the same device and filename as the module output file but with an extension of MP1. The link map (if requested) and any error messages are sent to this file. The error messages are also sent to the current console output file (if it is different from the file specified in the PRINT control).

The format of the PRINT control is:

PRINT[(*filename*)] | **NOPRINT**

PRINT specifies that an output diagnostic file is to be created. If *filename* is specified, the diagnostic output file is created on the specified file. If *filename* is omitted, the diagnostic output file is created on the disk with the same device and root filename as the output module file and an extension of MP1. *filename* cannot be the name of any file named in the command input list or in the command TO phrase. **PRINT** can be abbreviated **PR**.

NOPRINT specifies that no diagnostic output file is to be generated. If **NOPRINT** is specified, there will be no diagnostic output other than error messages to the current console output device, even if the MAP control is specified. **NOPRINT** can be abbreviated **NOPR**.

PRINT Control Examples. To explicitly specify that the diagnostic output file is to be sent to a disk file with the same name as the module output file but with an extension of MP1:

```
PRINT
```

To send the diagnostic output file to the line printer:

```
PRINT(:LP:)
```

To specify that no diagnostic output be generated:

```
NOPRINT
```

Output Module Controls

The output module controls give you the capability of controlling the content of the output module. You can specify the name of the output module, can change the names of groups in the module, and can specify whether public symbols, line numbers, local symbols, and comments are to be included.

NAME Control

The NAME control specifies a module name for the output module. If the NAME control is omitted, the output module name is taken from the first module encountered in the input.

The format of the NAME control is:

NAME(*modulename*)

modulename specifies the name to be assigned to the output module. The *modulename* can be 1 through 40 characters in length. It can be composed of capital letters (A through Z), digits (0 through 9), question mark (?), colon (:), period (.), underscore (_), and commercial at sign (@). LINK86 automatically converts lower-case letters in the command line to upper-case.

NAME can be abbreviated NA.

NAME Control Example. To assign the name OUTPUT@MOD@3 to the output module:

```
NAME(OUTPUT@MOD@3)
```

RENAMEGROUPS Control

The RENAMEGROUPS control lets you change the name of groups in the output module. The control has no effect on the modules in the input list. The control can contain a list of from-to changes. The changes take effect in a left-to-right order. Thus you should be careful not to change a group name that you have already changed.

The format of the RENAMEGROUPS control is:

RENAMEGROUPS(*oldname* TO *newname*[,...])

oldname is the name of a group to be changed. The *oldname* can be 1 through 40 characters. It can be composed of capital letters (A through Z), digits (0 through 9), question mark (?), colon (:), period (.), underscore (_), and commercial at sign (@).

newname is the name to be assigned to the group. It can be from 1 through 40 characters and can be composed of the same characters as *oldname*.

RENAMEGROUPS can be abbreviated **RG**.

RENAMEGROUPS Control Example. To change the names of the groups CGROUP and DGROUP to CODEGP and DATAGP in the output module:

```
RENAMEGROUPS(CGROUP TO CODEGP,DGROUP TO DATAGP)
```

LINES Control

The LINES control specifies whether line numbers from the input modules are to be included in the output module.

The format of the LINES control is:

LINES | NOLINES

LINES specifies that line numbers are to be included in the output module. **LINES** is the default if no control is specified. **LINES** can be abbreviated **LI**.

NOLINES specifies that no lines numbers are to be included in the output module. **NOLINES** can be abbreviated **NOLI**.

LINES Control Examples. To explicitly specify that line numbers are to be included in the output module:

```
LINES
```

To specify that line numbers are to be excluded from the output module:

```
NOLINES
```

COMMENTS Control

The COMMENTS control specifies whether comments from the input modules are to be included in the output module.

The format of the COMMENTS control is:

COMMENTS | NOCOMMENTS

COMMENTS specifies that comments are to be included in the output module. **COMMENTS** is the default if no control is specified. **COMMENTS** can be abbreviated **CM**.

NOCOMMENTS specifies that no comments are to be included in the output module. **NOCOMMENTS** can be abbreviated **NOCM**.

COMMENTS Control Examples. To explicitly specify that comments are to be included in the output module:

```
COMMENTS
```

To specify that comments are to be excluded from the output module:

```
NOCOMMENTS
```

SYMBOLS Control

The SYMBOLS control specifies whether local symbols from the input modules are to be included in the output module.

The format of the SYMBOLS control is:

SYMBOLS | NOSYMBOLS

SYMBOLS specifies that local symbols are to be included in the output module. **SYMBOLS** is the default if no control is specified. **SYMBOLS** can be abbreviated **SB**.

NOSYMBOLS specifies that no local symbols are to be included in the output module. **NOSYMBOLS** can be abbreviated **NOSB**.

SYMBOLS Control Examples. To explicitly specify that local symbols are to be included in the output module:

```
SYMBOLS
```

To specify that local symbols are to be excluded from the output module:

```
NOSYMBOLS
```

PUBLICS Control

The PUBLICS control specifies whether public symbols from the input modules are to be included in the output module. The PUBLICS controls gives you the capability of selectively including or excluding public symbols from the output module.

The format of the PUBLICS control is:

PUBLICS |NOPUBLICS [**EXCEPT(***publicsymbol*[,...])]

PUBLICS specifies that public symbols are to be included in the output module. **PUBLICS** is the default if no control is specified. **PUBLICS** can be abbreviated **PL**.

NOPUBLICS specifies that no public symbols are to be included in the output module. **NOPUBLICS** can be abbreviated **NOPL**. **EXCEPT** specifies one or more public symbols that are not to be affected by the control. If an **EXCEPT** parameter follows a PUBLICS control, all public symbols are included in the output module except those specified in the **EXCEPT** parameter. Likewise, if an **EXCEPT** parameter follows a **NOPUBLICS** control, all public symbols are excluded from the output module except those specified in the **EXCEPT** parameter. **EXCEPT** can be abbreviated **EC**.

PUBLICS Control Examples. To explicitly specify that public symbols are to be included in the output module:

```
PUBLICS
```

To specify that public symbols are to be excluded from the output module:

```
NOPUBLICS
```

To specify that all public symbols except COSINE, SINE, and TANG are to included in the output module:

```
PUBLICS EXCEPT(COSINE,SINE,TANG)
```

PURGE Control

The **PURGE** Control is a combination control. **PURGE** is exactly equivalent to coding "NOLINES NOSYMBOLS NOCOMMENTS NOPUBLICS NOTYPE". The format of the PURGE control is:

PURGE | NOPURGE

NOPURGE is exactly equivalent to coding "LINES SYMBOLS COMMENTS PUBLICS TYPE".

TYPE Control

The **TYPE** control indicates that type checking on external and public symbols is to be done. The format of the **TYPE** control is:

TYPE | NOTYPE

NOTYPE indicates that type checking on external and public symbols is not to be done. **NOTYPE** will not specify a symbol type in the output.

CHAPTER 3
LOC86 COMMAND

The LOC86 command binds relocatable 8086 modules to absolute addresses. Additionally, the command:

- Creates an absolute output object module from a single input object module.
- Generates a memory map that summarizes the results of the address binding.
- Generates a symbol table that shows the address assignments for translator produced symbols.
- Detects and lists errors found in the input module and the LOC86 command itself.
- Filters relocating information and translator generated debugging information.

The input to the LOC86 command is a single file that contains a single module. The input is usually the output module from LINK86. However, it can be the output of a translation.

The LOC86 absolute output module is usually the input to an 8086 loader or OH86, but can be input to other products and their loaders.

The command signs on with the message:

```
ISIS-II MCS-86 LOCATOR, Vx.y
```

to the current output console device. A longer form of this message is also placed in the file designated for the diagnostic information, if it is different than the console output device. The longer form of the message is:

```
ISIS-II MCS-86 LOCATOR, Vx.y INVOKED BY:
LOC86 parameters
```

Any messages generated by the command and any diagnostic information requested with the command are sent to the file designated for the diagnostic information. The messages are also sent to the console output device, if it is different than the diagnostic information file.

The format of the LOC86 command is:

LOC86 *inputfile* [**TO** *outputfile*][*controls*][;*comment*]

where:

LOC86 is the command name.

inputfile is an ISIS-II file that contains the module to be located.

TO *outputfile* is an optional parameter that specifies the name of the file that is to contain the located module. If **TO** *outputfile* is omitted, LOC86 uses the name portion of the *inputfile*, dropping the extension. If *inputfile* has no extension, an error message is issued unless **TO** is specified.

controls specifies one or more of the optional LOC86 command controls that modify the operation of the command. There are three categories of controls:

- Diagnostic controls which control the output and content of the diagnostic information.
- Output module controls which control the output module.
- Diagnostic and output module controls which can be applicable to the diagnostic output or the output module or both.

The following is a table of the controls:

Table 3-1. LOC86 Controls

Control	Abbrevi.	Default
Diagnostic Controls		
MAP	MA	MAP
NOMAP	NOMA	
PRINT[(*filename*)]	PR	PRINT
NOPRINT	NOPR	
SYMBOLCOLUMNS(*value*)	SC	SYMBOLCOLUMNS(2)
Output Module Controls		
ADDRESSES(SEGMENTS(*segment*[/*class*](*addr*)[,...])| CLASSES(*class*(*addr*)[,...])| GROUPS(*group*(*addr*)[,...]) [,...])	AD SM CS GR	Not applicable
BOOTSTRAP	BS	Not applicable
NAME(*modname*)	NA	Name of first module in input
ORDER(SEGMENTS(*segment*[/*class*][, ...]| CLASSES(*class*[(*segment*[, ...])] [,...])	OD SM CS	Order of appearance in input
RESERVE (*addr* TO *addr*[,...])	RS	Not applicable
SEGSIZE(*segment*[/*class*] (*value*)[,...])	SS	Actual size of segment
START({ *publicsymbol*| *paragraph,offset* })	ST	Start address taken from the input module

Table 3-1. LOC86 Controls (Cont'd)

Control	Abbrevi.	Default
Diagnostic and Output Module Controls		
OBJECTCONTROLS(*controls*)	OC	Diagnostic and Output module controls apply to both the output file and the diagnostic output file.
PRINTCONTROLS(*controls*)	PC	Diagnostic and Output module controls apply to both the output file and the diagnostic output file.
COMMENTS	CM	COMMENTS
NOCOMMENTS	NOCM	
LINES	LI	LINES
NOLINES	NOLI	
PUBLICS	PL	PUBLICS
NOPUBLICS	NOPL	
PURGE	PU	NOPURGE
NOPURGE	NOPU	
SYMBOLS	SB	SYMBOLS
NOSYMBOLS	NOSB	

;*comment* is any additional information you may want to enter into the hard copy of the LOCATE map. Anything entered between a semicolon (;) and the carriage return is treated as a comment.

Continuation Lines

A LOC86 command with several controls might not fit on a single terminal input line (which may not exceed 122 characters). You can continue a LOC86 command to the next line by entering an ampersand (&) before the carriage return. Anything entered between the ampersand and the carriage return is treated as a comment. The LOC86 command prompts for the next line with two asterisks (**). An input line may not be broken within a keyword or value. A line can be broken anywhere a parenthesis, comma, or space is normally used. Note that the line break does not replace the parenthesis or comma; it can, however, replace a space. Any number of continuations are allowed. The following is an example of line continuation:

```
LOC86 :F1:IRON.LNK TO STEEL &
** MAP PRINT(:F2:IRON.OUT) OBJECTCONTROLS(NOCOMMENTS) &
** PRINTCONTROLS(COMMENTS,NOPUBLICS)<CR>
```

LOC86 Command Controls

There are three categories of LOC86 controls: those that affect the diagnostic output information, those that affect the output module, and those that affect either or both diagnostic output or output module. Included in the last category are two controls that limit certain controls to either the diagnostic output or the output module.

If the same control appears more then once in a control list (e.g., "PRINT(:LP:)" and "PRINT(FOO)", or "LINES" and "NOLINES"), then the rightmost control is the only one with effect. The only exception is that when a control appears within a PRINTCONTROLS or OBJECTCONTROLS control scope, it does not cancel other controls.

Diagnostic Controls

There are three controls in this category: MAP, PRINT, and SYMBOL-COLUMNS.

MAP Control

The MAP control specifies whether a memory map is to be included in the diagnostic output file. The memory map consists of three parts:

- general module information such as module name, input and output files, and start address.
- a segment map that includes segment start and stop addresses, length, alignment attribute, name, and class name.
- a group map that includes group names, starting addresses, and component segment names.

The memory map also shows where there are conflicts between individual segments. A conflict is shown by the character C between the stop address and length in the conflicting segment entry.

The following is an example of a memory map:

```
MEMORY MAP OF MODULE CELESTIGATION
READ FROM FILE :F1:SHIP.LNK
WRITTEN TO FILE :F1:SHIP
MODULE START ADDRESS  PARAGRAPH = 020H  OFFSET = 0F8H

SEGMENT MAP

START     STOP      LENGTH     ALIGN     NAME      CLASS

00200H    00A40H      840H       W       CODE      CODE
00A42H    00B42H      100H       W       CONST     CONST
00B44H    00D50H      20BH       B       DATA      DATA
00D52H    00D58H        6H       W       STACK     STACK
00D5AH    00D5AH        0H       W       MEMORY    MEMORY

GROUP MAP

ADDRESS GROUP OR SEGMENT NAME

00200H      CGROUP
            CODE
00A40H      DGROUP
            CONST
            DATA
            STACK
            MEMORY
```

The map shows the locations of segments, groups, and classes. The address and lengths are always given in hexadecimal. The ALIGN column specifies the alignment attribute of the segments. The alignment codes are:

B — byte
W — word
G — paragraph
xR — inpage
P — page
A — absolute

Note that the x in the inpage alignment code can be any other alignment code. That is, a segment can have the inpage attribute, meaning it must reside within a 256 byte page and can have the word attribute, meaning it must reside on an even numbered byte.

The format of the MAP control is:

MAP | NOMAP

MAP specifies that a map be produced and sent to the diagnostic output file specified by the PRINT control. This is the default condition. **MAP** can be abbreviated **MA**.

NOMAP specifies that no map be produced. **NOMAP** can be abbreviated **NOMA**.

MAP Control Example. To explicitly generate a memory map:

```
MAP
```

To specify that no map is to be produced:

```
NOMAP
```

PRINT Control

The PRINT control specifies where the diagnostic output information is to be sent. If no control is specified, the diagnostic output goes to a disk file with the same device and name as the output module file and an extension of MP2. The memory map (if not suppressed), any error messages, and any other diagnostic information are sent to this file. The error messages are also sent to the current console output file (if it is different than the file specified in the PRINT control).

The format of the PRINT control is:

PRINT[(*filename*)] | **NOPRINT**

PRINT specifies that a diagnostic output file is to be created. If *filename* is specified, the specified file is created. If *filename* is omitted, the file is created on the same disk that holds the output module. The file has the same name as the output module and an extension of MP2. *filename* cannot be the name of a file named elsewhere in the command. **PRINT** (with no filename) is the default condition. **PRINT** can be abbreviated **PR**.

NOPRINT specifies that no diagnostic output file is to be created. If NOPRINT is specified, there will be no diagnostic output other than error messages to the current console output device, even if requested with specific controls (such as MAP). **NOPRINT** can be abbreviated **NOPR**.

PRINT Control Examples. To explicitly specify that the diagnostic output file is to be sent to a disk file with the same name as the module output file and an extension of MP2:

```
PRINT
```

To send the diagnostic output file to the line printer:

```
PRINT(:LP:)
```

To specify that no diagnostic output is to generated:

```
NOPRINT
```

SYMBOLCOLUMNS Control

The SYMBOLCOLUMNS control specifies the number of columns the symbol table is formatted into. The symbol table is described later in this chapter.

The format of the SYMBOLCOLUMNS control is:

SYMBOLCOLUMNS(*value*)

SYMBOLCOLUMNS can be abbreviated **SC**.

value specifies the number of columns into which the table is to be formatted. If the **SYMBOLCOLUMNS** control is not entered, the number of columns defaults to 2. The possible values for *value* are 1, 2, 3, or 4.

Output Module Controls

There are seven controls in the output module category. They control the content of the output module, the order of the segments in the module, and the assignment of addresses to the segments.

ADDRESSES Control

The ADDRESSES control specifies addresses for specific segments, classes, and groups named in the control. The control is entered with any or all of three subcontrols: SEGMENTS, CLASSES, and GROUPS. All segment names for which addresses are supplied must be included in a SEGMENTS list and likewise for classes and groups. The subcontrols can be specified multiple times in an ADDRESSES control.

When ADDRESSES is used to assign an address to a class, the specified address is assigned to the segment belonging to the class that resides lowest in memory.

If an address is specified that causes a conflict with a segment specified in an ORDER control, an error message is issued and processing is terminated. For example, the controls:

```
ORDER(SEGMENTS(A,B)) ADDRESSES(SEGMENTS(A(500H),B(400H)))
```

specify that segments A and B be located in the sequence A,B but the addresses specified for them are in the opposite order, B,A.

The format of the ADDRESSES control is:

```
ADDRESSES( SEGMENTS(segment[/class](addr)[,...]) |
           CLASSES(class(addr)[,...]) |
           GROUPS(group(addr)[,...])[,...])
```

The optional class specification ensures that the LOC86 command will locate the proper segment in cases where segments in different classes have the same name.

ADDRESSES can be abbreviated **AD**.

SEGMENTS specifies that a list of one or more segment names is included. Each segment name must be followed by an address enclosed in parentheses. Multiple entries in the SEGMENTS list are separated with commas. **SEGMENTS** can be abbreviated **SM**.

segment is the name of a relocatable segment in the module being processed.

/class is the name of the class to which a segment belongs, if any.

addr is an address entered in any base to which the preceding segment is to be bound. The address can be in the range 0H through 0FFFFFH.

CLASSES specifies that a list of one or more class names is included. Each class name must be followed by an address enclosed in parentheses. Multiple entries in the CLASSES list are separated with commas. **CLASSES** can be abbreviated **CS**.

class is the name of a class in the module being processed.

GROUPS specifies that a list of one or more group names must be located at the specified addresses. Each group name should be followed by a paragraph address (if not LOC86 will round down to a paragraph address) enclosed in parentheses. Multiple entries in the GROUPS list must be separated by commas. GROUPS may be abbreviated GR.

group is the name of a group in the module being processed.

To see how the GROUPS subcontrol may be used to help create overlay modules, see Chapter 4.

The LOC86 command is designed to detect conflicts whenever possible. However, it is possible that you may want to cause a conflict in locating some segments. To intentionally cause a conflict, you must specify the relocatable segments in an ADDRESSES/SEGMENTS control.

When an address is specified for a specific segment with the ADDRESSES control, the segment is located exactly where specified. When CLASSES is used, the first segment in the class is located at the specified address or at the first available address after the specified address.

For example, if there is an absolute segment, 100H bytes in length, located at address 800H and a LOC86 command with the control:

```
ADDRESSES(CLASSES(RAM(800H)))
```

is issued, the first segment in the class RAM will be located at address 900H (or higher, depending on the alignment attribute). The command does not allow a conflict at this point. If a conflict was allowed, it would not be known what segment will be involved in the conflict because the segments in RAM will be located in a first come, first served manner.

If the ADDRESSES control specified a segment instead of a class, a conflict would have been allowed.

ADDRESSES Control Examples. To specify that segment Code is to be located at address 550H:

```
ADDRESSES(SEGMENTS(CODE(550H)))
```

To specify that class RAM is to be located at address 400H:

```
ADDRESSES(CLASSES(RAM(400H)))
```

To specify that segments S1 and S4 are to be located at addresses 2000H and 03BF0H, respectively, and that class ROM be located at address 0C000H:

```
ADDRESSES(SEGMENTS(S1(2000H),S4(03BF0H)),CLASSES(ROM(0C000H )))
```

To specify that segments S1 and S4, of classes RAM and CODE, respectively, are to be located at addresses 050H and 00F4H, respectively:

```
ADDRESSES(SEGMENTS (S1 / RAM (050H), S4 / CODE(00F4H)))
```

BOOTSTRAP Control

The BOOTSTRAP control specifies that a long jump instruction to the module's start address be stored at location 0FFFF0H through 0FFFF4H. This control establishes a means of restarting a program when the microprocessor is reset. If no start address is specified in the input module or with the START control, an error message is issued.

The format of the BOOTSTRAP control is:

BOOTSTRAP

BOOTSTRAP can be abbreviated **BS**.

BOOTSTRAP Control Example. To cause a jump instruction to the start address to be stored at location 0FFFF0H:

```
BOOTSTRAP
```

NAME Control

The NAME control specifies a module name to be included in the output module. If the NAME control is omitted, the first module name encountered in the input module is used as the output module name.

The format of the NAME control is:

NAME(*modulename*)

NAME can be abbreviated **NA**.

modulename specifies the name to be assigned to the output module. The *modulename* can be 1 through 40 characters. It can be composed of capital letters (A through Z), digits (0 through 9), question mark (?), colon (:), period (.), underscore (_), and commercial at sign (@). LOC86 automatically converts lower-case letters to upper-case.

NAME Control Example. To assign the name OUTPUT_MODU_44 to the output module:

```
NAME(OUTPUT_MODU_44)
```

ORDER Control

The ORDER control specifies the order in which segments and classes are to be located in the output module. The ORDER control does not specify addresses for the segments or classes. The addresses are determined by LOC86 or assigned with the ADDRESSES control.

There are two subcontrols that are used with the ORDER control: SEGMENTS and CLASSES. SEGMENTS precedes a list of one or more segments. CLASSES precedes a list of one or more classes. The subcontrols can be used multiple times in an ORDER control. For example, you may want to specify that three segments are to be located before a certain class and the class is to be followed by two more segments. This will require one CLASSES subcontrol and two SEGMENTS subcontrols.

The order of the segments within a class specified with the CLASSES subcontrol can be specified with the order control.

If an order specified with the ORDER control causes a conflict with a segment or class specified in an ADDRESSES control, an error message is issued and processing is terminated.

The format of the ORDER control is:

```
ORDER(  SEGMENTS(segment[ / class][,...]) | CLASSES(class [(segment [,...])][,...])  )
```

ORDER can be abbreviated **OD**.

SEGMENTS specifies that a list of one or more segments follows. The segment list is enclosed in parentheses and separated with commas. If a non-existent segment name (i.e., not found in the input module) appears in the list, an error message is issued. **SEGMENTS** can be abbreviated **SM**.

CLASSES specifies that a list of one or more classes follows. The class list is enclosed in parentheses and separated with commas. If a non-existent class name appears in the list, an error message is issued. Optionally, the order of the segments within a class can be specified with the CLASSES subcontrol. The segment list is entered immediately following the class name, enclosed in parentheses and separated with commas. The segment list does not have to be complete. That is, a partial list can be entered. When a partial list is entered, the listed segments are located first within the class then any remaining segments of the same class are located following the last specified segment in the class. **CLASSES** can be abbreviated **CS**.

ORDER Control Examples. To specify that segments A1, A5, and A10 be located in that order:

```
ORDER(SEGMENTS(A1,A5,A10))
```

To specify that classes RAM and ROM be located in that order and that the segments RA11, RA34, and RA55 be located in that order at the beginning of the class RAM (segments within ROM will be located in default order):

```
ORDER(CLASSES(RAM(RA11,RA34,RA55),ROM))
```

To specify the following order of segments and classes:

```
segments   — CODE
           — DATA
classes    — ROM (with segments in order R5, R4, and R1)
           — DATA
segments   — STACK
           — DATA1
           — MEMORY

ORDER(SEGMENTS(CODE,DATA),CLASSES(ROM(R5,R4,R1),DATA), &
**SEGMENTS(STACK,DATA1,MEMORY))
```

RESERVE Control

The RESERVE control specifies portions of 8086 memory space that are not to be used in locating the output module. A list of pairs of memory locations are included with the control. The first address of each pair must be lower or equal to the second address or an error message is issued.

If no RESERVE control is entered, the available memory address range is assumed to be 00200H through 0FFFFFH. The memory from 0H through 00200H can be used by specifying an address in this range in an ADDRESSES control.

The format of the RESERVE control is:

RESERVE(*address1* **TO** *address2* [,...])

RESERVE can be abbreviated **RS**.

address1 and *address2* specify a range of memory addresses. The two addresses are separated with the keyword **TO**. If multiple address ranges are specified, the ranges are separated with commas. *address1* must be lower or equal to *address2*.

RESERVE Control Examples. To specify that the addresses from 0200H through 0FFFFH are not to be assigned to any segments:

```
RESERVE(0200H TO 0FFFFH)
```

To reserve the areas 1000H through 2000H and 8000H through 0A000H:

```
RESERVE(1000H TO 2000H,8000H TO 0A000H)
```

SEGSIZE Control

The SEGSIZE control specifies the size of one or more segments in the output module. The SEGSIZE control can be used to add or subtract a value from the size

of a segment or to specify an exact size. If a segment size is changed such that it will no longer hold the code or data, a warning message is issued. Normally, the error condition occurs only when a segment is reduced in size.

The most common usage of the SEGSIZE control is to change the size of a stack segment.

The format of the SEGSIZE control is:

SEGSIZE(*segment*[/ class]([+ | −]*value*)[,...])

SEGSIZE can be abbreviated **SS**.

segment is the name of a segment whose size is to be changed.

class is the name of the class to which a segment belongs, if any.

+ and **−** are optional operators that specify whether the following *value* is to be added to or subtracted from the current size of the segment. If neither the **+** or **−** is entered, the *value* is used as an exact size. That is, *value* becomes the segment size. The value can be in the range 0H through 0FFFFH.

SEGSIZE Control Examples. To specify that the STACK segment is to be 40H bytes:

```
SEGSIZE(STACK(40H))
```

To specify that the STACK segment is to be 18H bytes larger than it is in the input segment:

```
SEGSIZE(STACK(+18H))
```

To specify that the STACK segment of class CODE is to be 40H bytes:

```
SEGSIZE(STACK / CODE (40H)).
```

START Control

The START control specifies the starting address of the output module. If no START control is included in the command, the start address is taken from the input module.

The start address can be specified as an absolute address in the form of paragraph and offset or symbolically as a public symbol within the module. If the latter form is used, the public symbol must be defined in the input module or an error message is issued.

The format of the START control is:

START(*publicsymbol* | *paragraph*,*offset*)

START can be abbreviated **ST**.

publicsymbol is the name of a public symbol defined in the input module. If *publicsymbol* is within a group, the base address of the group is loaded into the CS register and the offset to the public symbol is loaded into the IP register. If *publicsymbol* is not a part of a group, the base address of the segment is loaded into the CS register and the offset to the public symbol is loaded into the IP register.

paragraph is a paragraph number in the range 0H through 0FFFFH specifying paragraphs beginning at addresses 0H through 0FFFF0H. *paragraph* must be followed by an *offset* value. The *offset* can be in the range 0H through 0FFFFH. The *paragraph* value is used to initialize the CS register and the *offset* value is used to initialize the IP register.

START Control Examples. To specify that the address of the public symbol HERE is to be used as the start address in the output module:

```
START(HERE)
```

To specify that the start address of the output module is to be an offset of 80H from paragraph number 20H (the generated address is 280H):

```
START(20H,80H)
```

Diagnostic and Output Module Controls

This category of controls contains seven controls. Two of the controls are used to limit the scope of the other five. The two limiting controls specify whether the included controls apply to the object module only or to the diagnostic output only. The other five controls specify whether comments, line numbers, public symbols, and local symbols are to be included in the output.

The five output controls can be included without the limiting controls. In this case they apply to both the output module and the diagnostic output.

When the information controlled by these controls is output to the print file, it is put into a symbol table. The following is an example of a symbol table with all types of information included:

```
SYMBOL TABLE OF MODULE SHIPREK
READ FROM FILE :F1:NAVAL.SRC
WRITTEN TO FILE :F1:WRECK
```

BASE	OFFSET	TYPE	SYMBOL	BASE	OFFSET	TYPE	SYMBOL
SHIPREK: SYMBOLS AND LINES							
0500H	0084H	LIN	1054	0400H	0032H	PUB	ENTERHERE
0500H	0088H	LIN	1055	STACK	0090H	PUB	CHAR1PTR
0200H	0008H	SYM	STARTHERE	0600H	0102H	PUB	CHARPTR
0200H	0016H	SYM	LOOP1	0600H	0102H	BAS	CHAR
STACK	0004H	SYM	NEWLOOP	STACK	0090H	BAS	CHAR1

A symbol may be BASED on another symbol, or non-BASED. A symbol may also be either a local symbol or a STACK symbol. A local symbol has the address components "base" and "offset". A STACK symbol has address components of "STACK" and "offset", the offset being relative to the register BP. In the symbol table above, the symbols CHAR and CHAR1 are BASED symbols (BASED on the symbols CHARPTR and CHAR1PTR, respectively). In the table there are also two examples of STACK symbols: NEWLOOP, which is a non- BASED STACK symbol, and CHAR1, which is a BASED STACK symbol.

OBJECTCONTROLS Control

The OBJECTCONTROLS control specifies that the controls included with it are to only apply to the output module, not the diagnostic output file. The controls that can be included with the OBJECTCONTROLS control are:

COMMENTS and NOCOMMENTS
LINE and NOLINES
PUBLICS and NOPUBLICS
PURGE and NOPURGE
SYMBOLS and NOSYMBOLS

If any other control is included, an error message is issued.

The format of the OBJECTCONTROLS control is:

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

OBJECTCONTROLS can be abbreviated **OC**.

control is any of the allowed controls.

OBJECTCONTROLS Control Example. To specify that comments are to excluded from the output module and that line numbers are to be included:

```
OBJECTCONTROLS(NOCOMMENTS,LINES)
```

PRINTCONTROLS Control

The PRINTCONTROLS control specifies that the controls included with it are to only apply to the diagnostic output file, not the output module. The controls that can be included with the PRINTCONTROLS control are:

COMMENTS and NOCOMMENTS
LINES and NOLINES
PUBLICS and NOPUBLICS
PURGE and NOPURGE
SYMBOLS and NOSYMBOLS

If any other control is included, an error message is issued.

The format of the PRINTCONTROLS control is:

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

PRINTCONTROLS can be abbreviated **PC**.

control is any of the allowed controls.

PRINTCONTROLS Control Example. To specify that local symbols are to be excluded from the diagnostic output and that line numbers are to be included:

```
PRINTCONTROLS(NOSYMBOLS,LINES)
```

COMMENTS Control

The COMMENTS control specifies whether comments are to be included in the output module. This control can be specified in the OBJECTCONTROLS and PRINTCONTROLS controls. However, it has no effect on the diagnostic output file; comments cannot be included in the diagnostic output file. Thus if COMMENTS is specified outside the OBJECTCONTROLS control, it only affects the output module.

The format of the COMMENTS control is:

```
COMMENTS | NOCOMMENTS
```

COMMENTS specifies that comments are to be included in the output module. This is the default condition if no COMMENTS control is included. **COMMENTS** can be abbreviated **CM**.

NOCOMMENTS specifies that no comments are to be included in the output module. **NOCOMMENTS** can be abbreviated **NOCM**.

COMMENTS Control Examples. To explicitly specify that comments be included in the output module:

```
COMMENTS
```

The same effect can be achieved with the following:

```
OBJECTCONTROLS(COMMENTS)
```

To specify that comments are to be excluded from the output module:

```
NOCOMMENTS
```

LINES Control

The LINES control specifies whether line numbers are to be included in the output module and whether the line numbers and their addresses are to be included in the diagnostic output.

If the LINES control is included in an OBJECTCONTROLS control, it only applies to the output module and to the diagnostic output if included in a PRINTCONTROLS control.

The format of the LINES control is:

```
LINES | NOLINES
```

LINES specifies that line numbers and line number addresses be included in the output module and the diagnostic output. This is the default if no LINES control is included. **LINES** can be abbreviated **LI**.

NOLINES specifies that all line number information be excluded from the output. **NOLINES** can be abbreviated **NOLI**.

LINES Control Examples. To explicitly specify that line numbers and their addresses be included in the output module and line numbers and their addresses in the diagnostic output:

```
LINES
```

To specify that line numbers and their addresses be excluded from the diagnostic output:

```
PRINTCONTROLS(NOLINES)
```

PURGE Control

The PURGE control is a combination control. PURGE is exactly equivalent to coding "NOLINES NOSYMBOLS NOCOMMENTS NOPUBLICS". NOPURGE is exactly equivalent to coding "LINES SYMBOLS COMMENTS PUBLICS".

The format of the PURGE control is:

```
PURGE | NOPURGE
```

PUBLICS Control

The PUBLICS control specifies whether public symbols and their addresses are to be included in the output module and whether the public symbols and their addresses are to be included in the diagnostic output.

If the PUBLICS control is included in an OBJECTCONTROLS control, it only applies to the output module and to the diagnostic output if included in a PRINTCONTROLS control.

The format of the PUBLICS control is:

```
PUBLICS | NOPUBLICS
```

PUBLICS specifies that public symbols and their addresses be included in the output module and the diagnostic output. This is the default if no PUBLICS control is included. **PUBLICS** can be abbreviated **PL**.

NOPUBLICS specifies that all public symbol information be excluded from the output. **NOPUBLICS** can be abbreviated **NOPL**.

PUBLICS Control Examples. To explicitly specify that public symbols and their addresses be included in the output module and the public symbols and their addresses in the diagnostic output:

```
PUBLICS
```

To specify that public symbols and their addresses be excluded from the diagnostic output:

```
PRINTCONTROLS(NOPUBLICS)
```

SYMBOLS Control

The SYMBOLS control specifies whether local symbols and their addresses are to be included in the output module and whether the local symbols and their addresses are to be included in the diagnostic output.

If the SYMBOLS control is included in an OBJECTCONTROLS control, it only applies to the output module and to the diagnostic output if included in a PRINTCONTROLS control.

The format of the SYMBOLS control is:

```
SYMBOLS | NOSYMBOLS
```

SYMBOLS specifies that local symbols and their addresses be included in the output module and the diagnostic output. This is the default if no SYMBOLS control is included. **SYMBOLS** can be abbreviated **SB**.

NOSYMBOLS specifies that all local symbol information be excluded from the output. **NOSYMBOLS** can be abbreviated **NOSB**.

SYMBOLS Control Examples. To explicitly specify that local symbols and their addresses be included in the output module and the local symbols and their addresses in the diagnostic output:

```
SYMBOLS
```

To specify that local symbols and their addresses be excluded from the diagnostic output:

```
PRINTCONTROLS(NOSYMBOLS)
```

CHAPTER 4
HOW LINK86 AND LOC86 HANDLE MODULES

How LINK86 Combines Segments

LINK86 combines segments on the basis of the order in which segments are encountered in the input and on the complete segment name. The complete segment name consists of the segment name and the class name.

The LINK86 output module consists of one or more segments in the order in which unique segment names were encountered in the input modules. When a non-unique complete segment name (a name that was previously read) is encountered, the segment is combined with the previous segment. The only way that you can change the sequence in which LINK86 combines segments is to change the names of the segments or to change the order in which modules are listed in the command input list.

How LOC86 Locates Segments

LOC86 goes through two processes to assign addresses to segments. First, the segments are assigned a sequence according to:

- the ORDER control (if one is included in the command)
- their sequence in the input module
- their class name.

After the segments are put in order, addresses are assigned to them. Any addresses supplied with an ADDRESSES control are assigned. Then LOC86 assigns addresses to any remaining segments without addresses.

When LOC86 assigns addresses, it begins at address 00200H or the first available address above a preceding segment with an address assigned. For example, if no addresses are assigned with the ADDRESSES control, the first segment is assigned address 00200H. If the first segment is assigned by the LOC86 command or with the ADDRESSES control, the next segment is assigned the first available address above the preceding segment that meets the segments alignment attribute (byte, word, paragraph, or page).

LOC86 keeps a list of the segments being located. To examine how LOC86 locates segments, we will look at a simplified version of the list. Before assigning addresses to segments, LOC86 develops the segment list in the following manner:

- Segments specified explicitly and implicitly in the ORDER control are inserted into the list. (An explicit reference is by segment name and an implicit reference is by class name.)
- Segments not mentioned in the ORDER control are added to the end of the list or the end of a class collection if the class already exists in the list. Segments are added in the order in which they are encountered in the input module.

Let's run through an example of this sequence. The input module consists of the following segments and classes in the given order:

SEGMENT NAME	CLASS NAME
A	X1
B	X2
C	X2
D	X1
E	X3
F	X1
G	X2
H	X2
I	X1

The LOC86 command contains the following ORDER control:

```
ORDER(SEGMENTS(D),CLASSES(X2))
```

The ORDER control causes the following segments to be inserted into the list:

D
B
C
G
H

Segment D is inserted because it is explicitly referenced in the ORDER control. Segments B, C, G, and H are inserted because they are all the segments with the class X2.

Next, LOC86 searches the input module for segments that are not yet in the list. The first segment encountered is A which has class X1. Although segment D, already in the list, has class X1, D's class is not known in the list. Therefore, the segment A is added to the bottom of the list:

D
B
C
G
H
A

LOC86 next searches the input module for all other segments, not already in the list, that have class X1. Segment D is ignored because it is already in the list. Segments F, I, which have class X1 are not in the list, so they are added. The list now looks like:

D
B
C
G
H
A
F
I

LOC86 continues to search the input module for segments that are not in the list. This time it finds segment E with class X3. This segment is added to the end of the list. This is the last segment in the input module, so the process is finished. The final list, with classes is:

SEGMENT NAME	CLASS NAME
D	X1
B	X2
C	X2
G	X2
H	X2
A	X1
F	X1
I	X1
E	X3

There is a variation to this ordering algorithm. If one of the segments has the MEMORY attribute it is automatically located at the end of the list no matter where it was encountered in the input module and independent of its class name. It can only be located elsewhere in the list if it is specified in the ORDER control or the ADDRESSES control.

Assigning Addresses

After the segments are put into the correct relative order, addresses must be assigned. Addresses can come from:

- The translator.
- The ADDRESSES control in the LOC86 command.
- The LOC86 command defaults.

The 8086 has address space 0H through 0FFFFFH available for programs. LOC86 will only use address space 200H through 0FFFFFH unless a lower address is specified with the ADDRESSES control.

If there are no translator-generated absolute segments in the input module and no ADDRESSES or RESERVE controls, LOC86 assigns the first segment the address 200H. The address assigned to the second segment depends on the length of the first segment and the alignment attribute of the second segment. For example if the first segment begins at address 200H and is 84H bytes long and the second segment has the paragraph alignment attribute, the second segment will be located at address 290H, the first paragraph boundary following the first segment. If the second segment had the byte alignment attribute, it would be located at address 284H.

Additionally, there must be enough available space for a segment when an address is assigned. If in the last example, the second segment was 100H bytes long but there was a RESERVEd area at address 300H, the segment would be located above the area beginning at 300H.

If addresses are assigned to some segments with the ADDRESSES control and others are to be assigned by LOC86 there must be room for any segments that come between user assigned addresses. If there is a conflict between the determined order and address assignment, error messages are generated.

How to Create Overlays With LINK86 and LOC86

Sometimes your 8086 program is too large to fit into the memory available on the system. Overlays permit programs to be larger than the available memory.

Typically, an overlay is composed of code and data that is executed in one phase of a program's execution, but not used at any other time. Once executed the memory used by this code can be overwritten with code and data used in an other phase. Sections of code that occupy the same part of memory at different times during execution are called overlays.

Part of an overlayed program is always resident in memory; it usually is comprised of the main program module, frequently used routines, and the overlay loader. This part of the program is called the root. Figure 4-1 illustrates the memory configuration of one program that uses overlays.

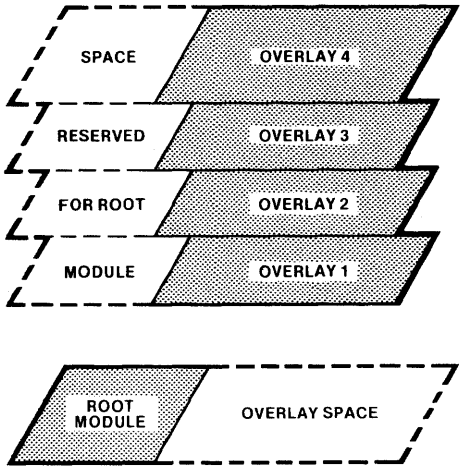


Figure 4-1. Memory Configuration of Program With Overlays 639-7

Before you can create overlays, you must divide your source program into separately translatable modules (they form the root and overlays). If you are using PL/M, you must compile each module with the same program size control.

Once your source code has been divided and translated, there are a maximum 8 steps required to create an executable root and overlays. These steps are listed below:

1. Link root with all necessary library routines, but without overlays. A link map is needed.
2. Link overlays with all necessary library modules. A link map is needed. If overlays only use externals contained in the root, this step may be skipped. The translator's module information may be used.
3. Examine link maps or translator module information to compute overlay placement.
4. Locate root module leaving room for overlays.
5. Link root to overlays with PUBLICSONLY control.
6. Locate overlays in area reserved when root was located.
7. Link overlays to root with PUBLICSONLY control.
8. Locate root in the same area used in step 4.

Annotated Example

To examine how LINK86 and LOC86 are used to create overlays, let us consider the following problem. A PL/M program has been divided into three modules (a root and two overlays). For purpose of efficiency we want all code and all data to fit within 64K bytes (the CS and DS registers need not be changed during execution).

First we must compile all modules with the same program size controls (either COMPACT or SMALL). Examples of the invocation lines are shown below.

```
PLM86 :F1:ROOT.SRC SMALL
PLM86 :F1:OV1.SRC SMALL
PLM86 :F1:OV2.SRC SMALL
```

In the next step we must link the root module. There are unresolved references to code and data in the overlays, but we will need the link map for locating purposes. The link map shows the size of each segment in the root. Since the overlays are self contained except for references to the root, we do not need a link map for them. The PL/M listing files will show the size of each overlay's segments.

```
LINK86 :F1:ROOT.OBJ, USER.LIB MAP
```

Figure 4-2 shows the link map from the root and the Module Information from each overlay's listing file.

```
                              OV1's segment size information
MODULE INFORMATION:
      CODE AREA SIZE      = 7531H  30001D       this is the CODE segment
      CONSTANT AREA SIZE  = 0081H    129D       this is the CONST segment
      VARIABLE AREA SIZE  = 0181H    385D       this is the DATA segment
      MAXIMUM STACK SIZE  = 0040H     64D       this is the STACK segment
      918 LINES READ
      0 PROGRAM ERROR(S)
END OF PL/M-86 COMPILATION
```

```
                              OV2's segment size information
MODULE INFORMATION
      CODE AREA SIZE      = 1B9AH  7066D        this is the CODE segment
      CONSTANT AREA SIZE  = 0101H   257D        this is the CONST segment
      VARIABLE AREA SIZE  = 0454H  1108D        this is the DATA segment
      MAXIMUM STACK SIZE  = 0067H   103D        this is the STACK segment
      918 LINES READ
      0 PROGRAM ERROR(S)
END OF PL/M-86 COMPILATION
```

```
ISIS-II MCS-86 LINKER, V1.2, INVOKED BY:
LINK86 :F1:ROOT.OBJ, USER.LIB MAP
LINK MAP FOR :F1:ROOT.LNK(ROOT)
LOGICAL SEGMENTS INCLUDED:
LENGTH ADDRESS   SEGMENT          CLASS
 8A9BH  ------   CODE             CODE
 0381H  ------   CONST            CONST
 0291H  ------   DATA             DATA
 0030H  ------   STACK            STACK
 0000H  ------   MEMORY           MEMORY
INPUT MODULES INCLUDED:
:F1:ROOT.OBJ(ROOT)
:F0:USER.LIB(LOADER)
:F0:USER.LIB(EXIT)
:F0:USER.LIB(ERROR)
:F0:USER.LIB(TIME)
```

Figure 4-2. Link Map for :F1:ROOT.LNK and Module Information for Overlays

Note that the length of the root's code segment and OV1's code segment just fit within 64K. This means that the code for the overlays must be in a part of memory contiguous with the root (to avoid altering the CS register during execution). OV2's CONST and DATA segments are larger than OV1's so that the STACK segment must be placed to leave room for OV2's CONST and DATA segments. If the overlays share the STACK and MEMORY segments with the root, they must be located at the same address.

After computing the required location for the root's DGROUP and STACK, we can locate the root module. The resulting file will not be executable, but it allows us to resolve references to the root's code and data symbols in the overlays. The following LOC86 invocation will leave room for the overlays' code segments, and place the DGROUP in the first unused memory location. (In the command line below the DS register is initialized to 0FFCH, and the CS register is initialized to 0.) The STACK and MEMORY segments will be located above OV2's DATA segment.

```
LOC86 :F1:ROOT.LNK ADDRESSES(GROUPS(CGROUP(0H), &
                                    DGROUP(0FFCEH)), &
                             SEGMENTS(CODE(0H), &
                                      CONST(0FFCEH), &
                                      STACK(10B34H))) &
                   ORDER(SEGMENTS(CODE, CONST, DATA, STACK, MEMORY)) &
                   SEGSIZE(STACK(100H))
```

Once the root is located, we can use it to resolve external references in the overlay modules. The overlay modules cannot call each other, since only one is resident in memory at any time. The link commands are shown below. The NOPUBLICS with the EXCEPT control is used to avoid conflicts when we use the located overlays to resolve external references in the root.

```
LINK86 :F1:OV1.OBJ, PUBLICSONLY(:F1:ROOT) &
                    NOPUBLICS EXCEPT(OV1CODE, OV1DATA)

LINK86 :F1:OV2.OBJ, PUBLICSONLY(:F1:ROOT) &
                    NOPUBLICS EXCEPT(OV2CODE, OV2DATA)
```

The PUBLICSONLY control resolves references to public symbols contained in the root.

After the overlays have been linked, they must be located. The code and data segments must be placed in the memory locations that were reserved when we first located the root. In this case the STACK and MEMORY segments must be the same for the overlays and the root.

```
LOC86 :F1:OV1.LNK ADDRESSES(GROUPS(CGROUP(0H), &
                                   DGROUP(0FFCEH)), &
                            SEGMENTS(CODE(8A9CH), &
                                     CONST(105E0H), &
                                     STACK(10B34H))) &
                  ORDER(SEGMENTS(CODE, CONST, DATA, STACK, MEMORY)) &
                  SEGSIZE(STACK(100H))

LOC86 :F1:OV2.LNK ADDRESSES(GROUPS(CGROUP(0H), &
                                   DGROUP(0FFCEH)), &
                            SEGMENTS(CODE(8A9CH), &
                                     CONST(105E0H), &
                                     STACK(10B34H))) &
                  ORDER(SEGMENTS(CODE, CONST, DATA, STACK, MEMORY)) &
                  SEGSIZE(STACK(100H))
```

The CGROUP and DGROUP base address must be specified in order to compute offset information. LOC86 will round 0FFCEH to 0FFC0H.

Once the overlays are located, the root is linked and located into an executable form. The PUBLICSONLY control will resolve references to symbols in the overlay modules. Other than the addition of this input control, the LINK86 and LOC86 command must be identical to those used previously.

```
LINK86 :F1:ROOT.OBJ, USER.LIB, PUBLICSONLY(:F1:OV1, :F1:OV2)

LOC86 :F1:ROOT.LNK ADDRESSES(GROUPS(CGROUP(0), &
                                    DGROUP(0F FCEH)), &
                             SEGMENTS(CODE(0), &
                                      CONST(0FFCEH), &
                                      STACK(10B3 4H)) &
                   ORDER(SEGMENTS(CODE, CONST, DATA, STACK, MEMORY)) &
                   SEGSIZE(STACK(100H))
```

The executable forms of the root and its overlay files are contained in :F1:ROOT, :F1:OV1, and :F1:OV2. Figure 4-3 shows the resulting layout of memory.

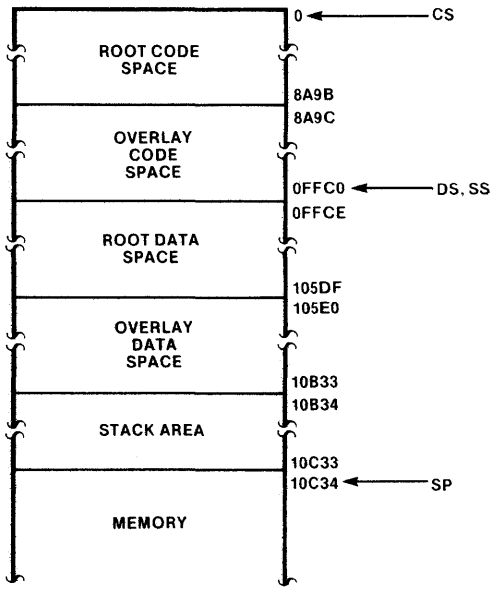


Figure 4-3. Memory Organization for Example 639-9

CHAPTER 5
LIB86 COMMAND

The LIB86 command allows you to create specially formatted files to contain libraries of object modules, to maintain these libraries by adding and deleting modules, and to obtain a listing of the modules and public symbols in a library file. Libraries can be used as input to LINK86 which may automatically link modules from the library that satisfy external references in the modules being linked.

The library manager program is called into operation by the LIB86 command. The format of the LIB86 command is:

```
LIB86
```

LIB86 ignores the command tail, if any.

The operation of LIB86 is controlled by entering commands to indicate which operation LIB86 is to perform. LIB86 prompts for commands with an asterisk (*). The commands are:

COMMAND	ABBREVIATION
CREATE	C
ADD	A
DELETE	D
LIST	L
EXIT	E

Continuation Lines

If a command to LIB86 is longer than one line on your console (which must not be greater than 122 characters), you can continue it by entering an ampersand (&) before the carriage return. The ampersand cannot appear within a filename or control keyword. It can be placed between a keyword and a parameter, for example:

```
ADD PVT.LIB &
**(MOD1)
```

LIB86 prompts for the continued line with a double asterisk (**). If necessary, subsequent lines can be continued also.

CREATE - Create a Library File

The CREATE command creates an empty library file. You may use the ADD command to add modules to the library file. The format of the CREATE command is:

```
CREATE filename
```

where

filename specifies the name to be assigned to the new library file. If a file with that name already exists, an error message is sent to the console and LIB86 prompts for another command.

ADD - Add Modules to a Library File

The ADD command adds object modules to a library file. The format of the ADD command is:

ADD *sourcefile* [(*modname*,...)][,...] **TO** *libfile*

where

sourcefile can be the name of a library file or the name of a file containing one or more object modules. If a library file is specified, all the object modules contained in it are added to *libfile* unless *modnames* are specified.

modname can be specified only if *sourcefile* is a library file. Only the object modules specified by *modnames* are added to *libfile* .

libfile is the library file being modified by the addition of modules in *sourcefile* .

DELETE - Delete Modules from Library File

The DELETE command deletes modules from a library file. The format of the DELETE command is:

DELETE *libfile* (*modname* ,...)

where

modname specifies the object modules to be deleted from *libfile* .

LIST - List Library Modules and Their Public Symbols

The LIST command lists the module directory of library files. The format of the LIST command is:

LIST *libfile* [(*modname*,...)],... [**TO** *listfile*] [**PUBLICS**]

where

libfile is the name of the library file whose module directory is to be listed unless *modname* is also specified. In that case, only information about the specified modules is listed.

listfile is the name of the file to contain the library listing. If omitted, the directory is listed on the current console output device (:CO:).

PUBLICS specifies that public names in each module are to be listed. If omitted, only the module names are listed.

The format of the listing when public names are requested is (note that PUBLICS is abbreviated by P):

```
*LIST TEST.LIB P

TEST.LIB
  OPEN
      NOREX
      ABEX
  REDUCE
      HEX
      OCT
      DATUM
  CLOCK
      TIME
      LAPSE
      CYC
        └──── public names
    └──────── module names
└──────────── library name
```

EXIT - Return to ISIS-II

The EXIT command returns control to ISIS-II. When finished with LIB86, enter the EXIT command, followed by a carriage return. This terminates the LIB86 program and returns control to ISIS-II, which prompts for a command with a hyphen (-).

The format of the EXIT command is:

```
EXIT
```

Example. The following example shows the creation of a library file and the entry in the library of two modules. The directory of the library is listed before exiting to ISIS-II.

```
-LIB86
ISIS-II MCS-86 LIBRARIAN Vx.y
*CREATE FOO.LIB
*ADD SIN.OBJ,COS.OBJ TO FOO.LIB
*LIST FOO.LIB

FOO.LIB
    SINE
    COSINE
*EXIT
```

CHAPTER 6
OH86 COMMAND

The OH86 command converts a 8086 absolute object module to the hexadecimal format. This conversion may be necessary to punch a module to paper tape for later loading by a hexadecimal loader such as the Monitor or UPM. The conversion may be made to put the module in a more readable format that can be displayed or printed.

The module to be converted must be in absolute format; the output from LOC86 is in absolute format. That is, it must not contain any relocatable records. If relocatable records are encountered, the command processing is terminated and control is returned to ISIS-II. If the input file contains symbol table information used for debugging, this information is not converted and does not appear in the output file.

The OH86 command signs on with the message:

```
ISIS-II MCS-86 OBJECT TO HEX FILE CONVERTER, Vx.y
```

Following a correct conversion, processing is terminated with the message:

```
CONVERSION COMPLETE, NO ERRORS
```

If error conditions are encountered, the error messages are issued to the system console at this time. The possible error messages are listed in Appendix A of this manual.

The format of the OH86 command is:

OH86 *inputfile* **TO** *outputfile*

where:

OH86 is the command name.

inputfile is the name of a file containing the 8086 absolute object module to be converted. All restrictions concerning filenames as noted in the *ISIS-II User's Guide* apply to OH86 filenames.

outputfile is the name of the file which is to receive the converted object module.

All filenames default to disk drive :F0: unless a different device is specified.

OH86 Command Example

The following command will convert the file TEST on drive :F0: (the default) to hexadecimal format and store the result in the file TEST.HEX on drive :F1:

```
OH86 TEST TO :F1:TEST.HEX
```

APPENDIX A
ERROR MESSAGES

This appendix contains the error and warning messages for utility commands. The error messages are in numerical order unless they do not contain error numbers, in which case they are in alphabetical order.

LINK86 Error Messages

The LINK86 error messages are numbered. The error messages may be followed by additional lines of information relating to the error.

ERROR 1: I/O ERROR; *ISIS-II error*
FILE: *filename*
ERROR 2: I/O ERROR; *ISIS-II error*
FILE: *filename*
ERROR 3: I/O ERROR; *ISIS-II error*
FILE: *filename*
ERROR 4: CONSOLE I/O ERROR; *ISIS-II error*

An ISIS-II I/O error was detected. The text of the message includes a description of the error similar to the errors listed in the *ISIS-II User's Guide*. *ISIS-II error* is one of the following:

- ILLEGAL FILENAME
- ILLEGAL DEVICE NAME
- ATTEMPT TO WRITE TO INPUT FILE
- ATTEMPT TO READ FROM OUTPUT FILE
- FULL DIRECTORY
- FILE IS ALREADY OPEN
- NO SUCH FILE
- FILE IS WRITE PROTECTED
- ATTEMPT TO SEEK ON NON-DISK FILE
- MISSING FILENAME
- MISSING FILE EXTENSION
- ATTEMPT TO DELETE OPEN FILE

ERROR 1 is issued when the error occurs in an input file, 2 for the output file, 3 for the print file, and 4 for the console file. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 5: INPUT PHASE ERROR
FILE: *filename*
MODULE: *module name*

This error occurs when LINK86 encounters different data during pass two than it read during pass one. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 6: CHECK SUM ERROR
FILE: *filename*
MODULE: *module name*

A bad check sum was detected in the input module. This indicates a bad input module or transmission error. Processing is terminated, all open files are closed, and control is returned to ISIS-II. Rerun the last operation performed on the module (LINK86, compilation, assembly) and then reissue the LINK86 command.

ERROR 7: COMMAND INPUT ERROR

There is an error in the command. This occurs when LINK86 cannot read some portion of the command line or the command line is improperly terminated. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

WARNING 8: SEGMENT COMBINATION ERROR
FILE: *filename*
MODULE: *module name*
SEGMENT: *segment name*
CLASS: *class name*

Two segments with the same complete name have different combining attributes. For example, one may be a memory segment and one may be a stack segment. Processing for error messages continues, but the output module is invalid.

WARNING 9: TYPE MISMATCH
FILE: *filename*
MODULE: *module name*
SYMBOL: *symbol*

The symbol named in the message has more than one definition. The multiple definitions specify different types. The first defined type is used. This is a warning message, so processing continues.

WARNING 10: DIFFERENT VALUES FOR
FILE: *filename*
MODULE: *module name*
SYMBOL: *symbol*

There are multiple values for the symbol named in the message. The first value is used. This is a warning message, so processing continues.

ERROR 11: INSUFFICIENT MEMORY
FILE: *filename*
MODULE: *module name*

The memory available for execution of LINK86 has been used up. This is generally caused by a large number of external and public symbols, or possibly a large number of segments in the input or a fantastically long command tail. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

WARNING 12: UNRESOLVED SYMBOLS

The input contains unresolved symbols. This may be the result of forgetting an input module. This is a warning, so processing continues.

WARNING 13: IMPROPER FIXUP
FILE: *filename*
MODULE: *module name*
SYMBOL: *symbol*

There is an improper fixup in the specified input file. This may be a transmission or translator error. Reissue the command and try again. If the error persists, retranslate the source file and then link again. The error may also be caused by a self relative fixup (e.g., a jump) occuring and a different segment register is assumed for the two locations (the jump-from and jump-to locations). If this is the case, you must fix your source code to make sure that the same segment register is assumed for both locations. This is a warning message, so processing continues.

WARNING 14: GROUP ENLARGED
FILE: *filename*
GROUP: *group name*
MODULE: *module name*

Two or more groups with the same name were encountered in the input. The groups were combined into a single group. This may cause the group to exceed the 64K maximum group size (if so, a message to that effect will be generated by LOC86). If you don't want the groups combined, change the name of groups to be left uncombined. This is a warning message, so processing continues.

ERROR 15: LINK86 ERROR
FILE: *filename*
MODULE: *module name*

There is an apparent problem with the LINK86 command you entered. There might have been a transmission error. Reconstruct the input files and try again. If the error persists, contact your friendly neighborhood Intel service representative. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 16: STACK OVERFLOW
FILE: *filename*
MODULE: *module name*

Type definitions within the input modules are too complex to be handled by the current version of LINK86. Contact your Intel service representative. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

WARNING 17: SEGMENT OVERFLOW
SEGMENT: *segment name*
CLASS: *class name*

The combination of two or more segments has resulted in a segment that exceeds 64K bytes in size. Processing continues, but the output module is invalid.

ERROR 18: GROUP HAS TOO MANY ELEMENTS
FILE: *filename*
GROUP: *group name*
MODULE: *module name*

Internal constraints in LINK86 prohibit a group from having more than twenty internal descriptors. The specified group has exceeded this limitation. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 19: TYPE DESCRIPTION TOO LONG
FILE: *filename*
MODULE: *module name*

Internal constraints limit the length of the description of the type of a symbol. A type has been encountered whose type is too long to be so described. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

WARNING 20: NO SUCH GROUP
NAME: *name*

The name specified in the error message is used as a group name in the RENAMEGROUPS control of the command. No group by this name exists in the input. This may be a typing error or you may have forgotten to include an input file. This is a warning message, so processing continues.

WARNING 21: RENAME ERROR
NAME: *name*

The name specified as a new name in a RENAMEGROUPS control is already in use as the name of another group. The group name will not be changed. Reissue the command and specify a unique group name. This is a warning message, so processing continues.

ERROR 22: INVALID SYNTAX
ERROR IN COMMMAND TAIL NEAR #:
partial command tail

A syntax error was detected in the command. The command tail is repeated up to a point near where the error was detected. A # character indicates the point where the error was detected. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 23: BAD OBJECT FILE
FILE: *filename*
MODULE: *module name*

There is a record in the input that has an incorrect format. This may be the result of an error in the translator or a data transmission error. You may also have specified a non-object file by mistake. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

WARNING 24: CANNOT FIND MODULE
FILE: *filename*
MODULE: *module name*

The module named in the message cannot be found. You may have mistyped the name while entering the command or you may have the wrong disk in the drive. Make sure the file exists and then reissue the command. This is a warning message, so processing continues.

WARNING 25: EXTRA START ADDRESS IGNORED
FILE: *filename*
MODULE: *module name*

More than one main module was included in the input list. Or, no main module was included and more than one non-main module with a start address was included in the input list. The first start addresses are used in the output module, all other start addresses encountered are ignored. This is a warning message, so processing continues.

ERROR 26: NOT AN OBJECT FILE
FILE: *filename*

The file named in the message, judging from its first byte of data, is not a valid object module. This is probably caused by a mistake in entering the name. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 27: NO OVERLAY
FILE: *filename*

The LINK86 overlay file, LINK86.OV0, does not exist on the disk that contained the primary portion of the processor. The overlay may have been accidently erased, or it may not have been copied when a new linker disk was generated. Reissue the command making sure that both LINK86 files (LINK86 and LINK86.OV0) are on the same disk.

WARNING 28: POSSIBLE OVERLAP
FILE: *filename*
MODULE: *module name*
SEGMENT: *segment name*
CLASS: *class name*

Absolute segments have been combined. LINK86 has no way of knowing the size of these segments, therefore, there is the possibility of an overlap. This is a warning message, so processing continues.

WARNING 29: GROUP HAS BAD EXTERNAL REFERENCE
GROUP: *group name*
SEGMENT: *segment name*

If the public symbol corresponds to the external reference has been specified by its absolute address, and does not reside in any segment. This is a warning message, so processing continues.

ERROR 30: LIBRARY IS NOT ALLOWED WITH
PUBLICSONLY CONTROL
FILE: *filename*

The file specified is a library and, as such, is not permitted as an argument to the PUBLICSONLY control. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

WARNING 31: REFERENCED LOCATION OFFSET UNDERFLOW
FILE: *filename*
MODULE: *module name*
SYMBOL: *symbol*

A (8089) self relative reference destination is outside of the segment containing the symbol. This is a warning message, so processing continues.

LOC86 Error Messages

The LOC86 error messages are numbered. The error messages may be followed by additional lines of information relating to the error.

ERROR 1: I/O ERROR; *ISIS-II error*

An ISIS-II I/O error was detected. The text of the message includes a description of the error similar to the errors listed in the *ISIS-II User's Guide*. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 2: INVALID SYNTAX
ERROR IN COMMAND TAIL NEAR #:
partial command tail

A syntax error was detected in the command. The command is repeated up to and including the point of error. A # character immediately follows the point of error. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 3: MISSING INPUT FILE NAME
ERROR IN COMMAND TAIL NEAR #:
partial command tail

The command was issued without supplying an input file name. A # character immediately follows the point of error. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 4: INSUFFICIENT MEMORY

The memory available for execution of LOC86 has been used up. This is generally caused by a large number of segments in the input or an excessively long command tail. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 5: BAD RECORD FORMAT
MODULE: *module name*

There is a record in the specified input module that has an incorrect format. This may be the result of an error in the translator or a data transmission error. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 6: INVALID KEY WORD
ERROR IN COMMAND TAIL NEAR #:
partial command tail

A invalid keyword was found in the command. The command is repeated up to and including the point of error. A # character immediately follows the point of error. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 7: NUMERIC CONSTANT LARGER THAN 20 BITS
ERROR IN COMMAND TAIL NEAR #:
partial command tail

A numeric constant specifying more than 20 bits in binary was found in the command where a 20 bit constant was expected. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 8: NON NUMERIC CHARACTER IN NUMERIC CONSTANT
ERROR IN COMMAND TAIL NEAR #:
partial command tail

A non-numeric character was found in a 16 bit or 20 bit constant. The characters allowed in numeric constants are 0-9, A-F, and B, Q, H, or O as base identifiers. This error is often the result of entering a hexadecimal number with a letter rather than digit first. For example, FFH must be entered as 0FFH. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 9: NUMERIC CONSTANT LARGER THAN 16 BITS
ERROR IN COMMAND TAIL NEAR #:
partial command tail

A numeric constant specifying more than 16 bits in binary was found in the command where a 16 bit constant was expected. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 10: INVALID SEGMENT NAME
ERROR IN COMMAND TAIL NEAR #:
partial command tail

An identifier was found where a segment name was expected. The identifier does not represent a valid segment name. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 11: INVALID CLASS NAME
ERROR IN COMMAND TAIL NEAR #:
partial command tail

An identifier was found where a class name was expected. The identifier does not represent a valid class name. A # character immediately follows the point of error. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 12: INVALID INPUT MODULE
MODULE: *module name*

The specified input module was found to be invalid for one of several reasons: the record order may be incorrect, an invalid field may have been detected within a record, or a required record was found to be missing. The error is probably the result of a translator error or you are trying to locate a non-8086-object module. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

WARNING 13: MORE THAN ONE SEGMENT WITH MEMORY ATTRIBUTE
SEGMENT: *segment name*

LOC86 found more than one segment with the memory attribute. The first segment with the memory attribute is treated as such. Any other segments with the memory attribute are treated as non-memory segments. This is a warning message, so processing continues.

WARNING 14: GROUP DEFINED BY AN EXTERNAL REFERENCE
NAME: *external name*
GROUP: *group name*

A group has been found that is defined by an external symbol. This is a form of an unresolved external reference. This is a warning message, so processing continues.

WARNING 15: PUBLIC SYMBOL NOT ADDRESSABLE
NAME: *public symbol name*

One or more public symbols have been found for which an offset from the specified base cannot be calculated. This can result from having the public symbol more than 64K bytes from the base, i.e., the segment containing the public symbol is not wholly contained within the 64K byte range defined by the group base. This is a warning message, so processing continues.

WARNING 16: LOCAL SYMBOL NOT ADDRESSABLE
NAME: *local symbol name*

One or more local symbols have been found for which an offset from the specified base cannot be calculated. This can result from having the local symbol more than 64K bytes from the base, i.e., the segment containing the local symbol is not wholly contained within the 64K byte range defined by the group base. This is a warning message, so processing continues.

WARNING 17: LINE NUMBER NOT ADDRESSABLE
NAME: *line number*

One or more line numbers have been found for which an offset from the specified base cannot be calculated. This can result from having the line number more than 64K bytes from the base, i.e., the segment containing the line number is not wholly contained within the 64K byte range defined by the group base. This is a warning message, so processing continues.

WARNING 18: SIZE OF GROUP EXCEEDS 64K
GROUP: *group name*

A group has been found whose component segments do not all lie within the 64K byte range defined by the group base address. Addressing error may result from this condition. This is a warning message, so processing continues.

WARNING 19: BOOTSTRAP SPECIFIED FOR MODULE WITHOUT START ADDRESS

The BOOTSTRAP control was specified for an input module that does not have a starting address. The BOOTSTRAP control is ignored. This is a warning message, so processing continues.

ERROR 20: INVALID NAME
NAME: *bad name*

The input filename is not a valid ISIS-II filename. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 21: START ADDRESS DEFINED BY SPECIFIED EXTERNAL NAME
NAME: *external name*

The start address for the module is defined in terms of an undefined external symbol. No start address is calculated and the start address in the module is left in terms of the external name. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 22: SEGMENT SIZE OVERFLOW; OLD SIZE + CHANGE > 64K
SEGMENT: *segment name*

The SEGSIZE control for the specified segment causes its size to exceed 64K bytes. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 23: SEGMENT SIZE UNDERFLOW; OLD SIZE - CHANGE < 0
SEGMENT: *segment name*

The SEGSIZE control for the specified segment causes its size to be less than zero. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 24: INVALID ADDRESS RANGE

The address range specified for the RESERVE control is not valid. The lower address is probably higher than the high address. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 25: PUBLIC SYMBOL NOT FOUND
NAME: *public symbol name*

The public symbol specified in the START control was not found in the input module. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

WARNING 26: DECREASING SIZE OF SEGMENT
SEGMENT: *segment name*

The SEGSIZE control in the command causes the size of the specified segment to be decremented. This may result in the segment content being unaccounted for in memory allocation, thus resulting in conflicts. This is a warning message, so processing continues.

ERROR 27: SPECIFIED SEGMENT IS ABSOLUTE
SEGMENT: *segment name*

An attempt was made to assign an address to an absolute segment. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

WARNING 28: PAGE RESIDENT SEGMENT CROSSES PAGE BOUNDARY
SEGMENT: *segment name*

The size of the specified page resident segment exceeds 256 bytes. The page residence request is ignored. This is a warning message, so processing continues.

WARNING 29: OFFSET FIXUP OVERFLOW
MODULE: *module name*
REFERENCED LOCATION: *20 bit address*
FRAME OF REFERENCE: *20 bit address*

An offset from a group base (FRAME OF REFERENCE) exceeds 64K. This can occur when you explicitly assign the order or addresses of segments, causing the specified module to fall outside of the 64K physical segment of the group base. This is a warning message, so processing continues.

WARNING 30: UNRESOLVED EXTERNAL REFERENCE TO NAME AT SPECIFIED ADDREESS
NAME: *external name*
SEGMENT: *segment name*
ADDRESS: *20 bit address*

The specified unresolved external is referenced near the specified address. This is a warning message, processing continues.

WARNING 31: UNRESOLVED EXTERNAL REFERENCE TO NAME NEAR SPECIFIED ADDRESS
NAME: *external name*
SEGMENT: *segment name*
ADDRESS: *20 bit address*

The unresolved external is referenced near the specified address. This is a warning message, processing continues.

WARNING 32: OVERFLOW OF LOW BYTE FIXUP VALUE
MODULE: *module name*
REFERENCED LOCATION: *20 bit address*
FRAME OF REFERENCE: *20 bit address*

An 8 bit displacement value was calculated whose value exceeds 255. This is a warning message, so processing continues.

ERROR 33: GROUP HAS NO CONSTITUENT SEGMENTS
GROUP: *group name*

There are no segments defined for the specified group. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 34: SPECIFIED CLASS NOT FOUND IN INPUT MODULE
CLASS: *class name*

The specified class was not found in the input module. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 35: SPECIFIED SEGMENT NOT FOUND IN INPUT MODULE
SEGMENT: *segment name*
CLASS: *class name*

The specified segment was not found in the input module. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

WARNING 36: SEGMENTS OVERLAP
SEGMENT: *segment name*
SEGMENT: *segment name*
LOW OVERLAP ADDRESS: *20 bit address*
HIGH OVERLAP ADDRESS: *20 bit address*

The two specified segments overlap for the specified address range. This is a warning message, processing continues.

ERROR 37: INPUT MODULE EXCEEDS 8086 MEMORY
SEGMENT: *segment name*

The memory required by the input module segments exceeds the available 8086 memory. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

WARNING 38: SEGMENT WITH MEMORY ATTRIBUTE NOT PLACED HIGHEST IN MEMORY
SEGMENT: *segment name*

As the result of an ORDER control or an absolute address assignment the memory segment was not located last, i.e., highest in memory. This is a warning because you requested this location in the command or the translation of the source code, processing continues.

ERROR 39: NO MEMORY BELOW SEGMENT FOR SPECIFIED SEGMENT
SEGMENT: *segment name*
SEGMENT: *segment name*

You requested that one segment be located below another. There is not enough memory below the specified segment, thus the ordering cannot be maintained. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

WARNING 40: CAN NOT MAINTAIN SPECIFIED ORDER
SEGMENT: *segment name*

The ordering request cannot be maintained due to lack of available 8086 address space and addresses already assigned to segments. LOC86 attempts to find a location for the specified segment beginning at address 00200H. This is a warning message, processing continues.

ERROR 41: SPECIFIED CLASS OUT OF ORDER
CLASS: *class name*

The ORDER control and the ADDRESSES control are in conflict relative to the specified class. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 42: SPECIFIED SEGMENT OUT OF ORDER
SEGMENT: *segment name*

The ORDER control and the ADDRESSES control are in conflict relative to the specified segment. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 43: ADDRESS FOR CLASS SPECIFIED MORE THAN ONCE
CLASS: *class name*

The specified class was specified more than once in the same ADDRESSES control. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 44: SEGMENT ADDRESS PREVIOUSLY SPECIFIED IN INPUT MODULE OR COMMAND LINE
SEGMENT: *segment name*

The specified segment was specified more than once in the same ADDRESSES control, or the specified segment has an absolute address in the input module. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 45: SEGMENT SPECIFIED MORE THAN ONCE IN ORDER
SEGMENT: *segment name*

The specified segment was specified more than once in the same ORDER control. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 46: CLASS SPECIFIED MORE THAN ONCE IN ORDER
CLASS: *class name*

The specified class was specified more than once in the same ORDER control. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 47: SPECIFIED SEGMENT NOT IN SPECIFIED CLASS
SEGMENT: *segment name*
CLASS: *class name*

The specified segment is not in the specified class. This is an error in the use of the ORDER control. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 48: INVALID COMMAND LINE

There is an error in the command, reenter the command correctly. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

WARNING 49: SEGMENT ALIGNMENT NOT COMPATIBLE WITH ASSIGNED ADDRESS
SEGMENT: *segment name*

The alignment of the specified segment is not compatible with the address specified in the ADDRESSES control. An address that meets the alignment criteria is assigned and the specified address is ignored. Reexecute the command with correct values in the ADDRESSES control if you want something different than that provided by the command. Note that there is no way to override the alignment attribute of the segment. This is a warning message, processing continues.

ERROR 50: INVALID COMMAND LINE; TOKEN TOO LONG
ERROR IN COMMAND TAIL NEAR #:
partial command tail

The command line contains a token that is too long. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

WARNING 51: REFERENCING LOCATION IS OUTSIDE 64K FRAME OF REFERENCE
MODULE: *module name*
ADDRESS: *20 bit address*
FRAME OF REFERENCE: *20 bit address*

The address of a jump or call lies outside of the 64K frame of reference for a self relative instruction. This warning occurs when locating the module *containing the self relative reference*. This is a warning message, so processing continues.

WARNING 52: REFERENCED LOCATION IS OUTSIDE 64K FRAME OF REFERENCE
MODULE: *module name*
REFERENCED LOCATION: *20 bit address*
FRAME OF REFERENCE: *20 bit address*

The object being addressed by a self relative jump or call is outside the 64K frame of reference for the self relative instruction. This warning occurs when locating the module *containing the self relative reference*. This is a warning message, so processing continues.

WARNING 53: CAN NOT ALLOCATE CLASS AT SPECIFIED ADDRESS
ADDRESS: *20 bit address*
CLASS: *class name*

The specified class can not be located at the specified address because a conflict would result. The class is located at the address nearest the specified address possible without causing a conflict. If you want conflicts, reissue the command specifying addresses for each segment with the ADDRESSES control. This is a warning message, processing continues.

ERROR 54: DATA ADDRESS OUTSIDE SEGMENT BOUNDARIES
SEGMENT: *segment name*
MODULE: *module name*

A data record for the specified segment contains data for addresses outside the segment boundary. The specified module name is the last module name encountered before the error. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

WARNING 55: UNDEFINABLE SYMBOL ADDRESS
SEGMENT: *segment name*
GROUP: *group name*

One or more line numbers, local symbols, or public symbols have been found that are addressed relative to the base of the specified group, however, the specified segment that contains the symbols(s) is not contained within the 64K byte address range. This is a warning message, processing continues.

WARNING 56: SEGMENT IN RESERVE SPACE
SEGMENT: *segment name*

A segment is located at an area reserved by the RESERVE control. This may happen if there is an absolute segment in the input module. This is a warning message, so processing continues.

ERROR 57: INVALID GROUP NAME
ERROR IN COMMAND TAIL NEAR #:
partial command tail

An identifier was found where a group name was expected. The identifier does not represent a valid group name. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 58: SPECIFIED GROUP NOT FOUND IN INPUT MODULE
GROUP: *group name*

The specified group was not found in the input module. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

ERROR 59: GROUP ADDRESS PREVIOUSLY SPECIFIED IN INPUT MODULE OR COMMAND LINE
GROUP: *group name*

The specified group was specified more than once in the same ADDRESSES control, or the specified group has an absolute address in the input module. Processing is terminated, all open files are closed, and control is returned to ISIS-II.

WARNING 60: REFERENCED LOCATION IS NOT WITHIN 32K OF SPECIFIED ADDRESS
MODULE: *module name*
ADDRESS: *20 bit address*
REFERENCED LOCATION: *20 bit address*

The object being referenced by a (8089) self relative reference is not within 32K of that reference. This is a warning message, so processing continues.

ERROR 61: NO OVERLAY FILE
FILE: *filename*

The LOC86 overlay file, LOC86.OV0, does not exist on the disk that contains the primary portion of the program. The overlay may have been accidently erased, or it may not have been copied.

When a new locator disk has been generated, reissue the same command making sure that both LOC86 and LOC86.OV0 files are on the same disk.

LIB86 Error Messages

All LIB86 command error messages are nonfatal because LIB86 is an interactive program. The command (ADD, CREATE, DELETE, EXIT, or LIST) that caused the error is aborted. The errors that are caused by improper command entry are followed by a partial image of the command with a cross hatch (#) in the vicinity of the error.

INSUFFICIENT MEMORY

There is not enough memory available for execution of the command.

INVALID MODULE NAME
partial command tail

A module name in the command is invalid. The name can be from 1 through 40 characters in length and must be composed of the letters A-Z, digits 0-9, a question mark (?), underscore (_), period (.), colon (:) or a commercial at (@) sign.

INVALID SYNTAX
partial command tail

There is an error in the command. Check for 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 <CR>.
- LIST: TO *filename* not followed by PUBLICS or <CR>.

filename FILE ALREADY EXISTS

The file specified in a CREATE command already exists.

filename , BAD RECORD SEQUENCE

The file specified in the command has an unexpected record sequence. It may not be terminated with an EOF record. You may have attempted to ADD a non-object or non-library file to a library.

filename , CHECKSUM ERROR

The specified file contains a record that has an invalid checksum. Go back and generate the file again.

filename , DUPLICATE SYMBOL IN INPUT

You have attempted to ADD modules that contain more than one definition for the same PUBLIC symbol.

filename , ILLEGAL RECORD FORMAT

The file specified in the command has an illegal format. The object file may contain a name that has more than 40 characters. The file may contain records in an improper order.

filename(modname) : NOT FOUND

You have attempted to delete, add, or list a non-existent module. You may have misspelled the name.

filename , NOT LIBRARY

The specified file is not a library.

filename , OBJECT RECORD TOO SHORT

The specified file contains a record of insufficient length.

filename , PREMATURE EOF

The EOF record occurred before the length of the file indicated it should.

LEFT PARENTHESIS EXPECTED
partial command tail

There is a missing left parenthesis "(" in the command.

modname- ATTEMPT TO ADD DUPLICATE MODULE

The specified module already exists in the library.

MODULE NAME TOO LONG
partial command tail

The specified module name exceeds 40 characters.

RIGHT PARENTHESIS EXPECTED
partial command tail

There is a missing right parenthesis ")" in the command.

symbol- PUBLIC SYMBOL ALREADY IN LIBRARY

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

'TO' EXPECTED
partial command tail

The TO file is not specified in the ADD command.

UNRECOGNIZED COMMAND

An illegal or misspelled command (i.e., not ADD, CREATE, DELETE, EXIT, or LIST) was entered.

OH86 Error Messages

All OH86 error messages are fatal. Control is returned to ISIS-II when an error is encountered.

See the *ISIS-II User's Guide* for information on ISIS-II errors that may be generated by OH86 execution.

filename, ILLEGAL RELO RECORD

OH86 encountered a relocatable type record in the input file. Process the input module through LOC86, creating an absolute input file, and then proceed with the conversion.

filename, PREMATURE EOF

No end-of-file record was found after the entire file was read. The file may be damaged or may have been processed incorrectly. Try to generate the absolute module again with the language translator, LINK86 and LOC86.

ILLEGAL INPUT FILE

All information required in an absolute input file is not present. Either the input file is defective or you specified the wrong file in the command. If the file is defective, generate a new absolute file starting from the source code. If you specified the wrong file, reenter the command with the correct filename.

INSUFFICIENT MEMORY FOR DATA RECORD TYPE CONTAINED IN FILE

This error indicates that an iterated data record (generated from a source language construct that produces repetitions in the object file) whose length will not fit in memory was encountered in the input. Use a system with more memory, if possible. Otherwise, go back to the source code and change the program so that each iterated data construct requires fewer repetitions.

APPENDIX B
HEXADECIMAL-DECIMAL CONVERSION

The following table 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 from the table from the starting number. Find the difference in the table. Continue this process until there is no difference.

BYTE				BYTE				BYTE			
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

APPENDIX C
PL/M-86 MODELS OF SEGMENTATION

The segments, classes, and groups in the PL/M-86 compiler output module vary according to the size control specified to the compiler. The segment, class, and group names generated by the PL/M-86 compiler for the SMALL, COMPACT, MEDIUM, and LARGE models are shown below.

Small Model

Segment Name	Class Name	Group Name
CODE	CODE	CGROUP
CONST	CONST	DGROUP
DATA	DATA	
STACK	STACK	
MEMORY	MEMORY	

Compact Model

Segment Name	Class Name	Group Name
CODE	CODE	CGROUP
CONST	CONST	DGROUP
DATA	DATA	
STACK	STACK	none
MEMORY	MEMORY	none

Medium Model

Segment Name	Class Name	Group Name
*modname*__CODE	CODE	none
CONST	CONST	DGROUP
DATA	DATA	
STACK	STACK	
MEMORY	MEMORY	

Large Model

Segment Name	Class Name	Group Name
*modname*__CODE	CODE	none
*modname*__DATA	DATA	none
STACK	STACK	none
MEMORY	MEMORY	none

APPENDIX D
MCS-86™ ABSOLUTE OBJECT FILE FORMATS

Introduction

The 8086 Absolute Object File Format herein described is a proper subset of the full 8086 Object File Formats. An absolute object file consists of a sequence of records defining a single absolute module. An absolute module is defined as a collection of absolute object information which is specified by a sequence of object records.

Definitions

This section defines certain terms fundamental to 8086 Relocation and Linkage (R&L). The terms are ordered not alphabetically, but so you can read forward without forward references.

Definition of Terms

OMF—acronym for Object Module Formats.

R&L—acronym for Relocation and Linkage.

MAS—acronym for Memory Address Space. The 8086 MAS is one megabyte (1,048,576 bytes). Note that the MAS should be distinguished from actual memory, which may occupy only a portion of the MAS.

MODULE—an "inseparable" collection of object code and other information produced by a translator or by the LINK86 program. When a distinction must be made:

T-MODULE—will denote a module created by a translator, such as PL/M-86 or ASM86, and

L-MODULE—will denote a module created by LINK86 from one or more constituent modules. (Note that modules are not "created" in this sense by the MCS-86 Locater, LOC86; the output module from LOC86 is merely a transformation of the input module).

Two observations about modules must be made:

1. Every module must have a name, so that the MCS-86 Librarian, LIB86, has a handle for the module for display to the user. (If there is no need to provide a handle for LIB86, the name may be null). Translators will provide names for T-modules, providing a default name (possibly the file name or a null name) if neither source code nor user specifies otherwise.
2. Every T-module in a collection of modules linked together may have a different name, so that symbolic debugging systems can distinguish the various symbols. This restriction is not required by R&L, and is not enforced by it.

FRAME—a contiguous region of 64K of MAS, beginning on a paragraph boundary (i.e., on a multiple of 16 bytes). This concept is useful because the content of the four 8086 segment registers define four (possibly overlapping) FRAME's; no 16-bit address in the 8086 code can access a memory location outside of the current four FRAME's. The FRAME starting at address 0000H is FRAME 0.

Module Identification

In order to determine that a file contains an object program, a module header record will always be the first record in a module. There are two kinds of header records and each provides a module name. The additional functions of the header records are explained below.

A module name may be generated during one of two processes: translation or linking. A module that results from translation is called a T-MODULE. A T-MODULE will have a T-MODULE HEADER RECORD (THEADR). A name may be provided in the THEADR record by a translator. This name is then used to identify the progenitor of all debug information found in the T-MODULE. The name may be null, i.e., of length zero.

A module that results from linking and locating is called an L-MODULE. An L-MODULE will always have an L-MODULE HEADER RECORD (LHEADR). In the LHEADR record a name is also provided. This name is available for use as a means of referring to the module without using any of its constituent T-MODULE names. An example would be two T-MODULES, A and B, linked together to form L-MODULE C. L-MODULE C will contain two THEADR records and will begin with an LHEADR record with the name C provided by the linker as a directive from the user. The L-MODULE C can be referred to by other tools such as the library manager without having to know about the originating module's names, yet the originating module's names are preserved for debugging purposes.

Module Attributes

In addition to a name, a module may have the attribute of being a main program as well as having a specified starting address.

If a module is not a main module yet has a starting address, then this value has been provided by a translator, possibly for debugging purposes. A starting address specified for a non-main module could be the entry point of a procedure, which may be loaded and initiated independent of a main program.

Physical Segment Definition

A module is defined as a collection of data bytes defined by a sequence of records produced by a translator. The data bytes represent contiguous regions of memory whose contents are determined at translation time.

Physical Segment Addressability

The 8086 addressing mechanism provides segment base registers from which a 64K byte region of memory, called a Frame, may be addressed. there is one code segment base register (CS), two data segment base registers (DS, ES), and one stack segment base register (SS).

Data

The data that defines the memory image represented by a module is maintained in two varieties of DATA records: PHYSICAL ENUMERATED DATA RECORD (PEDATA) and PHYSICAL ITERATED DATA RECORD (PIDATA). Both records specify the data to be loaded into a contiguous section of memory. The start address of this contiguous section is given in the record. PEDATA records contain an exact byte-by-byte copy of the desired memory image. The PIDATA record differs in that the data bytes are represented within a structure that must be expanded by the loader. The purpose of the PIDATA record is to reduce module size by encoding repeated data rather than explicitly enumerating each byte, as the PEDATA record does.

Record Syntax

The following syntax shows the valid orderings of records to form an absolute module. In addition, the given semantic rules provide information about how to interpret the record sequence. The syntactic description language used herein is defined in Wirth: CACM, November 1977, V20, N 11, pg. 822-823.

```
absolute_object_file = module.
module               = tmod | lmod.
tmod                 = THEADR {content_def} MODEND.
lmod                 = LHEADR {t_component} MODEND.
t_component          = [THEADR] {content_def}
content_def          = PEDATA | PIDATA.
```

NOTE

The character strings represented by capital letters above are not literals but are identifiers that are further defined in the section defining the Record Formats.

A proper Absolute Object File produced by Intel products will contain at least the above record types. It may also contain other record types which, if present, will follow the Module Header record and precede the Module End record. These other record types fall into two categories:

1. extraneous—These contain information which is not pertinent to an absolute loader. The record numbers in this category are:

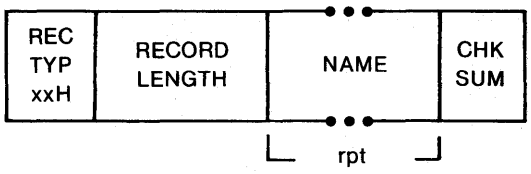
   7AH, 7CH, 7EH, 88H, 8CH, 8EH, 90H, 92H, 94H, 96H, 98H, 9AH, 9CH

2. erroneous—These contain information with regard to relocation, indicating that the object module is not yet in absolute form or are erroneous record types. The record numbers in this category are all other record type numbers.

Record Formats

The following pages present diagrams of Record Formats in schematic form. Here is a sample, to illustrate the various conventions:

Sample Record Format (SAMREC)



Title and Official Abbreviation

At the top is the name of the Record Format described, together with an official abbreviation. To promote uniformity among various programs, the abbreviation should be used in both code and documentation. The abbreviation is always six letters.

The Boxes

Each format is drawn with boxes of two sizes. The narrow boxes represent single bytes. The wide boxes represent two bytes (or one word) each. In the object file, the low order byte of a word value comes first. The wide boxes, with three dots in the top and bottom, represent a variable number of bytes, one or more, depending upon content.

Rec Typ

The first byte in each record contains a value between 0 and 255, indicating the type of record.

Record Length

The second field in each record contains the number of bytes in the record, exclusive of the first 2 fields.

Name

Any field that indicates a "NAME" has the following internal structure: the first byte contains a number between 0 and 40, inclusive, that indicates the number of remaining bytes in the field. The remaining bytes are interpreted as a byte string; each byte must represent the ASCII code of a character drawn from this set:

[?@ :._0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ].

Most translators will choose to constrain the character set more strictly; the above set has been chosen to "cover" that required by all current processors.

Repeated Fields

Some portions of a Record Format contain a field or series of fields that may occur an indefinite number of times (zero or more). Such fields are indicated by the "repeated" or "rpt" brackets below the boxes.

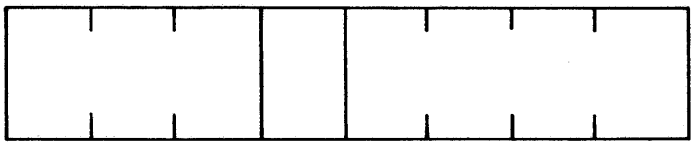
Similarly, some portions of the Record Format are present only if some given condition obtains; these fields are indicated by similar "conditional" brackets below the boxes.

Chk Sum

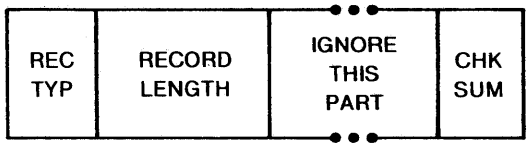
The last field in each record is a check sum, which contains the two's complement of the sum (modulo 256) of all other bytes in the record. Therefore, the sum (modulo 256) of all bytes in the record equals 0.

Bit Fields

Descriptions of contents of fields will sometimes get down to the bit level. Boxes with vertical lines drawn through them represent bytes or words; the vertical lines indicate bit boundaries; thus this byte has three bit-fields of three, one, and four bits:

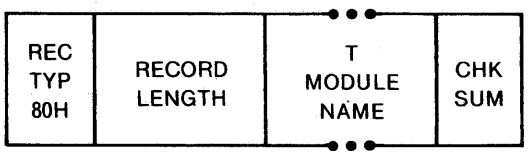


Ignored Records



All record types that may be in an object module that provide information not pertinent to an absolute loader must be ignored. They may all be treated as if they have the above format. Records in this category have REC TYP in the set 7AH, 7CH, 7EH, 88H, 8CH, 8EH, 90H, 92H, 94H, 96H, 98H, 9AH, 9CH.

T-Module Header Record (THEADR)



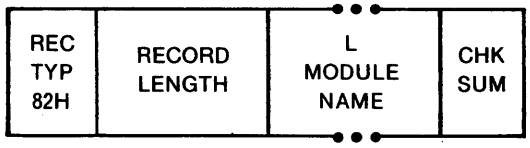
Every module output from a translator must have a T-MODULE HEADER RECORD. Its purpose is to provide the identity of the original defining module for all debug information encountered in the module up to the following T-MODULE HEADER RECORD or MODULE END RECORD.

This record can also serve as the header for a module, i.e., it can be the first record, and will be for modules output from translators.

T-Module Name

The T-MODULE NAME provides a name for the T-MODULE.

L-Module Header Record (LHEADR)

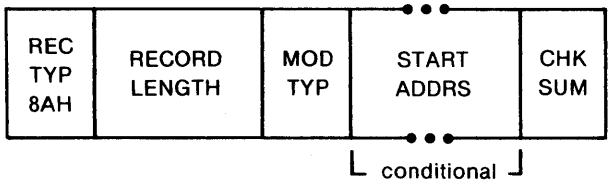


Every module created by LINK86 and LOC86 will have an L-MODULE HEADER RECORD. This record serves only to identify a module that has been processed (output) by the MCS-86 LINKER and/or the MCS-86 LOCATER. When several modules are linked to form another module, the new module requires a name, perhaps unique from those of the linked modules, by which it can be referred to (by the LIB86 program, for example).

L-Module Name

The L-MODULE NAME provides a name for the L-Module.

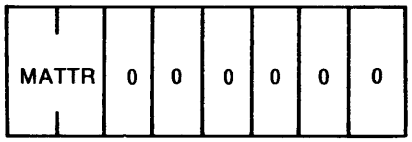
Module End Record (MODEND)



This record serves two purposes. it denotes the end of a module and indicates whether the module just terminated has a specified entry point for initiation of execution. If the latter is true, then the execution address is specified.

Mod Typ

This field specifies the attributes of the module. The bit allocation and their associated meanings are as follows:

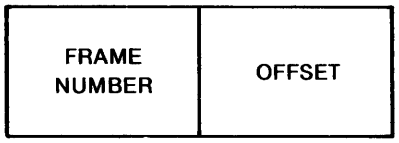


MATTR is a two-bit subfield that specifies the following module attributes:

MATTR	MODULE ATTRIBUTE
0	Non-main module with no starting address
1	Non-main module with starting address
2	(invalid value for MATTR)
3	Main module with starting address

Start Addrs

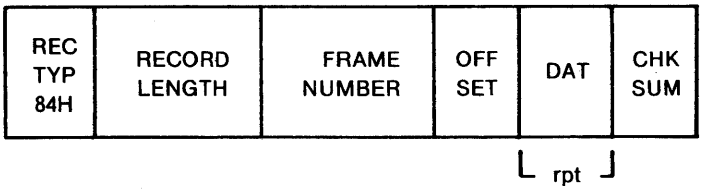
The START ADRS field has the following format:



FRAME NUMBER. This field specifies a frame number relative to which the module will begin execution. This value is appropriate for insertion into the CS register for program initiation.

OFFSET. This field specifies an offset relative to the FRAME NUMBER which defines the exact location of the first byte at which to begin execution. This value is appropriate for insertion into the IP register for program initiation.

Physical Enumerated Data Record (PEDATA)



This record provides contiguous data, from which a portion of an 8086 memory image may be constructed.

Frame Number

This field specifies a Frame Number relative to which the data bytes will be loaded.

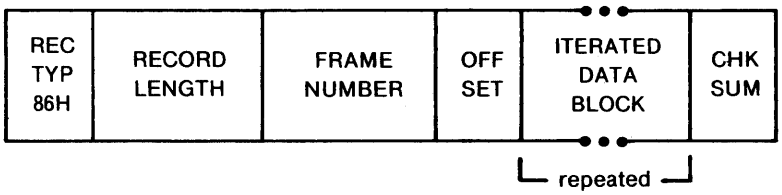
Offset

This field specifies an offset relative to the FRAME NUMBER which defines the location of the first data byte of the DAT field. Successive data bytes in the DAT field occupy successively higher locations of memory. The value of OFFSET is constrained to be in the range 0 to 15 inclusive. If an OFFSET value greater than 15 is desired, then an adjustment of the FRAME NUMBER should be done.

Dat

This field provides consecutive bytes of an 8086 memory image. The number of DAT bytes is constrained only by the RECORD LENGTH field. The address of each byte must be within the frame specified by FRAME NUMBER.

Physical Iterated Data Record (PIDATA)



This record provides contiguous data, from which a portion of an 8086 memory image may be constructed. It allows initialization of data segments and provides a mechanism to reduce the size of object modules when there are repeated data to be used to initialize a memory image.

Frame Number

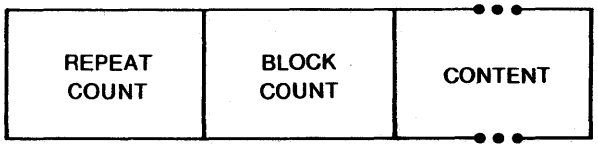
This field specifies a frame number relative to which the data bytes will be loaded.

Offset

This field specifies an offset relative to the FRAME NUMBER which defines the location of the first data byte in the ITERATED DATA BLOCK. Successive data bytes in the ITERATED DATA BLOCK occupy successively higher locations of memory. The range of OFFSET is constrained to be between 0 and 15 inclusive. If a value larger than 15 is desired for OFFSET, then an adjustment of FRAME NUMBER should be done.

Iterated Data Block

This repeated field is a structure specifying the repeated data bytes. It is a structure that has the following format:



Repeat Count. This field specifies the number of times that the CONTENT portion of this ITERATED DATA BLOCK is to be repeated, and must be greater than zero.

Block Count. This field specifies the number of ITERATED DATA BLOCKS that are to be found in the CONTENT portion of this ITERATED DATA BLOCK. If this field has value zero then the CONTENT portion of this ITERATED DATA BLOCK is interpreted as data bytes.

If BLOCK COUNT is non-zero then the CONTENT portion of this ITERATED DATA BLOCK is interpreted as that number of ITERATED DATA BLOCKS.

Content. This field may be interpreted in one of two ways, depending on the value of the previous BLOCK COUNT field.

If BLOCK COUNT is zero, then this field is a one-byte count followed by the indicated number of data bytes.

If BLOCK COUNT is non-zero, then this field is interpreted as the first byte of another ITERATED DATA BLOCK.

NOTE

From the outermost level, the number of nested ITERATED DATA BLOCKS is limited to 17, i.e., the number of levels of recursion is limited to 17.

The address of each data byte must be within the frame specified by FRAME NUMBER.

Hexadecimal Object File Format

Hexadecimal object file format is a way of representing an object file in ASCII.

The function of the utility program, 0H86, is to convert 8086 absolute object modules to 8086 hexadecimal object modules.

The hexadecimal representation of binary is coded in ASCII. For example, the eight-bit binary value 0011 1111 is 3F in hexadecimal. To code this ASCII, one eight-bit byte containing the ASCII code for 3(00110011, or 33H) and one eight-bit byte containing the ASCII code for F(0100 0110, or 46H) are required. This representation (ASCII hexadecimal) requires twice as many bytes as the binary.

There are four different types of records that may make up an 8086 hexadecimal object file. They are:

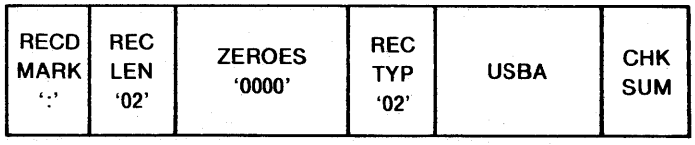
- Extended Address Record
- Start Address Record
- Data Record
- End of File Record

Each record begins with a RECORD MARK field containing 3AH, the ASCII code for colon (:).

Each record has a REC LEN field which specifies the number of bytes of information or data which follows the RECTYP field of each record. Note that one byte is represented by two ASCII characters.

Each record ends with a CHECKSUM field that contains the ASCII hexadecimal representation of the two's complement of the eight-bit sum of the eight-bit bytes that result from converting each pair of ASCII hexadecimal digits to one byte of binary, from and including the RECORD LENGTH field to and including the last byte of the DATA field. Therefore, the sum of all the ASCII pairs in a record after converting to binary, from the RECORD LENGTH field to and including the CHECKSUM field, is zero.

Extended Address Record



The 8086 EXTENDED ADDRESS RECORD is used to specify bits 4-19 of the Segment Base Address (SBA) where bits 0-3 of the SBA are zero. Bits 4-19 of the SBA are referred to as the Upper Segment Base Address (USBA). The absolute memory address of a content byte in a subsequent DATA RECORD is obtained by adding the SBA to an offset calculated by adding the Load Address Field of the containing DATA RECORD to the index of the byte in the DATA RECORD (0, 1, 2, ... n). The offset addition is done modulo 64K, ignoring a carry, so that offset wrap-around loading (from 0FFFFH to 00000H) results in wrapping around from the end to the beginning of the 64K segment defined by the SBA. The address at which a particular data byte is loaded is calculated as:

SBA + ([DRLA + DRI] MOD 64K)

where

DRLA is the DATA RECORD LOAD ADDRESS.

DRI is the data byte index within a DATA RECORD.

When an EXTENDED ADDRESS RECORD defines the value of SBA, the EXTENDED ADDRESS RECORD may appear anywhere within an 8086 hexadecimal object file. This value remains in effect until another EXTENDED ADDRESS RECORD is encountered. The SBA defaults to zero until an EXTENDED ADDRESS RECORD is encountered.

Recd Mark

The RECD MARK field contains 03AH, the hex encoding of ASCII ':'.

Rec Len

The Record Length field contains 3032H, the hex encoding of ASCII '02'.

Zeroes

The Load Address field contains 30303030H, the hex encoding of ASCII '0000'.

Rec Typ

The Record Type field contains 3032H, the hex encoding of ASCII '02'.

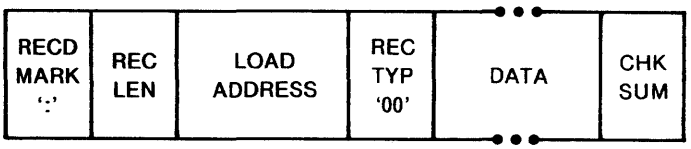
USBA

The USBA field contains four ASCII hexacecimal digits that specify the 8086 USBA value. The high-order digit is the 10th character of the record. The low order digit is the 13th character of the record.

Chk Sum

This is the check sum on the REC LEN, ZEROES, REC TYP, and USBA fields.

Data Record



The DATA RECORD provides a set of hexadecimal digits that represent the ASCII code for data bytes that make up a portion of an 8086 memory image. The method for calculating the absolute address for each byte of DATA is described in the discussion of the Extended Address Record.

Recd Mark

The RECD MARK field contains 03AH, the hex encoding of ASCII ':'.

Rec Len

The REC LEN field contains two ASCII hexadecimal digits representing the number of data bytes in the record. The high-order digit comes first. The maximum value is 'FF' or 4646H (255 decimal).

Load Address

The LOAD ADDRESS field contains four ASCII hexadecimal digits representing the offset from the SBA (see EXTENDED ADDRESS RECORD) defining the address at which byte 0 of the DATA is to be placed. The LOAD ADDRESS value is used in calculation of the address of all DATA bytes.

Rec Typ

The REC TYP field in a DATA record contains 3030H, the hex encoding of ASCII '00'.

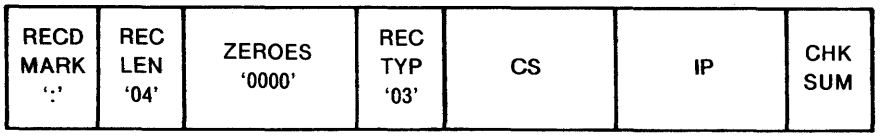
Data

The DATA field contains a pair of hexadecimal digits that represent the ASCII code for each data byte. The high order digit is the first digit of each pair.

Chk Sum

This is the check sum on the REC LEN, LOAD ADDRESS, REC TYPE, and DATA fields.

Start Address Record



The START ADDRESS RECORD is used to specify the execution start address for the object file. Values are given for both the Instruction Pointer (IP) and Code Segment (CS) registers. This record can appear anywhere in a hexadecimal object file.

If a START ADDRESS RECORD is not present in an 8086 hexadecimal file, a loader is free to assign a default start address.

Recd Mark

The RECD MARK field contains 03AH, the hex encoding for ASCII ':'.

Rec Len

The REC LEN field contains 3034H, the hex encoding for ASCII '04'.

Zeroes

The ZEROES field contains 30303030H, the hex encoding for ASCII '0000'.

Rec Typ

The REC TYP field contains 3033H, the hex encoding for ASCII '03'.

CS

The CS field contains four ASCII hexadecimal digits that specify the 8086 CS value. The high-order digit is the 10th character of the record; the low-order digit is the 13th character of the record.

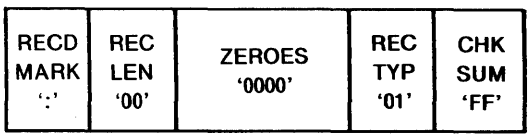
IP

The IP field contains the four ASCII hexadecimal digits that specify the 8086 IP value. The high-order digit is the 14th character of the record, the low order digit is the 17th character of the record.

Chk Sum

This is the check sum on the REC LEN, ZEROES, REC TYP, CS, and IP fields.

End of File Record



The END OF FILE RECORD specifies the end of the hexadecimal object file.

Recd Mark

The RECD MARK field contains 03AH, the ASCII code for colon (:).

Rec Len

The REC LEN field contains two ASCII zeroes (3030H).

Zeroes

The ZEROES field contains four ASCII zeroes (30303030H).

Rec Typ

The REC TYP field contains 3031H, the ASCII code for 01H.

Chk Sum

The CHK SUM field contains 4646H, the ASCII code for FFH, which is the check sum on the REC LEN, ZEROES and REC TYP fields.

Examples

A. Sample Absolute Object File

The following is an example of an absolute object file. The file contains eight records. The eight records perform the following functions:

Record	Function
1	LHEADR record, begins the object module and defines the module name.
2	THEADR record, defines the translator-generated module name which is the same as the name in the LHEADR record.
3	PEDATA record that defines a contiguous memory image from 00200H to 00215H.
4	PEDATA record that defines a contiguous memory image from 00360H to 00377H.
5	PEDATA record that defines a contiguous memory image from 00415H to 0042BH.
6	PEDATA record that defines a contiguous memory image from 051620H to 051633H.
7	PIDATA record that defines a contiguous memory image from 051B00H to 051B1DH. The iterated data consists of three repetitions of "ABC" (414243H), followed by three repetitions of (four repetitions of "D" (44H)), three repetitions of "E" (45H).
8	MODEND record that specifies that the module should be started with CS = 5162H and IP = 0005H.

```
(1) 82 0008 0653414D504C45 AE
(2) 80 0008 0653414D504C45 B0
(3) 84 001A 0020 00
    004992DB246DB6FF4891DA236CB5FE47
    90D9226BB4FD 63
```

(4) 84 001C 0036 00
0062C42688EA4CAE1072D43698FA5CBE
2082E446A80A6CCE 82
(5) 84 001B 0041 05
001D3A577491AECBE805223F5C7996B3
D0ED0A2744617E 72
(6) 84 0018 5162 00
00850A8F14991EA328AD32B73CC146CB
50D55ADF FB
(7) 86 001C 51B0 00
0003 0000 03 414243
0003 0002
0004 0000 01 44
0003 0000 01 45 FA
(8) 8A 0006 C0 5162 0005 F8

NOTE

The blank characters and carriage return and line feed characters are inserted here to improve readability. They do not occur in an object module. This file has been converted to ASCII hex so that it may be printed here. All word values (RECORD LENGTH, REPEAT COUNT, etc.) have been byte-reversed to improve readability.

B. Sample Absolute Hexadecimal Object File

The following is the hexadecimal object file representation of the object file given in Example A.

```
:020000020020DC
:10000000004992DB246DB6FF4891DA236CB5FE47B8
:0600100090D9226BB4FD43
:020000020036C6
:100000000062C42688EA4CAE1072D43698FA5CBE00
:080010002082E446A80A6CCE30
:020000020041BB
:10000500001D3A577491AECBE805223F5C7996B353
:07001500D0ED0A2744617ED3
:02000002516249
:1000000000850A8F14991EA328AD32B73CC146CB98
:0400100050D55ADF8E
:0200000251B0FB
:10000000414243414243414243444444444454545BF
:0E00100044444444454545444444444454454524
:040000035162000541
:00000001FF
```

INDEX

REQUEST FOR READER'S COMMENTS

The Microcomputer Division Technical Publications Department attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

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

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

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

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. _____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

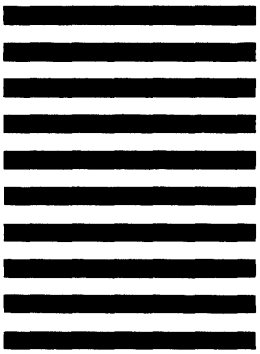
CITY _____ STATE _____ ZIP CODE _____

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 carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN U.S.A.



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications M/S 6-2000
3065 Bowers Avenue
Santa Clara, CA 95051

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

Printed in U.S.A.