iAPX 86,88 FAMILY UTILITIES
USER'S GUIDE
FOR DOS SYSTEMS

Order Number: 122395-001

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

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

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

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

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

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

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

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

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

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

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

Above	iDBP	int$_e$ligent Programming	MAPNET	QueX
BITBUS	iDIS	Intellec	MCS	QUEST
COMMputer	iLBX	Intellink	Megachassis	Quick-Erase
CREDIT	i_m	iOSP	MICROMAINFRAME	Quick-Pulse Programming
Data Pipeline	iMDDX	iPDS	MULTIBUS	Ripplemode
ETOX	iMMX	iPSC	MULTICHANNEL	RMX/80
FASTPATH	Inboard	iRMK	MULTIMODULE	RUPI
GENIUS	Insite	iRMX	ONCE	Seamless
Δ^i_i	Intel	iSBC	OpenNET	SLD
I^2ICE	int$_e$l	iSBX	OTP	SugarCube
ICE	int$_e$lBOS	iSDM	PC BUBBLE	UPI
iCEL	Intel Certified	iSXM	Plug-A-Bubble	VLSiCEL
iCS	Intelevision	KEPROM	PROMPT	4-SITE
	int$_e$ligent Identifier	Library Manager	Promware	

REV.	REVISION HISTORY	DATE	APPD.
-001	Original issue.	6/85	B.N.

Preface

This manual describes how to use the iAPX 86,88 Family utilities:

- LINK86
- CREF86
- LIB86
- LOC86
- OH86

These products run under the DOS operating system. They are used by programmers developing programs with ASM86, ASM89, PL/M-86, Pascal-86, Fortran-86, or any other language translator that produces object code compatible with the iAPX 86,88 Family of processors. The iAPX 86,88 Family of processors includes 8086, 8088, 8087, and 8089 processor chips. Because the 8086 is the first member of this family, this manual uses 8086 generically to represent the entire family.

This manual presumes familiarity with the conventions of the operating system under which the iAPX 86,88 utilities are being executed. It also presumes familiarity with the basic requirements of individual languages and translators.

This manual is divided into the following chapters:

- Chapter 1, Introduction: a summary of the relationship among the utilities and basic concepts governing their use
- Chapter 2, LINK86: how to invoke, use the controls for, and read the printed listing from LINK86
- Chapter 3, CREF86: how to invoke, use the controls for, and read the output listing from CREF86
- Chapter 4, LIB86: how to invoke and use the commands for LIB86
- Chapter 5, LOC86: how to invoke, use the controls for, and read the printed listing from LOC86
- Chapter 6, OH86: how to invoke OH86
- Chapter 7, how to use the utilities under the DOS operating system

This manual also contains several appendixes, meant for quick access to the following information:

- iAPX 86,88 absolute object file format definitions (Appendix A)
- Hexadecimal-decimal conversion information (Appendix B)
- The effect of available memory on the performance of LINK86, CREF86, LIB86, and LOC86 (Appendix C)
- Summaries of iAPX 86,88 Family utility controls and error messages:
    - LINK86 (Appendix D)
    - CREF86 (Appendix E)
    - LIB86 (Appendix F)
    - LOC86 (Appendix G)
    - OH86 (Appendix H)

Once you have gained sufficient familiarity with the basic principles of iAPX 86,88 Family utilities operation, you will find the following publication convenient for quick syntax reference:

- *iAPX 86,88 Family Utilities Pocket Reference for DOS Systems*, order number 122397.

Before reading this manual, ensure that you are familiar with the following terms and conventions.

Notational Conventions

punctuation	other than the following must be entered if required by the control syntax.
{ }	indicates that one and only one of the syntactic items contained within the braces is required.
[ ]	indicates that the syntactic item or items contained within the brackets are optional.
...	indicates that the preceding syntactic item may be repeated an indefinite number of times (the ellipsis is often used within brackets and with a comma "[,...]" to indicate that preceding item may be repeated, but each repetition must be separated by a comma)
|	vertical bar separates various options within the brackets [ ] or braces { }.
UPPERCASE	indicates that these characters must be entered exactly as shown.
italic	indicates a meta symbol which may be replaced with an item that fulfills the rules for that symbol. The actual symbol may be any of the following:
pathname	is a valid designation for a file; in its entirety, it consists of a *directory-name* and a *filename*.
directory-name	is that portion of a *pathname* which acts as a file locator by identifying the device and/or directory containing the *filename*.
filename	*is a valid name for the part of a pathname* which names a file.
minimum-size *maximum-size* *paragraph* *offset* *address*	are numbers and must follow Intel standards for number representation (see PL/M-86 or ASM86). Use the H suffix for hexadecimal, B suffix for binary, O or Q suffix for octal and D or nothing for decimal.

segment name *module-name* *class name* *group name* *overlay name* *public symbol* *variable name*	are defined by the 8086 object file formats described in Appendix A. They may be up to forty characters long and may contain any of the following characters in any order: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, X, X, Y, Z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ?, @, :, ., _
black	black background is used in examples to indicate the user's entries.
system-id	is a generic label placed on sample listings where an operating system-dependent name would actually be printed.
pathname1 *pathname2, ...*	are generic labels placed on sample listings where one or more user-specified pathnames would actually be printed.
Vx.y	is a generic label placed on sample listings where the version number of the product that produced the listing would actually be printed.

Related Publications

The following list provides the manual title and order number for all Intel software development tools that run on DOS systems. Note that each manual has two formats and two order numbers. One version of the manual is provided in a binder. This version is not sold separately; it can only be purchased when purchasing a software product. The second version, which has a soft cover, is sold separately. Use the soft cover number when ordering a manual separately.

Manual Title	Binder	Soft Cover
ASM86 Assembly Language Reference Manual	122385	122386
ASM86 Macro Assembler Operating Instructions for DOS Systems	122390	122391
ASM86 Pocket Reference for DOS Systems	122387	
Operating System Interface Libraries Manual	122400	122401
8087 Support Library Reference Manual	122405	122406
ic-86 Compiler User's Guide for DOS Systems	122415	122416
The C Programming Language	122008	
PL/M-86 User's Guide for DOS Systems	122410	122411
PL/M-86 Pocket Reference for DOS Systems	122412	
Pascal-86 User's Guide for DOS Systems	122425	122426
Pascal-86 Pocket Reference for DOS Systems	122427	
Fortran-86 User's Guide for DOS Systems	122430	122431
Fortran-86 Pocket Reference for DOS Systems	122432	

Table of Contents

Page

Chapter 5
LOC86

Chapter 6
OH86

Chapter 7
Using the iAPX 86,88 Utilities under DOS

Figures

Tables

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.

Figure 1-1 shows the development process and the tools available for development of an iAPX 86,88 Family-based product.

The tools described in this manual are:

- LINK86, which is a linkage and binding tool
- CREF86, which provides a cross-reference of information on symbols in several modules
- LIB86, which is the librarian function for 8086 object modules
- LOC86, which is the relocation tool
- OH86, which converts 8086 absolute object information to the hexadecimal format

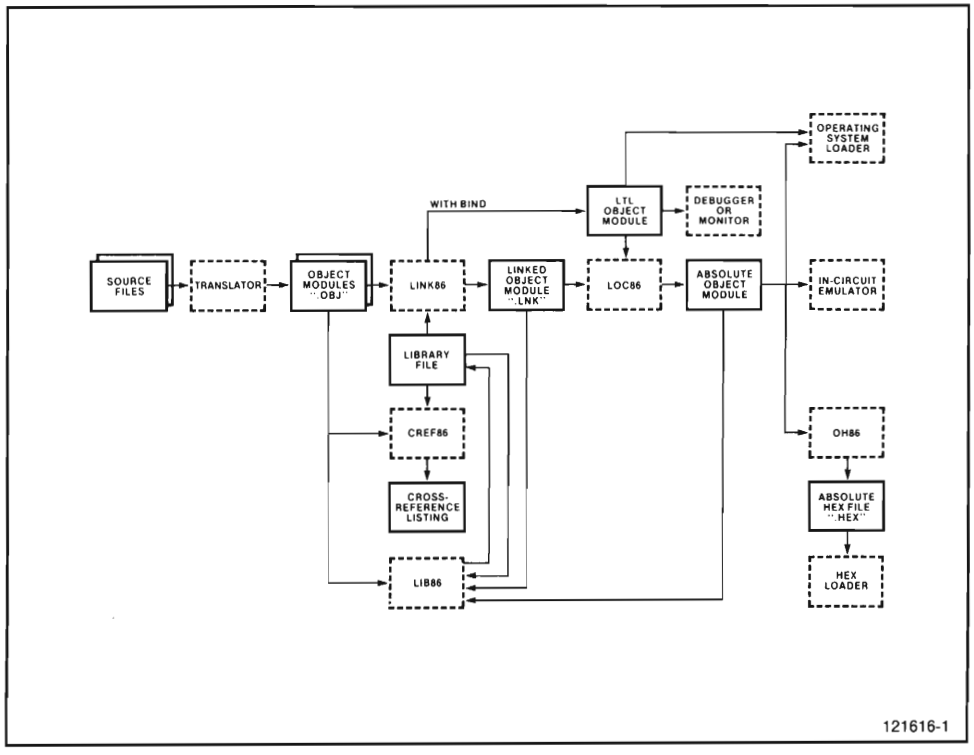


Figure 1-1. The iAPX 86,88 Family Development Process

Overview of the Utilities

ASM86, ASM89, PL/M-86, PASCAL-86, FORTRAN-86, and other translators as well as LINK86 and LOC86 produce 8086 object modules. The language translators produce 8086 relocatable object modules that must usually be processed by utilities before execution. (Under certain circumstances the translators can produce absolute object modules, but this is rare and does not contribute to modular design.) LINK86 combines 8086 object modules, and LOC86 converts relocatable object modules into absolute object modules. OH86 converts 8086 absolute object modules to 8086 hexadecimal format.

LINK86 combines a list of 8086 object modules into a single object module and attempts to match all external symbol declarations with their public symbol definitions in library modules. (LIB86 is the utility used to create and maintain program libraries.) The output of LINK86 is a relocatable object module. However, when specified in the controls, LINK86 produces a load-time-locatable (LTL) object module; an LTL module can be executed on an 8086-based system. (See the description of LTL modules later in this chapter.) Whether the LINK86 output is an LTL or a relocatable object module, it can serve as input to LOC86.

CREF86 provides a means for producing a cross-reference listing of public and external symbols in multiple 8086 object modules. The object modules may include library modules. The output produced by CREF86 should help the programmer to identify how symbols will be resolved by LINK86, given the same input files.

LOC86 converts relocatable (or LTL) object modules to absolute object modules. Absolute object modules contain references that require the module segments to be placed at particular places in 8086 memory.

The sequence in which the segments in the input modules are combined and absolute addresses assigned to segments is determined by the controls supplied and the order in which the modules are listed in the LINK86 and LOC86 invocations.

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 modules 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) utilities, 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 define these symbols as public when coding the program. This tells the source translator and the R&L utilities 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 and CREF86 treat library files in a special manner. If you specify a library file in the input to these utilities, they search the library for modules that satisfy unresolved external references in the input modules *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 have been the subject of a search of all public symbols in the library modules.

When LINK86 and CREF86 search a library, they *normally* include only library modules that satisfy external references in the output. If no external references are satisfied by a library, no modules from the library are included in the LINK86 output module or the CREF86 output listing. However, LINK86 and CREF86 provide 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.

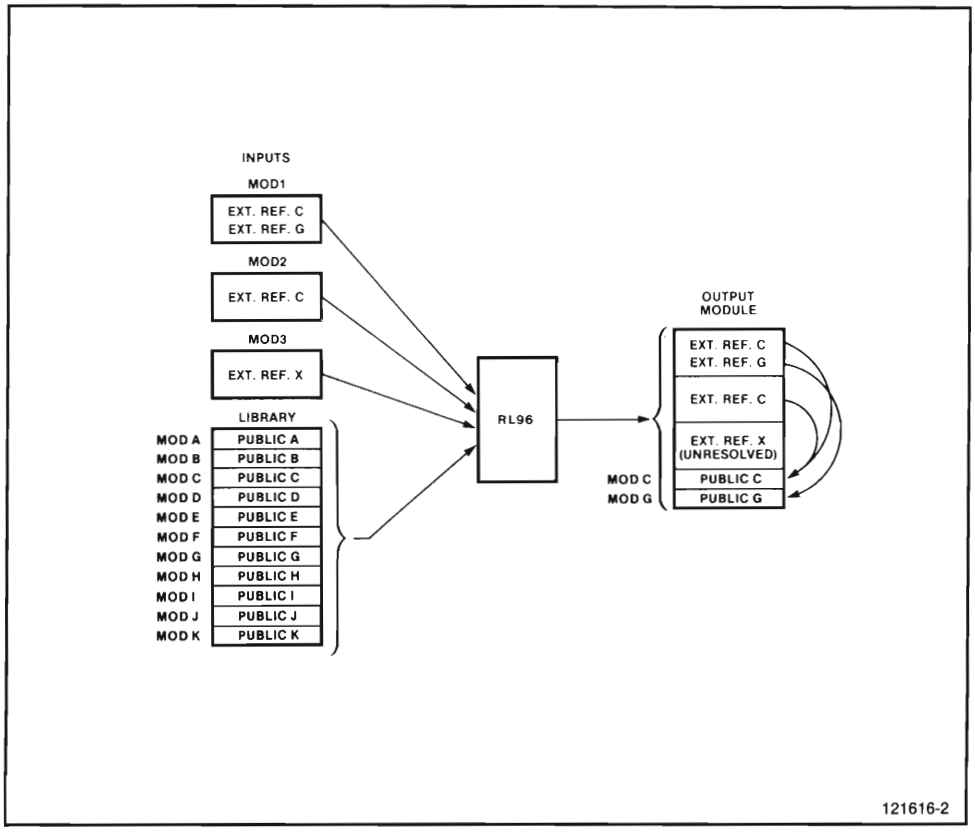


Figure 1-2. Library Linkage by LINK86

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.

After LINK86 combines all the input segments, LOC86 can be used to assign 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. If LINK86 is used to create a bound object module, LOC86 is not needed to execute the program.

The LINK86/LOC86 Process

Although controls are not required for LINK86 and LOC86 execution, the commands invoking them may contain controls that affect their output. The controls make it possible to change the defaults for module combination, address assignment, and output information.

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 8086 translators and LINK86 and LOC86 themselves.

LINK86 combines segments from the input modules, and for LTL object modules LINK86 orders segments in groups and assigns offsets. 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.

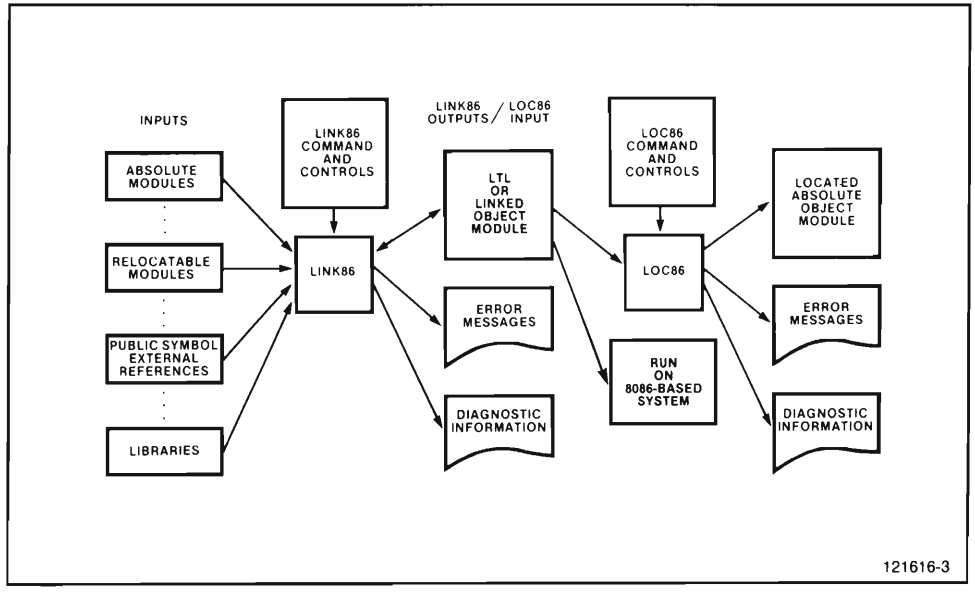


Figure 1-3. The LINK86/LOC86 Process

An 8086 Overview

To use the R&L commands you must have an understanding of the following concepts:

- Addresses, given as offsets into segments, which must be translated into absolute memory addresses, or base offsets
- Segment definitions, which identify contiguous pieces of information, usually code or data
- Class definitions, which identify segments that share common attributes and should be kept together
- Group definitions, which identify segments that must be kept within a 64K byte range of memory
- Overlay definitions, which identify modules that will be loaded in memory at different times during execution.
- Load-time-locatable object modules

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 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 changing only 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.

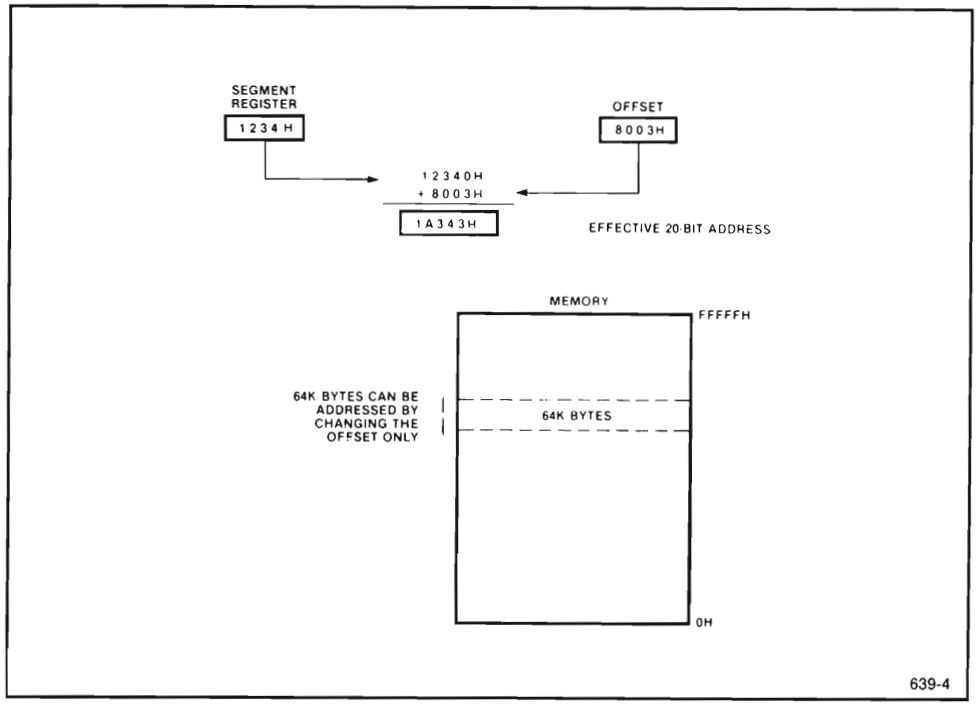


Figure 1-4. 8086 Addressing

Segments

Programs comprise 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/M-86 compiler may generate predefined names for segments.

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, class and overlay) 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 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.

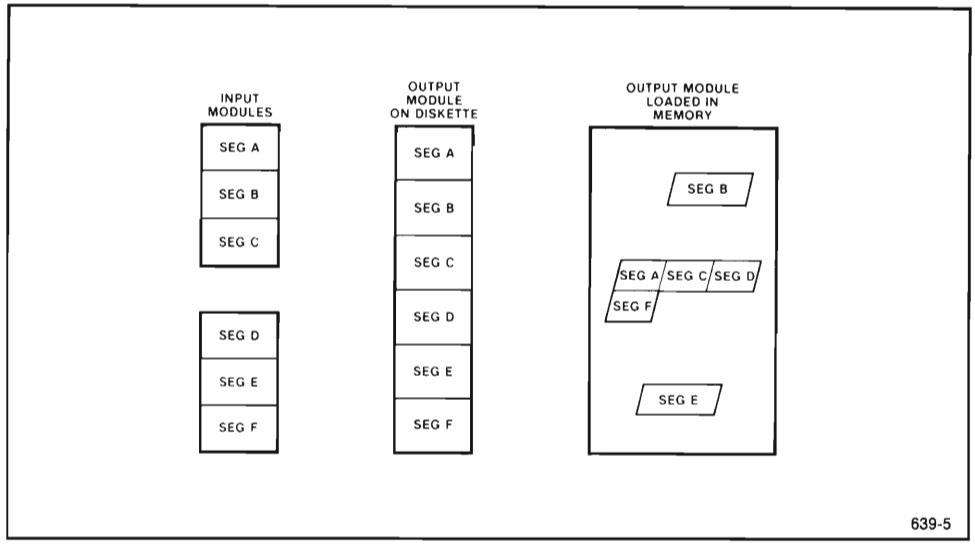


Figure 1-5. Segment Physical Relationships

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 be located only at an address that is a multiple of two, starting from address 0H
- Paragraph, which means a segment can be located only at an address that is a multiple of 16, starting from address 0
- Page, which means a segment can be located only 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.

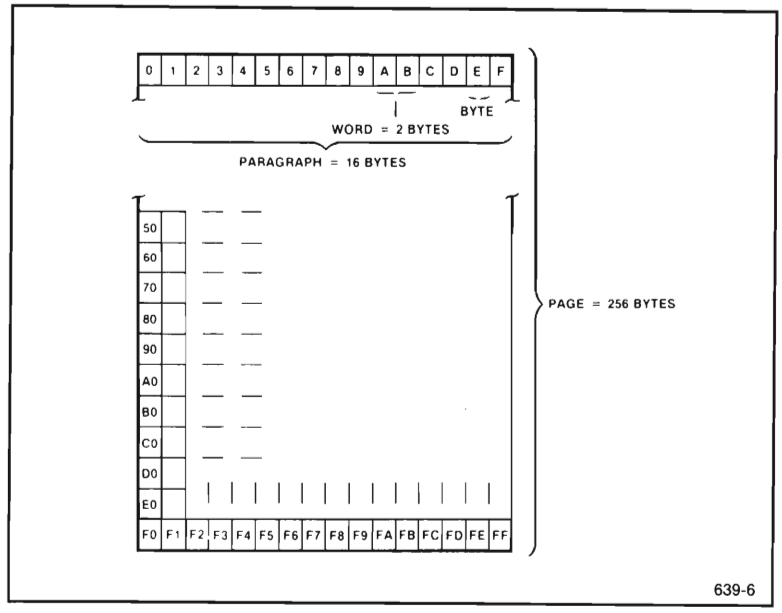


Figure 1-6. Segment Alignment Boundaries

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. Such translators as PL/M-86 define 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 LOC86 locating controls.

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 get 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 such translators as PL/M-86 are generated with predefined class names. 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 that 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.

Overlays

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 overlaid program is always resident in memory; it usually comprises the main program module, frequently used routines, and the overlay loader. This part of the program is called the root. Figure 1-7 illustrates the memory configuration of one program that uses overlays.

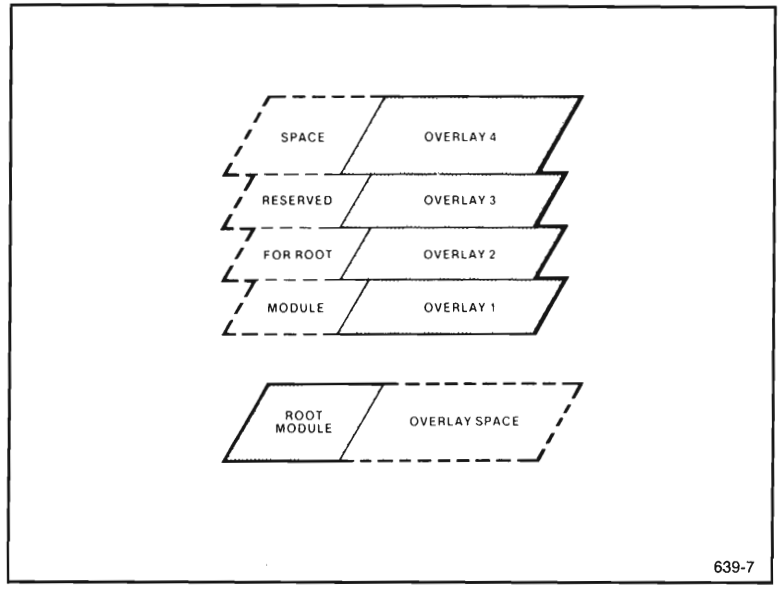


Figure 1-7. Memory Configuration of Program with Overlays

Position-Independent Code and Load-Time-Locatable Code

An LTL (load-time-locatable) program can be loaded anywhere in memory (assuming alignment attributes are honored). Code and data addresses are assigned by the system loader. References to segment bases (segment registers) are permitted. The loader, when it determines where to locate each segment, must resolve these references to the segment bases. Before executing the LTL program, the loader must also initialize the segment registers.

A PIC (position-independent-code) program is an LTL program, but it contains no references to segment bases. To execute these programs the loader need only place the program in memory (recognizing alignment attributes) and initialize the segment registers and go. No fixup of segment bases is required.

LINK86 2

LINK86 combines 8086 object modules and resolves references between independently translated modules. LINK86 takes a list of files and controls as input and produces two output files: a print file and an object file.

Figure 2-1 illustrates the linkage process. The input files may be any object module (output from a translator, LINK86, LOC86 or an 8086 library file). The print file contains diagnostic information. The output object file is a bound load-time-locatable module or simply a relocatable module.

This chapter provides details concerning the LINK86 invocation, controls, and print file. For definition of file-naming and syntax notation conventions used in this chapter, refer to Notational Conventions following the Preface. For a summary of the LINK86 controls and information on error and warning messages that may be produced, refer to Appendix D. For details concerning symbol table space limitations, refer to Appendix C.

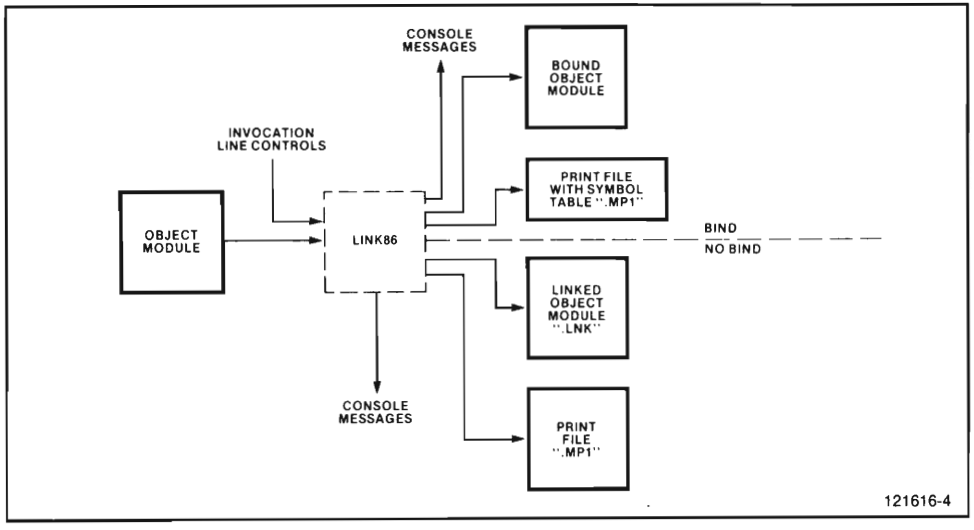


Figure 2-1. LINK86 Input and Output Files

LINK86 Invocation Line

The general syntax for the invocation line is:

[*directory-name*]`LINK86` *input list*[`TO` *output file*][*controls*]

The *input list* is one or more modules to be linked together into a single object module:

pathname[`(`*module name*[`,`...]`)`][`,`...]

Unless a *module name* is specified, all modules in a *pathname* are included. If the *pathname* is a library file, any modules named in parentheses are included in the *output file* even if they do not contain public symbol definitions for external symbols declared elsewhere in the input list.

The *input list* may also contain the control PUBLICSONLY before selected *pathnames*. If you wish to include a file called PO or PUBLICSONLY, ensure that the filename is preceded by a directory-name in order to distinguish it from the control or its abbreviation.

The order of modules in the input list affects the order of segments in the output file.

TO *output file* designates the file to receive the linked object module. If *output file* is not specified, then output is directed to a file that has the same pathname as the first element in the input list, but its extension is .LNK. If the first element in the list is a PUBLICSONLY control, then the first pathname in its argument is used for the default name.

If the BIND control is specified, then the default name for the output file has no extension, and the object module can be executed without locating.

The *controls* can be any subset of the controls specified in the next section.

LINK86 Controls

The *controls* are described in table 2-1.

Table 2-1. Summary of LINK86 Controls

Control	Abbrev.	Default
ASSIGN({ *variable*(address) } [,...])	AS	Not applicable
ASSUMEROOT(*pathname*)	AR	Not applicable
BIND	BI	NOBIND
NOBIND	NOBI	
COMMENTS	CM	COMMENTS
NOCOMMENTS	NOCM	
FASTLOAD	FL	NOFASTLOAD
NOFASTLOAD	NOFL	
GROUPOVERLAYS	GO	GROUPOVERLAYS
NOGROUPOVERLAYS	NOGO	
INITCODE	IC	Not applicable
LINES	LI	LINES
NOLINES	NOLI	
MAP	MA	MAP
NOMAP	NOMA	
MEMPOOL(*min-size* [, *max-size*])	MP	Not applicable
NAME(*module name*)	NA	Not applicable
OBJECTCONTROLS({ LINES | NOLINES | COMMENTS | NOCOMMENTS | SYMBOLS | NOSYMBOLS | PUBLICS [EXCEPT(*symbol* [,...])] | NOPUBLICS [EXCEPT (*symbol* [,...])] | TYPE | NOTYPE | PURGE | NOPURGE } [,...])	OC	Not applicable

Table 2-1. Summary of LINK86 Controls (Cont'd.)

Control	Abbrev.	Default
ORDER({ *group*({ *segment* [\ *class* [\ *overlay*]] } [,...]) } [,...])	OD	Not applicable
OVERLAY[(*overlay*)]	OV	NOOVERLAY
NOOVERLAY	NOOV	
PRINT[(*pathname*)]	PR	PRINT(*object file*.MP1)
NOPRINT	NOPR	
PRINTCONTROLS({ LINES | NOLINES | COMMENTS | NOCOMMENTS | SYMBOLS | NOSYMBOLS | PUBLICS[EXCEPT(*symbol*[,...])] | NOPUBLICS [EXCEPT(*symbol*[,...])] | TYPE | NOTYPE | PURGE | NOPURGE } [,...])	PC	Not applicable
PUBLICS [EXCEPT(*symbol*[,...])]	PL [EC]	PUBLICS
NOPUBLICS [EXCEPT(*symbol*[,...])]	NOPL[EC]	
PUBLICSONLY(*pathname*[,...])	PO	Not applicable
PURGE	PU	NOPURGE
NOPURGE	NOPU	
RENAMEGROUPS({ *group* TO *group* } [,...])	RG	Not applicable
SEGSIZE({ *segment*[\ *class*[\ *overlay*]] (*min-size*[,[*max-size*]]) } [,...])	SS	Not applicable
SYMBOLS	SB	SYMBOLS
NOSYMBOLS	NOSB	
SYMBOLCOLUMNS({ 1 | 2 | 3 | 4 })	SC	SYMBOLCOLUMNS(2)
TYPE	TY	TYPE
NOTYPE	NOTY	

If you specify a control more than once in a single invocation line, only the last version entered counts. For example, if you enter NOMAP on the invocation line and then later decide you want a link map, you can specify MAP. The ASSIGN control, however, is an exception to this general rule.

The following controls are effective only when the BIND control is specified:

FASTLOAD
MEMPOOL
ORDER
PRINTCONTROLS
SEGSIZE
SYMBOLCOLUMNS

The following control is effective only when the BIND control is NOT specified:

INITCODE

The following control is effective only when the OVERLAY control is specified:

ASSUMEROOT

See Chapter 7 for operating system information and for operating system-specific examples of the LINK86 controls.

ASSIGN

Syntax

ASSIGN({*variable-name*(*address*)}[,...])

Abbreviation

AS

Default

Not applicable

Definition

ASSIGN makes it possible to define absolute addresses for symbols at LINK time. The absolute address associated with the *variable-name* is specified in *address*, which should be an absolute 20-bit memory address that conforms to PL/M-86 notation. The *variable-name* is internally defined as a PUBLIC symbol.

Notes

- This control is particularly useful for memory-mapped I/O.
- If the *variable-name* has a matching public definition in another module, the public definition in that module is flagged as as duplicate. Whenever a reference to the *variable-name* occurs, the variable defined in the ASSIGN control governs.
- If multiple ASSIGN specifications are provided in one LINK86 invocation, *all* will be effective (not only the final entry).

ASSUMEROOT

Syntax

ASSUMEROOT (*pathname*)

Abbreviation

AR

Default

Not applicable

Definition

ASSUMEROOT suppresses the inclusion of any library module(s) in an overlay if the library module(s) have already been included in a root file identified by *pathname*. When this control is used, the root file is scanned, and all external, undefined symbols in the overlay modules which have a matching definition in the root file are marked "temporarily resolved." This marking means that while a library search for the symbols will not be made, their status remains externally undefined until the overlays are linked with the root.

Notes

- This control should be used only in conjunction with the OVERLAY control and libraries.
- This control will not eliminate common library modules from overlay to overlay.
- This control may not be used when an input module already has an overlay record.

BIND/NOBIND

Syntax

```
BIND
NOBIND
```

Abbreviation

```
BI
NOBI
```

Default

```
NOBIND
```

Definition

BIND combines the input modules into a load-time-locatable (LTL) module. An LTL module may be loaded and executed, and any logical reference to a segment or group base can be resolved at load time. The load-time-locatable output cannot be loaded by the ICE-86 loader or UPM.

Notes

- FASTLOAD, MEMPOOL, ORDER, SEGSIZE, and SYMBOLCOLUMNS have no effect when NOBIND is specified.
- When NOBIND is in effect, [NO]LINES, [NO]SYMBOLS, [NO]PUBLICS, and [NO]PURGE affect only the output object module.
- When BIND is specified, the default object file name has no extension.

COMMENTS/NOCOMMENTS

Syntax

```
COMMENTS
NOCOMMENTS
```

Abbreviation

```
CM
NOCM
```

Default

```
COMMENTS
```

Definition

COMMENTS allows object file comment records to remain in the output module. The NOCOMMENTS control removes all comment records except those designated as nonpurgable.

Comment records are added to the object module for various reasons. All translators add a comment record, identifying the compiler or assembler that produced it.

Comment records are superfluous to the production of executable code and may be removed at any time during the development process.

Notes

- See PURGE, PRINTCONTROLS and OBJECTCONTROLS.
- COMMENT records should not be removed when you submit an object file in a Software Problem Report.
- NOCOMMENTS will decrease the size of the output object file.

FASTLOAD/NOFASTLOAD

Syntax

```
FASTLOAD
NOFASTLOAD
```

Abbreviation

```
FL
NOFL
```

Default

```
NOFASTLOAD
```

Definition

FASTLOAD reduces program loading time by causing data record concatenation. The data records are concatenated to a maximum length of 64K. FASTLOAD also makes the object file compact by removing such information as local symbols, public records, comments, and type information (unless the object file contains unresolved external symbols).

Notes

- This control is effective only when BIND is specified.
- Output produced with this control in effect may be incompatible with LINK86 versions earlier than 2.0.

GROUPOVERLAYS/ NOGROUPOVERLAYS

Syntax

```
GROUPOVERLAYS
NOGROUPOVERLAYS
```

Abbreviation

```
GO
NOGO
```

Default

```
GO
```

Definition

GROUPOVERLAYS causes LINK86 to optimize static memory space for LARGE Model Overlay programs. LINK86 achieves this optimization by creating Group definitions containing logically separate segments from the various overlays.

Notes

- GROUPS created by LINK86 are not listed in the group map of the print file.
- Segments that are already in some group are not affected.
- The STACK and MEMORY segments are not placed in any group created by the linker.

Notes

- The *minimum-size* must be less than or equal to the *maximum-size*.
- MEMPOOL has no effect unless the BIND control is also specified.
- The static size of a program is the size of the memory required to load the program itself; it is automatically calculated by the linker.

  Dynamic size is the size of the memory the program will need during its execution; it is specified by the user with the MEMPOOL control.

INITCODE

Syntax

```
INITCODE
```

Abbreviation

```
IC
```

Default

Not applicable

Definition

INITCODE causes LINK86 to create a new segment that contains code to initialize the segment registers. The equivalent assembly language code is shown below:

```
STACKFRAME   DW    stack frame
DATAFRAME    DW    data frame
EXTRAFRAME   DW    extra frame
             CLI
             MOV   SS, CS:STACKFRAME
             MOV   SP, stack offset
             MOV   DS, CS:DATAFRAME
             MOV   ES, CS:EXTRAFRAME
             JMP   program start
```

Notes

- The initialization code segment is created only if a register intialization record for 8086 segment registers exists in the input. These register initialization records are automatically produced by 8086-based translators for main modules.
- BIND and OVERLAY controls used in conjunction with INITCODE will cause LINK86 to ignore the INITCODE control and issue a warning message.
- INITCODE should be used to ensure compatibility with 8085-based LINK86, LOC86, and LIB86 products.
- The name of the new segment, if created, is ??INITCODE.

LINES/NOLINES

Syntax

```
LINES
NOLINES
```

Abbreviation

```
LI
NOLI
```

Default

```
LINES
```

Definition

LINES allows line number information to remain in the object file. ICE-86 and other debuggers use this information. The line number information is not needed to produce executable code. The NOLINES control removes this information from the output file.

Notes

- See PRINTCONTROLS and OBJECTCONTROLS.
- See the PURGE control.
- NOLINES will decrease the size of the output object file.
- Unless BIND is in effect, LINES/NOLINES affects only the object module.
- LINES has no effect on local symbols; the inclusion of local symbol records in the object file is controlled by SYMBOLS.

MAP/NOMAP

Syntax

```
MAP
NOMAP
```

Abbreviation

```
MA
NOMA
```

Default

```
MAP
```

Definition

MAP produces a link map and inserts it in the PRINT file. The link map contains information about the attributes of logical segments in the output module. This includes size, class, alignment attribute, address (if the segment is absolute) and overlay name (if the segment is a member of an overlay).

NOMAP inhibits the production of the link map.

Notes

- MAP can be overridden by the NOPRINT control.
- See the discussion of the link map at the end of this chapter.

MEMPOOL

Syntax

MEMPOOL(*minimum-size*[,*maximum-size*])

Abbreviation

MP

Default

Not applicable.

Definition

MEMPOOL specifies the dynamic memory requirements of the program. This allows the loader to check free memory at load time, and prevent a run-time error.

The *minimum size* is a 20-bit number. There are three ways of specifying this value:

- + indicates that the number should be added to the current dynamic memory requirements.
- — indicates that the number should be subtracted from the current dynamic memory requirements.
- no sign indicates that the number should become the new minimum dynamic memory requirement.

The *maximum size* is a 20-bit number. There are two ways of specifying this value:

- + indicates that the number should be added to the current minimum dynamic memory requirement.
- no sign indicates that the number should become the new maximum dynamic memory requirement.

Notes

- The *minimum-size* must be less than or equal to the *maximum-size*.
- MEMPOOL has no effect unless the BIND control is also specified.
- The static size of a program is the size of the memory required to load the program itself; it is automatically calculated by the linker.

  Dynamic size is the size of the memory the program will need during its execution; it is specified by the user with the MEMPOOL control.
- For the STACK segment, only the maximum size can be set.

NAME

Syntax

NAME (*module name*)

Abbreviation

NA

Default

The module name of the first element in the input list.

Definition

NAME assigns the specified *module name* to the output module. If NAME is not specified, then the output module will have the name of the first module in the input list.

The *module name* may be up to 40 characters long. It may be composed of any of the following characters in any order:

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

Lower-case letters may be used, but they are automatically converted to uppercase.

Notes

- NAME does not affect the output file's name. Only the module name in the output module's header record is changed.

OBJECTCONTROLS

Syntax

```
OBJECTCONTROLS({LINES | NOLINES |
                COMMENTS | NOCOMMENTS |
                SYMBOLS | NOSYMBOLS |
                PUBLICS [EXCEPT(symbol[,...])] |
                NOPUBLICS [EXCEPT(symbol[,...])]|
                TYPE | NOTYPE |
                PURGE | NOPURGE}[,...]
                )
```

Abbreviation

OC

Default

Controls apply to both the print file and the object file.

Definition

OBJECTCONTROLS causes the controls specified in its arguments to be applied to the object file only. Comment records, line number records, local and public symbol records, and symbol type records are selectively included or excluded from the object file. This will not affect the print file nor the information contained in it.

Notes

- Abbreviations for the controls within the parentheses may be given.
- A control specified in *both* OBJECTCONTROLS and PRINTCONTROLS has the same effect as specifying it *once* outside of these controls.

ORDER

Syntax

```
ORDER({group name({segment name
                        [\class name[\overlay name]]}
                    [,...])}
      [,...])
```

Abbreviation

OD

Default

Segments are placed into object file in the same order in which they were encountered in the input list.

Definition

ORDER specifies a partial or complete order for the segments in one or more groups.

The *group name* identifies the group whose segments are to be ordered.

The *segment name* identifies the segments to be ordered. The *class name* and *overlay name* may be used to resolve conflicts with duplicate segment names. If *overlay name* is specified, the *class name* is required.

Notes

- ORDER has no effect unless BIND is also specified.
- If one of the segments specified is not contained in the designated group, an error message is generated.
- See discussion of module combination at the end of the chapter for details of the default ordering.

OVERLAY/NOOVERLAY

Syntax

```
OVERLAY[(overlay name)]
NOOVERLAY
```

Abbreviation

```
OV
NOOV
```

Default

```
NOOVERLAY
```

Definition

OVERLAY specifies that all of the input modules shall be combined into a single overlay module. When the optional *overlay name* argument is specified, all segments contained within the overlay module have that name in addition to their segment names and class names. When *overlay name* is not specified, LINK86 uses the module name of the first module in the input list.

Notes

- Each overlay in a given program must be linked separately before they are all linked into a single object module.
- The overlay specified in the argument must be the same as the overlay name used when calling the operating system to load the overlay.
- When linking root and overlay files, LINK86 assumes the first file in the invocation line is the root.
- The ASSUMEROOT control can be specified in conjunction with the OVERLAY control.

PRINT/NOPRINT

Syntax

```
PRINT[(pathname)]
NOPRINT
```

Abbreviation

```
PR
NOPR
```

Default

```
PRINT(object file.MP1)
```

Definition

PRINT allows you to direct the link map and other diagnostic information to a particular file. If the PRINT control is not specified or if the control is given without an argument, the print file will have the same pathname as the output file except the extension will be .MP1. NOPRINT prevents the creation of this file.

Notes

- The discussion at the end of this chapter describes the contents of the print file.
- MAP, SYMBOLCOLUMNS, LINES, SYMBOLS, PUBLICS and PRINTCONTROLS affect the contents of the print file.

PRINTCONTROLS

Syntax

```
PRINTCONTROLS({LINES | NOLINES |
               COMMENTS | NOCOMMENTS |
               SYMBOLS | NOSYMBOLS |
               PUBLICS [EXCEPT(symbol[,...])] |
               NOPUBLICS [EXCEPT(symbol[,...])]|
               TYPE | NOTYPE |
               PURGE | NOPURGE} [,...]
               )
```

Abbreviation

PC

Default

Controls apply to both the print file and the object file.

Definition

PRINTCONTROLS causes the controls specified in its arguments to be applied to the print file only. Line number information, and local and public symbol information are selectively included or excluded from the print file. This will not affect the object file or the information contained in it.

Notes

- When a control is specified in both the PRINTCONTROLS and the OBJECTCONTROLS, it has the same effect as specifying it once outside of these controls.
- Abbreviations to the parenthesized controls may be used.
- Unless BIND is specified, PRINTCONTROLS and its arguments have no effect.

PUBLICS/NOPUBLICS

Syntax

```
PUBLICS [EXCEPT(public symbol [,...])]
NOPUBLICS[EXCEPT(public symbol [,...])]
```

Abbreviation

```
PL [EC]
NOPL [EC]
```

Default

```
PUBLICS
```

Definition

PUBLICS causes the public symbol records to be kept in the object file and the corresponding information to be placed in the print file. Public symbol records are needed to resolve external symbol definitions in other files. The EXCEPT subcontrol allows you to modify the control. Public records are used by LINK86 to resolve external references.

Notes

- The scope of PUBLICS can be modified by PRINTCONTROLS and OBJECTCONTROLS.
- Unless BIND is specified PUBLICS/NOPUBLICS affect only the object file.
- NOPUBLICS will decrease the size of the output object file.

PUBLICSONLY

Syntax

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

Abbreviation

PO

Default

Not applicable

Definition

PUBLICSONLY is an input list control. When used it must appear in the input list and not the control list.

PUBLICSONLY indicates that only the absolute public symbol records of the argument files will be used. The other records in the module will be ignored. This can be used to resolve external references to 8089 files and overlays when a multifile overlay system is desired.

Notes

- Although it is possible to create overlays using PUBLICSONLY, it is easier to use the OVERLAY control to create overlays.

PURGE/NOPURGE

Syntax

```
PURGE
NOPURGE
```

Abbreviation

```
PU
NOPU
```

Default

```
NOPURGE
```

Definition

PURGE in the control list is exactly the same as specifying NOLINES, NOSYMBOLS, NOCOMMENTS, NOPUBLICS, and NOTYPE. NOPURGE in the control list is the same as specifying LINES, SYMBOLS, COMMENTS, PUBLICS, and TYPE.

PURGE removes all of the debug or public records from the object file and their information from the print file. It will produce the most compact object file possible.

The records that would be included by NOPURGE are useful to debuggers, but otherwise they are unnecessary for producing executable code.

Notes

- PRINTCONTROLS and OBJECTCONTROLS can be used to modify the scope of PURGE.
- Unless BIND is specified, PURGE affects only the output object file.

RENAMEGROUPS

Syntax

RENAMEGROUPS({*group name* TO *group name*} [,...])

Abbreviation

RG

Default

All groups keep the name they already have.

Definition

RENAMEGROUPS allows you to change the group names assigned by the translator. The first *group name* must be an existing group in one of the modules in the input list.

Notes

None

SEGSIZE

Syntax

```
SEGSIZE({segment name[\class name[\overlay name]]
                (minimum size[,[maximum size]])}
        [,...])
```

Abbreviation

SS

Default

Not applicable

Definition

SEGSIZE allows you to specify the minimum memory space needed for any segment. If you specify the maximum size for a segment, that segment must either not be a member of any group or be the last segment in the group.

The *segment name* identifies the segment whose size is to be changed.

The *minimum size* is a 16-bit number. There are three ways of specifying this value:

- + indicates that the number should be added to the current segment length.
- — indicates that the number should be subtracted from the current segment length.
- no sign indicates that the number should become the new segment length.

The *maximum size* is a 16-bit number. There are two ways of specifying this value:

- + indicates that the number should be added to the minimum segment length.
- no sign indicates that the number should become the new maximum segment length.

Notes

- The maximum segment size must always be greater than or equal to the minimum segment size.
- Segment lengths are initially assigned by the translator.
- Unless BIND is also specified SEGSIZE has no effect.
- For the STACK segment, only the maximum size can be set.

SYMBOLS/NOSYMBOLS

Syntax

```
SYMBOLS
NOSYMBOLS
```

Abbreviation

```
SB
NOSB
```

Default

```
SYMBOLS
```

Definition

SYMBOLS specifies that all local symbol records shall be included in the object file. Local symbol records are used by debuggers.

Notes

- Unless BIND is also specified, SYMBOLS affects only the output object file.
- NOSYMBOLS will decrease the size of the output object file.
- SYMBOLS has no effect on line numbers; the inclusion of line numbers in the object file is controlled by the LINES control.

SYMBOLCOLUMNS

Syntax

```
SYMBOLCOLUMNS({1 | 2 | 3 | 4})
```

Abbreviation

```
SC
```

Default

```
SYMBOLCOLUMNS(2)
```

Definition

SYMBOLCOLUMNS indicates the number of columns to be used when producing the symbol table for the object module. Two columns fit on a 78-character line; four columns fit on a single 128-character line printer line.

Notes

- SYMBOLCOLUMNS has no effect unless BIND is also specified.

TYPE/NOTYPE

Syntax

```
TYPE
NOTYPE
```

Abbreviation

```
TY
NOTY
```

Default

TYPE

Definition

TYPE specifies that type checking is to be performed on the object file. Symbol type records produced by the translator are used by LINK86 to perform type checking on modules. Symbol type records should be kept in the file if it may be relinked with another file.

Notes

- NOTYPE will decrease the size of the ouput object file without affecting run-time operation.

LINK86's Print File

The print file is always created unless you specify NOPRINT. The optional argument to PRINT designates the name of the print file. The default print file is the object file with the extension .MP1.

The print file may contain as many as five parts:

1. A header (always in the print file)
2. A link map (requires MAP)
3. A group map (requires BIND)
4. A symbol table (requires BIND and PUBLICS, LINES, or SYMBOLS)
5. An error message list (always included when they occur)

The Header

The header is self-explanatory; it identifies the 8086 linker by version number and gives the important details about the input and output files used during this execution. Figure 2-2 shows an example of LINK86's print file header.

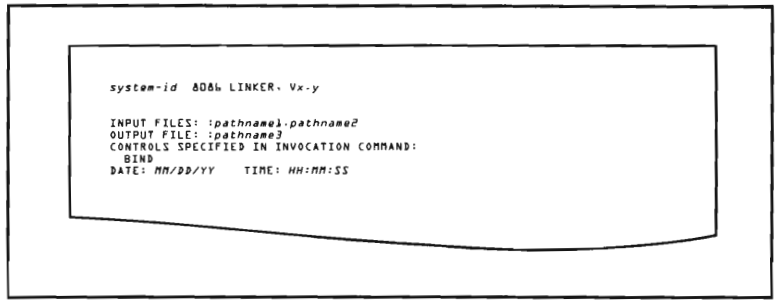


Figure 2-2. LINK86 Print File Header

The Link Map

The link map supplies useful information about segments in the object file — order, size, alignment attribute, and segment, class, and overlay names. Figure 2-3 shows LINK86's link map.

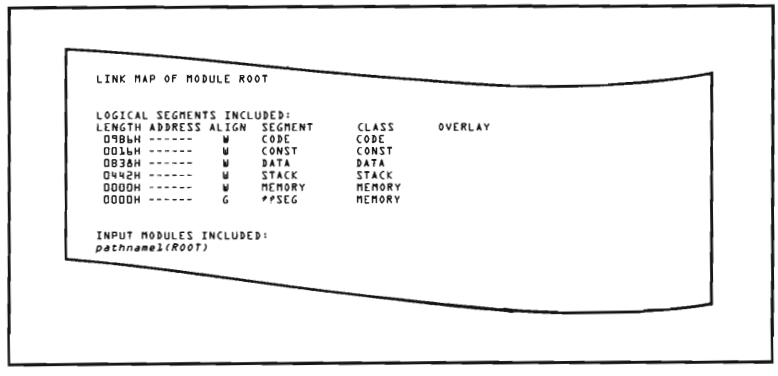


Figure 2-3. LINK86 Link Map

The map consists of three parts:

- Segment map
- Input module list
- Unresolved symbol list

The segment map describes all of the segments included in the object file. Each segment description includes five entries: length, the address (if the segment is absolute), alignment attribute, segment name, class name and overlay name, if any.

A segment may have any one of the following alignment attributes:

A absolute
B byte
G paragraph
M member of an LTL group
P page
W word
R in-page

In-page alignment means that the entire segment must be resident within a single 256-byte page. The address of the first byte in any page has zeros in the first 2-hexadecimal digits (00H, 100H, 200H,...0FFF00H).

The module list identifies the order of modules included in the output file. LINK86 gives both the file containing the module and the module name for each entry in the list.

The unresolved symbol list itemizes each external symbol whose public definition was not encountered. The module that references the unresolved symbol is also indicated. The printed message that appears under the heading UNRESOLVED EXTERNAL NAMES is as follows:

- *symbol name* `IN` *pathname* `(`*module name*`)`
- If ASSUMEROOT is specified, the message would read:

  symbol name `(DEFINED IN ROOT-FILE,`*pathname*`)`
- If PUBLICS/NOPUBLICS EXCEPT is specified, the message would read:

  symbol name `IN LINK86 COMMAND LINE`

The Group Map

LINK86 produces a group map when the BIND control is specified. Each group name and all segments contained in that group are listed. The offset from the group base for each segment appears to the right of the segment name. Figure 2-4 shows an example of the group map.

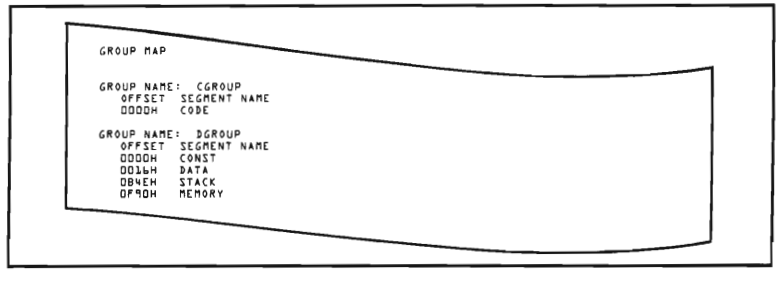


Figure 2-4. LINK86 Group Map

The Symbol Table

LINK86 produces a symbol table only when the following conditions are true:

1. BIND is specified
2. PRINT and MAP controls are in effect.
3. At least one of the following controls is in effect: PUBLICS, LINES, or SYMBOLS.

Figure 2-5 shows LINK86's symbol table with the SYMBOLCOLUMNS set at two (the default). The symbol table is shown in two parts: the top section contains the public symbol information; the lower section contains line and local symbol information.

```
SYMBOL TABLE OF MODULE ROOT

BASE     OFFSET  TYPE  SYMBOL                BASE   OFFSET  TYPE  SYMBOL

G(2)     0166H   PUB   BINDCONTROL           G(2)   004CH   PUB   BNODEBASE
G(2)     0018H   PUB   BUFBASE              G(2)   0016H   PUB   BUFLEN
G(2)     004EH   PUB   CLASHNODEBASE        G(2)   0060H   PUB   COCONN
G(2)     015AH   PUB   COMMENTSCONTROL      G(2)   0171H   PUB   CURRENTOVERLAYNU
                                                                 -M
G(2)     0173H   PUB   DEBUGTOGGLE          G(2)   00A7H   PUB   DEFAULTPRTFILENA
                                                                 -ME
G(2)     0062H   PUB   EXCEPTION            G(2)   0048H   PUB   FANODEBASE
G(2)     006EH   PUB   FBLOCKBASE           G(2)   006AH   PUB   FBLOCKLISTHEAD
G(2)     006CH   PUB   FBLOCKLISTTAIL       G(2)   013DH   PUB   FBLOCKSEQUENCENU
                                                                 -MBER
G(2)     0044H   PUB   FBNODEBASE           G(2)   0046H   PUB   FNODEBASE
G(2)     0040H   PUB   FENODEBASE           G(2)   0040H   PUB   FFNODEBASE
G(2)     0034H   PUB   FIRSTBNODEP          G(2)   0028H   PUB   FIRSTEXNODEP
G(2)     002CH   PUB   FIRSTGRNODEP         G(2)   001CH   PUB   FIRSTNMNODEP
G(2)     0030H   PUB   FIRSTOVNODEP         G(2)   0050H   PUB   FIRSTRENAMEBLOCK
                                                                 -P
G(2)     0020H   PUB   FIRSTSGNODEP         G(2)   0024H   PUB   FIRSTTDNODEP
G(2)     0038H   PUB   GRNODEBASE           G(2)   08A7H   PUB   HIGHESTDATALOCAT
                                                                 -ION

MODULE NAME = ROOT

BASE     OFFSET  TYPE  SYMBOL                BASE   OFFSET  TYPE  SYMBOL

G(2)     0F90H   SYM   MEMORY               G(2)   0000H   SYM   COPYRIGHT
G(2)     0016H   SYM   BUFLEN              G(2)   0018H   SYM   BUFBASE
G(1)     00F7H   SYM   ERROR               G(1)   00FEH   SYM   WARNING
G(2)     001AH   SYM   LASTNMNODEP         G(2)   001CH   SYM   FIRSTNMNODEP
G(2)     001EH   SYM   LASTSGNODEP         G(2)   0020H   SYM   FIRSTSGNODEP
G(2)     0022H   SYM   LASTTDNODEP         G(2)   0024H   SYM   FIRSTTDNODEP
G(2)     0026H   SYM   LASTEXNODEP         G(2)   0028H   SYM   FIRSTEXNODEP
G(2)     002AH   SYM   LASTGRNODEP         G(2)   002CH   SYM   FIRSTGRNODEP
G(2)     002EH   SYM   LASTOVNODEP         G(2)   0030H   SYM   FIRSTOVNODEP
G(2)     0032H   SYM   LASTBNODEP          G(2)   0034H   SYM   FIRSTBNODEP
G(2)     0036H   SYM   SGNODEBASE          G(2)   0038H   SYM   GRNODEBASE
G(2)     003AH   SYM   SYNODEBASE          G(2)   003CH   SYM   NMNODEBASE
G(2)     003EH   SYM   TDNODEBASE          G(2)   0040H   SYM   FFNODEBASE
G(2)     0042H   SYM   OVNODEBASE          G(2)   0044H   SYM   FENODEBASE
G(2)     0046H   SYM   FDNODEBASE          G(2)   0048H   SYM   FANODEBASE
G(2)     004AH   SYM   FBNODEBASE          G(2)   004CH   SYM   BNODEBASE
G(2)     004EH   SYM   CLASHNODEBASE       G(2)   0050H   SYM   FIRSTRENAMEBLOCK
                                                                 -P
```

Figure 2-5. LINK86 Symbol Table

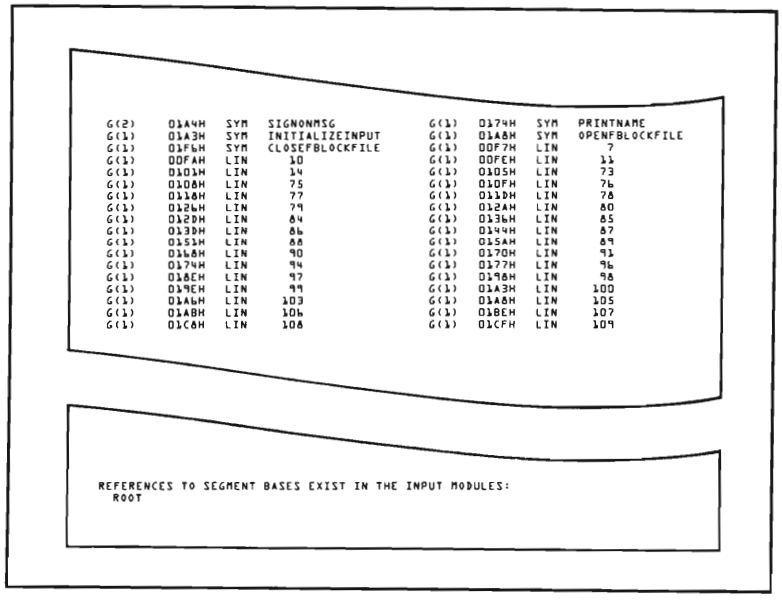


Figure 2-5. LINK86 Symbol Table (Cont'd.)

BASE is usually a symbolic group or segment index. If the base is the stack, then STACK is used instead of the index.

OFFSET is a four-digit hexadecimal number that is the offset of the symbol or line from BASE, or from the current BP for stack symbols.

TYPE describes the kind of symbol it is. There are four possible entries in the TYPE column:

BAS	based on an other symbol's value
LIN	line (not a symbol)
PUB	public symbol (alphabetized within each separate BASE)
SYM	local symbol

SYMBOL refers to the name of the symbol or number of the line. If the SYMBOLCOLUMNS value is one, this field is 40 characters wide. Otherwise, this field is 16 characters wide. If the symbol name is longer than the width of the field, then the name is hyphenated and continued on the next line.

If there are any references to segment bases in the input modules (if the output module is an LTL program), LINK86 prints the following message at the bottom of figure 2-5. The message identifies all input modules containing such references. These references are to be resolved by the system loader or LOC86.

Error Messages

The warning messages are listed consecutively as warning situations are encountered. They may appear before or after the link map. Errors always terminate processing—an error message will always be the last line in the print file.

See the discussion of the interpretation of individual messages in Appendix D.

CREF86 3

CREF86 scans 8086 object modules to provide a cross-reference among external and public symbols in multiple modules. CREF86 accepts a list of files and controls as input and produces one output file: a print file.

Figure 3-1 illustrates the types of input accepted and output produced. The input modules may include one or more of the following 8086 object modules:

- Unlinked modules from one or more translators
- Library files or specific library modules
- Linked modules

The output file consists of information about files and modules, plus an alphabetically sorted list of external and public symbols. Information printed for each symbol includes the name of the module defining the symbol and the name(s) of the module(s) declaring the symbol as external.

This chapter provides details concerning the CREF86 invocation, controls, and cross-reference listing. For definition of file-naming and syntax notation conventions used in this chapter, refer to Notational Conventions following the Preface. For a summary of the CREF86 controls and information on error and warning messages which may be produced, refer to Appendix E. For details concerning CREF86 symbol table space limitations, refer to Appendix C.

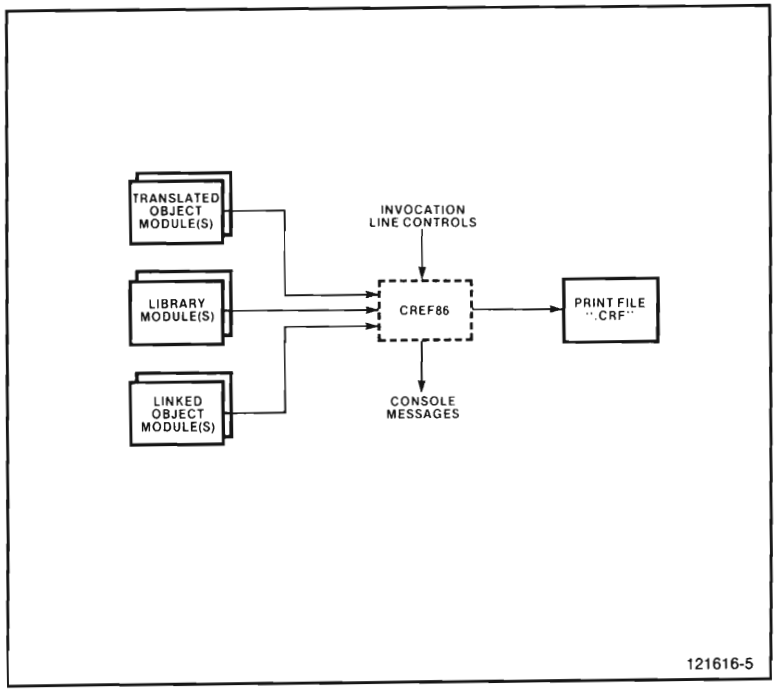


Figure 3-1. CREF86 Input and Output Files

CREF86 Invocation Line

The general syntax for invocation is:

[*directory-name*]CREF86 *input list*[*controls*]

The *input list* is one or more modules to be scanned for external-public cross-references:

pathname[(*module name*[,...])][,...]

Unless a *module name* is specified, all modules in a *pathname* are included in the cross-reference listing produced. If the *pathname* is a library file, any modules named in parentheses are included in the cross-reference listing, even if they do not contain public symbol definitions for external symbols declared elsewhere in the *input list*.

Either all or none of the *pathnames* may contain overlay records (produced by LINK86 with the OVERLAY control). If the input modules do contain overlay records, the first file named in the invocation is considered to be the root file; the rest are treated as overlays.

The *controls* can be any subset of the controls described in the next section.

CREF86 Controls

The *controls* specify cross-reference listing attributes such as print file name, the title at the top of each listing page, and the amount of information printed on each page. The *controls* are described in table 3-1.

Table 3-1. Summary of CREF86 Controls

Control	Abbrev.	Default
PAGELENGTH(*number*)	PL	PAGELENGTH(60)
PAGEWIDTH(*number*)	PW	PAGEWIDTH(120)
PRINT[(*pathname*)]	PR	PRINT(*first input file*.CRF)
TITLE(*character-string*)	TT	Not applicable

If there are multiple occurrences of any control in the invocation line, the rightmost occurrence governs.

See Chapter 7 for operating system information and for operating system-specific examples of the CREF86 controls.

PAGELENGTH

Syntax

PAGELENGTH(*number*)

Abbreviation

PL

Default

PAGELENGTH(60)

Definition

PAGELENGTH specifies the number of lines to be printed on each page. The *number* must be a decimal value between 10 and 255, inclusive.

Notes

None

PAGEWIDTH

Syntax

PAGEWIDTH(*number*)

Abbreviation

PW

Default

PAGEWIDTH(120)

Definition

PAGEWIDTH specifies the maximum number of characters to be printed on a single line. The *number* must be a decimal value from 80 to 132, inclusive.

Notes

- PAGEWIDTH truncates the TITLE if TITLE is greater than the number of unused character locations on the title line.
- If the specified PAGEWIDTH does not allow enough space to print the referring module name(s) on the same line as the defining module name, the referring module names(s) will be printed on separate lines.

PRINT

Syntax

PRINT[(*pathname*)]

Abbreviation

PR

Default

PRINT(*first input file*.CRF)

Definition

PRINT provides the ability to specify a *pathname* for the cross-reference listing. The *pathname* identifies the destination of the listing. If the PRINT control is not specified or if the control is given without an argument, the print file will have the same pathname as the first file in the input list, except the extension will be .CRF.

Notes

- If no PRINT control is specified, output goes to a default file. The name of the default file is the name of the first file in the invocation command with the extension .CRF.
- If PRINT is specified with no *pathname*, output goes to the default file.

TITLE

Syntax

TITLE (*character string*)

Abbreviation

TT

Default

Not applicable

Definition

TITLE may be used to specify a heading having a *character string* of null to 80 characters, inclusive. This heading appears on the first line of every page of the cross-reference listing.

Notes

- The TITLE string is truncated if the PAGEWIDTH control is not large enough to accommodate the entire string.
- If the *character string* contains any characters defined by the operating system as special, the string must be delimited in accord with operating system conventions for special characters and string delimiters.

CREF86's Print File

The print file is a cross-reference listing of external and public symbols in the input modules. This listing consists of the following parts:

- A header
- Warnings (if any)
- Module list
- Cross-reference information

Header

Figure 3-2 illustrates the components printed in this part of the cross-reference listing:

- A title line output by CREF86, usually consisting of a program identifier (CREF86), any user-defined TITLE, date of listing, and the page number
- A line identifying the CREF86 environment (operating system and version number)
- One or more lines summarizing the pathnames of input files
- A line for identifying the print file pathname
- One or more lines giving the controls specified at invocation, present only if controls were specified

Warnings

Figure 3-3 illustrates how warning messages appear on the cross-reference listing when CREF86 detects such conditions as mismatched types, modules not found, etc. Refer to Appendix E for information on CREF86 error and warning messages.

```
CREF86   EXAMPLE OF CROSS REFERENCE USING CREF86                    MM/DD/YY        PAGE

system-id  CREF86  Vx.y

INPUT FILES:   pathname1      pathname2      pathname3      pathname4      pathname5      pathname6
               pathname7      pathname8      pathname9      pathname10     pathname11     pathname12
               pathname13     pathname14     pathname15     pathname16     pathname17
OUTPUT FILE:   pathname18
CONTROLS SPECIFIED:  PR(OUT) TT(EXAMPLE OF CROSS REFERENCE USING CREF86) PW(120) PL(60)
```

Figure 3-2. Header of Cross-Reference Listing

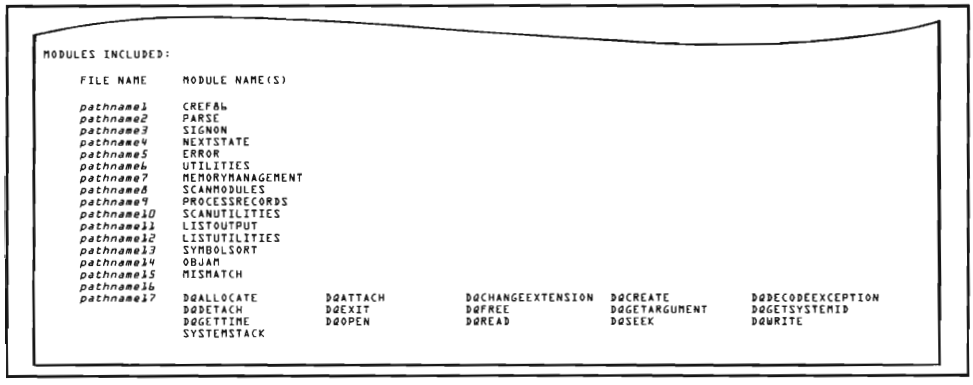
```
WARNING 19:  TYPE MISMATCH
  FILE:  pathname15
  MODULE:  MISMATCH
  SYMBOL:  ENAMEID
WARNING 19:  TYPE MISMATCH
  FILE:  pathname15
  MODULE:  MISMATCH
  SYMBOL:  FOUR
WARNING 20:  SPECIFIED MODULE NOT FOUND
  FILE:  pathname16
  MODULE:  UNKNOWN_MODULE
```

Figure 3-3. Warning Messages on CREF86 Listing

Module List

The module list, shown in figure 3-4, is a tabulated summary of all input files and corresponding modules included from these files.

After the module list is printed, the rest of the page is skipped, so that the symbol cross-references begin on the next new page.



```
MODULES INCLUDED:

     FILE NAME      MODULE NAME(S)

     pathname1      CREF86
     pathname2      PARSE
     pathname3      SIGNON
     pathname4      NEXTSTATE
     pathname5      ERROR
     pathname6      UTILITIES
     pathname7      MEMORYMANAGEMENT
     pathname8      SCANMODULES
     pathname9      PROCESSRECORDS
     pathname10     SCANUTILITIES
     pathname11     LISTOUTPUT
     pathname12     LISTUTILITIES
     pathname13     SYMBOLSORT
     pathname14     OBJAM
     pathname15     MISMATCH
     pathname16
     pathname17     D@ALLOCATE      D@ATTACH        D@CHANGEEXTENSION  D@CREATE        D@DECODEEXCEPTION
                    D@DETACH        D@EXIT          D@FREE             D@GETARGUMENT   D@GETSYSTEMID
                    D@GETTIME       D@OPEN          D@READ             D@SEEK          D@WRITE
                    SYSTEMSTACK
```

Figure 3-4. Module List on CREF86 Listing

Symbol Cross-Reference Information

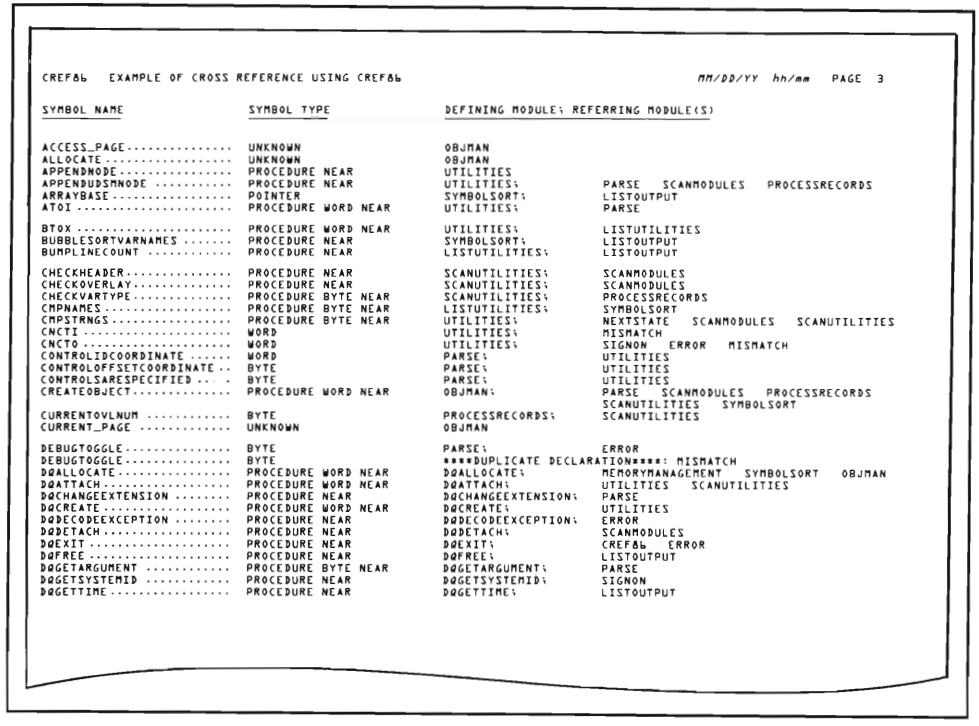
Figure 3-5 illustrates the format for listing data for all external and public symbols referenced in the Module List.

The first column contains the names of the external and public symbols, in alphabetical order.

The second column identifies the type of each symbol, as declared in the external or public reference. The following tabulation identifies the entries which may occur in this column:

CREF86 Entry	Symbol Type
BYTE	8-bit unsigned
WORD	16-bit unsigned
DWORD	32-bit unsigned
LWORD	64-bit unsigned
INTEGER(*n*)	*n* = 1, 2, 4, or 8 bytes
REAL(*n*)	n = 1, 2, 4, or 8 bytes
POINTER	
STRUCTURE	
ARRAY OF	
UNKNOWN	null
FILE	
LABEL	
PROCEDURE (NEAR, FAR)	
CONSTANT	
SELECTOR	

The symbol type that appears in the second column is that associated with the first occurrence of that symbol in the input list.



```
CREF86  EXAMPLE OF CROSS REFERENCE USING CREF86                    MM/DD/YY  hh/mm   PAGE  3

SYMBOL NAME                    SYMBOL TYPE                 DEFINING MODULE; REFERRING MODULE(S)


ACCESS_PAGE.................   UNKNOWN                     OBJMAN
ALLOCATE....................   UNKNOWN                     OBJMAN
APPENDNODE..................   PROCEDURE NEAR              UTILITIES
APPENDUDSMNODE .............   PROCEDURE NEAR              UTILITIES;          PARSE   SCANMODULES   PROCESSRECORDS
ARRAYBASE...................   POINTER                     SYMBOLSORT;         LISTOUTPUT
ATOI .......................   PROCEDURE WORD NEAR         UTILITIES;          PARSE

BTOX .......................   PROCEDURE WORD NEAR         UTILITIES;          LISTUTILITIES
BUBBLESORTVARNAMES .........   PROCEDURE NEAR              SYMBOLSORT;         LISTOUTPUT
BUMPLINECOUNT ..............   PROCEDURE NEAR              LISTUTILITIES;      LISTOUTPUT

CHECKHEADER.................   PROCEDURE NEAR              SCANUTILITIES;      SCANMODULES
CHECKOVERLAY................   PROCEDURE NEAR              SCANUTILITIES;      SCANMODULES
CHECKVARTYPE................   PROCEDURE BYTE NEAR         SCANUTILITIES;      PROCESSRECORDS
CMPNAMES....................   PROCEDURE BYTE NEAR         LISTUTILITIES;      SYMBOLSORT
CMPSTRNGS...................   PROCEDURE BYTE NEAR         UTILITIES;          NEXTSTATE   SCANMODULES   SCANUTILITIES
CNCTI ......................   WORD                        UTILITIES;          MISMATCH
CNCTO ......................   WORD                        UTILITIES;          SIGNON   ERROR   MISMATCH
CONTROLIDCOORDINATE ........   WORD                        PARSE;              UTILITIES
CONTROLOFFSETCOORDINATE ....   BYTE                        PARSE;              UTILITIES
CONTROLSARESPECIFIED .......   BYTE                        PARSE;              UTILITIES
CREATEOBJECT................   PROCEDURE WORD NEAR         OBJMAN;             PARSE   SCANMODULES   PROCESSRECORDS
                                                                               SCANUTILITIES   SYMBOLSORT
CURRENTOVLNUM ..............   BYTE                        PROCESSRECORDS;     SCANUTILITIES
CURRENT_PAGE ...............   UNKNOWN                     OBJMAN

DEBUGTOGGLE.................   BYTE                        PARSE;              ERROR
DEBUGTOGGLE.................   BYTE                        ****DUPLICATE DECLARATION****: MISMATCH
D@ALLOCATE..................   PROCEDURE WORD NEAR         D@ALLOCATE;         MEMORYMANAGEMENT   SYMBOLSORT   OBJMAN
D@ATTACH....................   PROCEDURE WORD NEAR         D@ATTACH;           UTILITIES   SCANUTILITIES
D@CHANGEEXTENSION ..........   PROCEDURE NEAR              D@CHANGEEXTENSION;  PARSE
D@CREATE....................   PROCEDURE WORD NEAR         D@CREATE;           UTILITIES
D@DECODEEXCEPTION ..........   PROCEDURE NEAR              D@DECODEEXCEPTION;  ERROR
D@DETACH....................   PROCEDURE NEAR              D@DETACH;           SCANMODULES
D@EXIT......................   PROCEDURE NEAR              D@EXIT;             CREF86   ERROR
D@FREE......................   PROCEDURE NEAR              D@FREE;             LISTOUTPUT
D@GETARGUMENT ..............   PROCEDURE BYTE NEAR         D@GETARGUMENT;      PARSE
D@GETSYSTEMID ..............   PROCEDURE NEAR              D@GETSYSTEMID;      SIGNON
D@GETTIME...................   PROCEDURE NEAR              D@GETTIME;          LISTOUTPUT
```

Figure 3-5. Symbol Cross-Reference Information

The third column contains the following for each symbols listed:

- The name of the module in which the symbol is defined public (defining module)
- A semicolon (;), if there are external references to the symbol in any of the input modules
- The name(s) of the modules(s) in which the symbol is declared external (referring module(s))

The third column is also used to flag unresolved and duplicate references. In the case of unresolved external references, the string ***UNRESOLVED*** appears before the semicolon. In the case of duplicate references, i.e., when a symbol has two or more public definitions, the first public declaration is considered legal, and the rest are flagged as duplicates. The string ***DUPLICATE DECLARATION*** appears, followed by a colon (:) and the name of the module containing the duplicate public declaration.

If the input files contain overlays, CREF86 produces a symbol cross-reference that consolidates all the symbols from all overlay and root modules. The first file in the input list is considered to be the root file. CREF86 distinguishes between public symbols with the same name in different overlays and does not flag these symbols as duplicates. However, CREF86 does flag duplicate public declarations within any one root/overlay combination.

LIB86 4

LIB86 allows you to create, modify, and examine library files. It is an interactive program.

This chapter provides details concerning LIB86 invocation and commands. For definition of file-naming and syntax notation conventions used in this chapter, refer to Notational Conventions following the Preface. For a summary of the LIB86 commands and information on error and warning messages that may be produced, refer to Appendix F. For details concerning LIB86 symbol table space limitations, refer to Appendix C.

LIB86 Invocation

The general syntax for the invocation line is:

[*directory-name*]LIB86[*comment*]

LIB86 Commands

Once LIB86 has begun execution, it displays an asterisk (*) and waits for a command. Table 4-1 lists all of LIB86's commands.

Table 4-1. Summary of LIB86 Commands

Command	Abbrev.	Description
ADD {*pathname*[(*module name* [,...])]} [,...] TO *pathname*	A	Adds modules to a library
CREATE *pathname*	C	Creates library files
DELETE *pathname*(*module name* [,...])	D	Deletes modules from a library file
EXIT	E	Terminates session with LIB86
LIST {*pathname*[(*module name* [,...])]} [,...] [TO *pathname*] [PUBLICS]	L [P]	Lists modules contained in a library file, and optionally lists all publics

See Chapter 7 for operating system information and for operating system-specific examples of the LIB86 commands.

ADD

Syntax

ADD {*pathname1*[(*module name*[,...])][,...]TO *pathname2*

Abbreviation

A

Definition

ADD adds modules to a library file.

The *pathname1* can be an object file or a library file.

The *pathname2* is the destination library file. The library must exist before the ADD command is given; it may contain other modules.

If *pathname1* is an object file produced by a translator, LINK86, or LOC86, then all modules contained within the object file will be added to the designated library.

If *pathname2* is a library file, it may be specified with or without the *module name* list. If no *module name* list is specified, all modules contained in the source library will be added to the destination library. If the module name list is specified, then only the modules specified within the parentheses are added to the destination library.

CREATE

Syntax

CREATE *pathname*

Abbreviation

C

Definition

CREATE creates a library file with the specified *pathname*.

Notes

- If a file with the specified *pathname* already exists, the library will not be created and an error message will be provided.

DELETE

Syntax

DELETE *pathname*(*module name* [,...])

Abbreviation

D

Definition

DELETE removes modules from a library file. Modules can be deleted from only one library at a time.

Notes

None

EXIT

Syntax

```
EXIT
```

Abbreviation

```
E
```

Definition

EXIT terminates a session with LIB86 and returns control to the operating system.

Notes

- LIB86 disassembles libraries into an internal form. The library is not reconstituted until the EXIT command is processed. Therefore significant I/O will take place following an EXIT command.

LIST

Syntax

```
LIST {pathname1[(module name [,...])]} [,...][TO pathname2]
     [PUBLICS]
```

Abbreviation

```
L [P]
```

Definition

LIST prints the names of modules, and optionally the public symbols contained in those modules, to the specified output pathname.

The *pathname1* is the library whose modules are to be listed.

The *module name*, if specified, identifies the modules to be listed.

TO *pathname2* identifies the device or file to receive the listing. If it is not specified, the listing is directed to the console output device.

PUBLICS indicates that, in addition to the module names, all public symbols contained within the module will also be listed. PUBLICS may be abbreviated as 'P'.

Notes

None

LOC86 5

LOC86 changes a relocatable 8086 object module into an absolute object module. As figure 5-1 illustrates, LOC86 takes a single 8086 object module as input and outputs a located object file and, optionally, a print file. The print file output contains diagnostic information. The object file contains absolute object code.

This chapter provides details concerning the LOC86 invocation, controls, and print file. For definition of file-naming and syntax notation conventions used in this chapter, refer to Notational Conventions following the Preface. For a summary of the LOC86 controls and information on error and warning messages that may be produced, refer to Appendix G. For details concerning LOC86 segment support capabilities, refer to Appendix C.

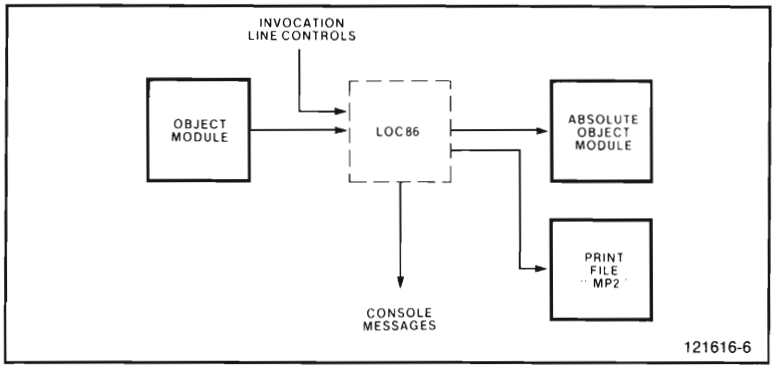


Figure 5-1. LOC86 Input and Output Files

LOC86 Invocation Line

The general syntax for the invocation line is:

[*directory-name*]`LOC86` *input file* [`TO` *object file*][*controls*]

The *input file* is a file containing an object module to be located. It is usually, but not necessarily, the output from LINK86.

TO *object file* specifies the file to receive the located object module. In most cases this is an executable file. If *object file* is not specified, then output will be directed to a file that has the same pathname as the *input file*, except it will have no extension.

The *controls* may be any subset of the controls described in the next section.

LOC86 Controls

The *controls* are described in table 5-1.

If you specify the same control more than once in the same invocation line, only the last version entered counts. For example, if you enter NOMAP, and then later decide you want a locate map, you can enter the MAP control without error. The second version of the control is recognized and the first is ignored.

See Chapter 7 for operating system information and for operating system-specific examples of the LOC86 controls.

Table 5-1. Summary of LOC86 Controls

Control	Abbrev.	Default
ADDRESSES({SEGMENTS({*segment*|*class*|*overlay*|| (*addr*)}|,...|) | CLASSES({*class*(*addr*)}|,...|) | GROUPS({*group*(*addr*)}|,...|) } |,...|)	AD (SM | CS | GR)	Not applicable
BOOTSTRAP	BS	Not applicable
COMMENTS	CM	COMMENTS
NOCOMMENTS	NOCM	
INITCODE|(*address*)|	IC	INITCODE(200H)
NOINITCODE	NOIC	
LINES	LI	LINES
NOLINES	NOLI	
MAP	MA	MAP
NOMAP	NOMA	
NAME(*module*)	NA	Not applicable
OBJECTCONTROLS({LINES | NOLINES | COMMENTS | NOCOMMENTS | SYMBOLS | NOSYMBOLS | PUBLICS | NOPUBLICS | PURGE | NOPURGE}|,...|)	OC	Not applicable
ORDER({SEGMENTS({*segment*| *class*|\ *overlay*||} |,...|) | CLASSES({*class*|(*segment* |,...|)|} |,...|)} |,...|)	OD (SM | CS) CS	Not applicable
PRINT|(*pathname*)|	PR	PRINT(*object file*.MP2)
NOPRINT	NOPR	
PRINTCONTROLS({LINES | NOLINES | COMMENTS | NOCOMMENTS | SYMBOLS | NOSYMBOLS | PUBLICS | NOPUBLICS | PURGE | NOPURGE}|,...|)	PC	Not applicable
PUBLICS	PL	PUBLICS
NOPUBLICS	NOPL	
PURGE	PU	NOPURGE
NOPURGE	NOPU	
RESERVE({*addr* TO *addr*} |,...|)	RS	Not applicable
SEGSIZE({*segment*|*class*|*overlay*|| (*size*)} |,...|)	SS	Not applicable
START({*symbol* |*paragraph*,*offset*})	ST	Not applicable
SYMBOLS	SB	SYMBOLS
NOSYMBOLS	NOSB	
SYMBOLCOLUMNS({1|2|3|4})	SC	SYMBOLCOLUMNS(2)

ADDRESSES

Syntax

```
ADDRESSES({SEGMENTS({segment name[\class name
                                 [\overlay name]](address)}
                    [,...])|
          CLASSES({class name(address)}[,...])|
          GROUPS({group name(address)}[,...])}
         [,...]
        )
```

Abbreviation

```
AD(SM|CS|GR)
```

Default

Not applicable

Definition

ADDRESSES allows you to override LOC86's default address assignment algorithm. You may assign a beginning address to segments, classes, or groups. All addresses must follow Intel rules for integer representation. (These rules are the same as those used by ASM86 and PL/M-86.) The subcontrols, SEGMENTS, CLASSES, and GROUPS, identify exactly what elements of the input module are being assigned addresses. When assigning an address with the SEGMENTS subcontrol, you may also specify the class name and overlay name of the particular segment.

LOC86 attempts to detect and avoid conflicts whenever possible. If the specified address does not agree with the alignment attribute of the specified segment or the first segment in the specified class, then the address is ignored. If an absolute segment is located at the address assigned to a class, then the class begins at the first free address after the absolute segment. If you assign a non-paragraph address to a group, LOC86 will assign the first paragraph address *below* the specified address.

Notes

- The subcontrols SEGMENTS, CLASSES, and GROUPS can be specified multiple times in a single ADDRESSES control.
- If an address assignment causes a conflict with an ORDER control, a RESERVE control or an absolute segment, LOC86 generates an error message.
- When locating bound object modules, you may not assign an address to a segment in a group.
- Segments of length 0 are ignored during address assignments.
- The address assignment for a GROUP does not affect the addresses assigned to segments in that GROUP.

BOOTSTRAP

Syntax

BOOTSTRAP

Abbreviation

BS

Default

Not applicable

Definition

BOOTSTRAP indicates that the code for a long jump to the module's start address should be placed at location 0FFFF0H, when the module is loaded. This is the first instruction executed by the 8086 after reset. If the input module has no start address and none is specified in the START control, LOC86 will generate an error message.

Notes

- See also the START and INITCODE controls.

COMMENTS/NOCOMMENTS

Syntax

```
COMMENTS
NOCOMMENTS
```

Abbreviation

```
CM
NOCM
```

Default

```
COMMENTS
```

Definition

COMMENTS allows object file comment records to remain in the output module. The NOCOMMENTS control removes all comment records except those designated as non-purgable.

Comment records are added to the object module for various reasons. All translators add a comment record to the object files they produce. The record identifies the compiler or assembler that produced the object file.

Comment records are superfluous to the production of executable code and may be removed at any time during the development process, or left in the file.

Notes

- See PRINTCONTROLS, OBJECTCONTROLS, and PURGE.
- Comment records should not be removed when you submit an object file in a Software Problem Report.
- NOCOMMENTS will decrease the size of the output object module.
- COMMENTS has no effect on the print file.

INITCODE/NOINITCODE

Syntax

```
INITCODE[(address)]
NOINITCODE
```

Abbreviation

```
IC
NOIC
```

Default

```
INITCODE(200H)
```

Definition

INITCODE causes LOC86 to create a new segment that contains code to initialize the segment registers. The optional *address* argument specifies the physical address of the code that performs this initialization. If no *address* is specified, the initialization code will be placed at 200H. The equivalent assembly language code is shown below:

```
STACKFRAME   DW    stack frame
DATAFRAME    DW    data frame
EXTRAFRAME   DW    extra frame
             CLI
             MOV   SS, CS:STACKFRAME
             MOV   SP, stack offset
             MOV   DS, CS:DATAFRAME
             MOV   ES, CS:EXTRAFRAME
             JMP   program start
```

Notes

- The initialization code segment is created only if a register initialization record for 8086 segment registers exists in the input. These register initialization records are automatically produced by 8086-based translators for main modules.
- If the area of memory used by the INITCODE default is reserved, LOC86 places the initialization code above the reserved space.
- If created, the new segment is called ??LOC86_INITCODE.

LINES/NOLINES

Syntax

```
LINES
NOLINES
```

Abbreviation

```
LI
NOLI
```

Default

```
LINES
```

Definition

LINES allows line number information to remain in the object file. In-circuit emulators and other debuggers use this information; it is not needed to produce executable code. The NOLINES control removes this information from the output file.

Notes

- The scope of the LINES control can be modified with PRINTCONTROLS and OBJECTCONTROLS.
- See the PURGE control.
- NOLINES will decrease the size of the output object file; however, this information is used by debuggers.

MAP/NOMAP

Syntax

```
MAP
NOMAP
```

Abbreviation

```
MA
NOMA
```

Default

```
MAP
```

Definition

MAP causes LOC86 to produce a locate map for the output module and add it to the print file. For all segments in the module the map shows the complete name (segment name, class name, and overlay name), size, alignment, start address, and stop address. A more complete description of the locate map and the rest of the print file is at the end of this chapter.

Notes

- MAP can be overridden by the NOPRINT control.

NAME

Syntax

NAME (*module name*)

Abbreviation

NA

Default

Module retains its current name.

Definition

NAME assigns the specified *module name* to the output module. If NAME is not specified, then the output module retains its current name.

The *module name* may be up to 40 characters long. It may be composed of any of the following characters in any order:

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

Lower case letters may be used, but they are automatically converted to upper case.

Notes

- NAME does not affect the output file's name, only the module name in the output module's header record.

OBJECTCONTROLS

Syntax

```
OBJECTCONTROLS({LINES | NOLINES |
                COMMENTS | NOCOMMENTS |
                SYMBOLS | NOSYMBOLS |
                PUBLICS | NOPUBLICS |
                PURGE | NOPURGE}
                [,...]
               )
```

Abbreviation

OC

Default

Controls apply to both the print file and the object file.

Definition

OBJECTCONTROLS causes the controls specified in its arguments to be applied to the object file only. Comment records, line number records, local and public symbol records, and symbol type records are selectively included or excluded from the object file. This will not affect the print file and the information contained in it.

Notes

- If you specify an invalid control in the arguments to OBJECTCONTROLS, LOC86 generates an error message.
- You may specify a control or control pair more than once within OBJECTCONTROLS, but only the last version specified counts.
- You may abbreviate the controls used within OBJECTCONTROLS.
- When you specify a control in both OBJECTCONTROLS and PRINTCONTROLS, it will have the same effect as specifying it once outside of these controls.

ORDER

Syntax

```
ORDER({SEGMENTS({segment name[\class name[\overlay name]]}
                [,...]),|
       CLASSES ({class name[(segment name[,...])]}[,...])}
      [,...])
```

Abbreviation

```
OD(SM|CS|)
```

Default

Not applicable

Definition

ORDER specifies a partial or complete order for segments, classes, and the segments within a class. Segments and classes listed in ORDER are located before any other relocatable segment.

The subcontrol SEGMENTS indicates that the list of segment names shall be ordered.

The *segment name* identifies the specific segments to be ordered. The *\class name* and *\overlay name* may be used to resolve conflicts with duplicate segment names. If *\overlay name* is specified, the *\class name* is required.

If one of the segments specified is not contained in the designated group, an error message is generated.

Notes

- See "LOC86's Algorithm for Locating Segments" at the end of this chapter.

PRINT/NOPRINT

Syntax

```
PRINT[(pathname)]
NOPRINT
```

Abbreviation

```
PR
NOPR
```

Default

PRINT(*object file*.MP2)

Definition

PRINT allows you to direct the locate map symbol table and other diagnostic information to a particula. file. If the PRINT control is not specified or if the control is given without an argument, the print file will have the same pathname as the output file except the extension will be .MP2. NOPRINT prevents the creation of this file.

Notes

- The discussion at the end of this chapter describes the contents of the print file.
- See also MAP, SYMBOLCOLUMNS, LINES, SYMBOLS, PUBLICS, and PRINTCONTROLS.

PRINTCONTROLS

Syntax

```
PRINTCONTROLS({LINES | NOLINES |
               COMMENTS | NOCOMMENTS |
               SYMBOLS | NOSYMBOLS |
               PUBLICS | NOPUBLICS |
               PURGE | NOPURGE}
               [,...]
              )
```

Abbreviation

PC

Default

Controls apply to both the print file and the object file.

Definition

PRINTCONTROLS causes the controls specified in its arguments to be applied to the print file only. Line number information, and local and public symbol information are selectively included or excluded from the print file. This will not affect the object file or the information contained in it.

Notes

- If you specify an invalid control in the arguments to PRINTCONTROLS, LOC86 generates an error message.
- You may specify a control in OBJECTCONTROLS more than once, but only the last version specified counts.
- You may abbreviate the controls used within PRINTCONTROLS.
- When you specify a control in both PRINTCONTROLS and OBJECTCONTROLS, it will have the same effect as specifying it once outside of these controls.

PUBLICS/NOPUBLICS

Syntax

```
PUBLICS
NOPUBLICS
```

Abbreviation

```
PL
NOPL
```

Default

```
PUBLICS
```

Definition

PUBLICS causes the public symbol records to be kept in the object file and the corresponding information to be placed in the print file.

Notes

- The scope of PUBLICS can be modified by PRINTCONTROLS and OBJECTCONTROLS.
- NOPUBLICS will reduce the size of the output object file; however, public symbol records may be used by debuggers.
- See the PURGE control.

PURGE/NOPURGE

Syntax

```
PURGE
NOPURGE
```

Abbreviation

```
PU
NOPU
```

Default

```
NOPURGE
```

Definition

PURGE is exactly the same as specifying NOLINES, NOSYMBOLS, NOCOMMENTS, and NOPUBLICS. NOPURGE in the control list is the same as specifying LINES, SYMBOLS, COMMENTS, and PUBLICS.

PURGE removes all of the public and debug information from the object file and the print file. It will produce the most compact object file possible. The records that would be included by NOPURGE are useful to debuggers and in-circuit emulators, but otherwise they are unnecessary for producing executable code.

Notes

- PRINTCONTROLS and OBJECTCONTROLS can be used to modify the scope of PURGE.

RESERVE

Syntax

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

Abbreviation

RS

Default

All of memory is assumed available.

Definition

RESERVE prevents LOC86 from locating segments in certain areas of memory. LOC86 will not use all memory addresses from *address1* to *address2* inclusive; *address1* must be less than or equal to *address2*.

Notes

- If an absolute segment uses a reserved memory area, a warning message is generated.
- Reserved areas may overlap.

SEGSIZE

Syntax

SEGSIZE({*segment name*[*class name*[*overlay name*]] (*size*)}
[,...])

Abbreviation

SS

Default

Not applicable

Definition

SEGSIZE allows you to specify the memory space used by a segment.

The *segment name* may be any segment contained in the input module.

The *size* is a 16-bit number that LOC86 uses to change the size of the specified segment. There are three ways of specifying this value.

- + indicates that the number should be added to the current segment length.
- — indicates that the number should be subtracted from the current segment length.
- No sign indicates that the number should become the new segment length.

Notes

- LOC86 issues a warning message when SEGSIZE decreases the size of a segment.

START

Syntax

START({*public symbol* | *paragraph*, *offset*})

Abbreviation

ST

Default

The start address designated in the input module

Definition

START allows you to specify the start address of your program.

If you specify *public symbol*, that symbol must be defined within the input module.

The *paragraph* value initializes the CS register and the *offset* value initializes the IP in an 8086 long jump when your program is started.

Notes

- See the BOOTSTRAP and INITCODE controls.

SYMBOLS/NOSYMBOLS

Syntax

```
SYMBOLS
NOSYMBOLS
```

Abbreviation

```
SB
NOSB
```

Default

```
SYMBOLS
```

Definition

SYMBOLS specifies that all local symbol records shall be included in the object file, and information concerning local symbols will also appear in the symbol table contained in the print file. Local symbol records are used by debuggers and in-circuit emulators.

Notes

- The scope can be modified by OBJECTIONCONTROLS and PRINTCONTROLS.
- NOSYMBOLS will decrease the size of the output object file; however, this information is used by debuggers.
- See the PURGE control.

SYMBOLCOLUMNS

Syntax

```
SYMBOLCOLUMNS({1|2|3|4})
```

Abbreviation

```
SC
```

Default

```
SYMBOLCOLUMNS(2)
```

Definition

SYMBOLCOLUMNS indicates the number of columns to be used when producing the symbol table for the object module. Two columns fit on a 78-character line; four columns fit on a single 128-character line printer line.

Notes

None

LOC86's Print File

The print file is always created unless you specify NOPRINT. The optional argument to PRINT designates the name of the print file. The default print file is the object file with the extension .MP2.

The print file may contain as many as three parts:

- A symbol table
- A memory map
- An error message list

The symbol table is included in the print file when a PUBLICS, LINES, or SYMBOLS control is in effect. The memory map is controlled by the MAP/NOMAP control. Error and warning messages, if any, are always added to the print file.

The Symbol Table

LOC86 produces a symbol table when any or all of the symbol controls (LINES, SYMBOLS, and PUBLICS) are in effect. No symbol table will be produced when PURGE is in effect for the print file.

Figure 5-2 shows LOC86's symbol table with the SYMBOLCOLUMNS set at 2 (the default).

BASE is usually a 4-digit hexadecimal number that is the base address of the group that contains the symbol. If the base is the stack, then STACK is used instead of a number. If the symbol is based on another symbol's value, then the BASE and OFFSET values for that symbol are given.

OFFSET is a 4-digit hexadecimal number that is the offset of the symbol or line from BASE, or from the current BP for stack symbols.

To compute the physical address of the specified symbol you would use the following equation:

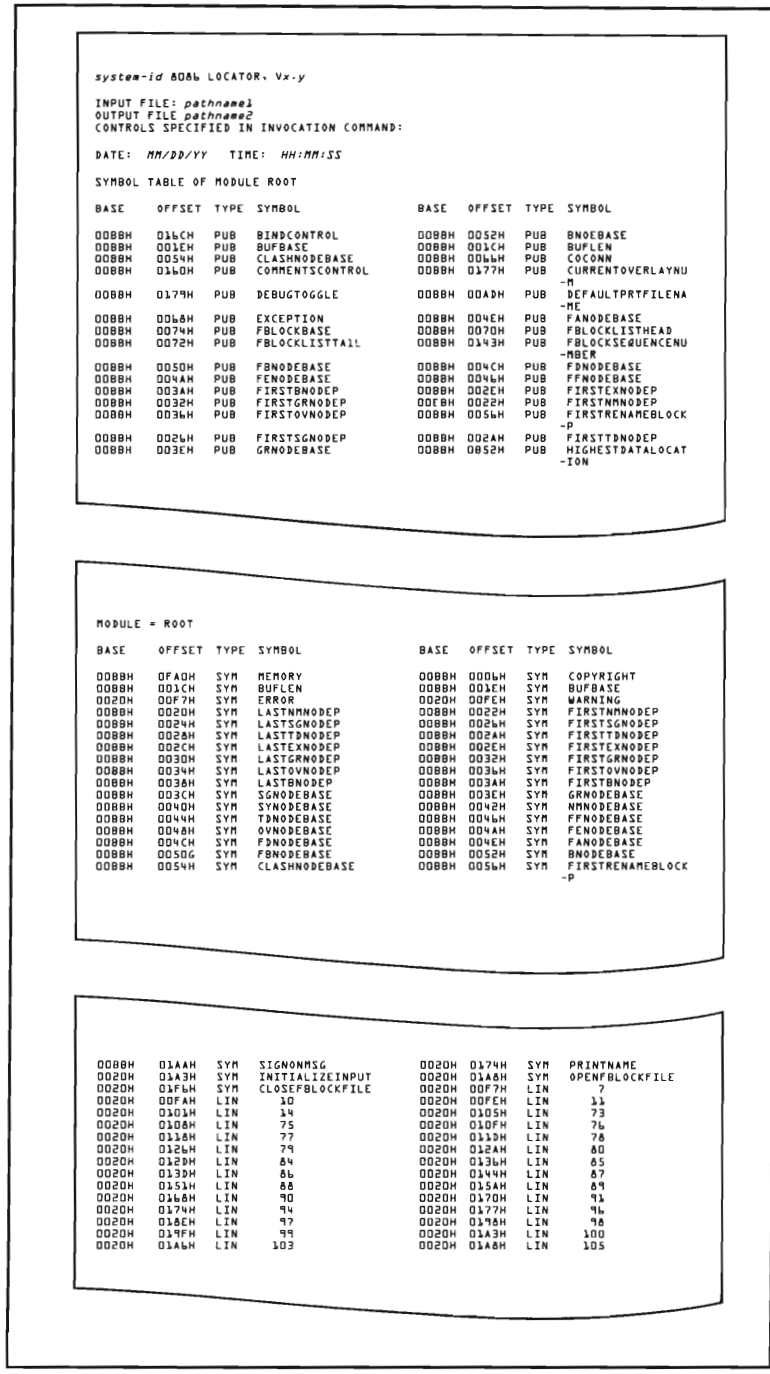
(BASE * 10H) + OFFSET = Physical Address

Of course, the physical address of the symbols whose base is the STACK, or symbols that are based on another symbol's value, cannot be computed until run-time.

TYPE describes the kind of symbol it is. There are four possible entries in the TYPE column:

BAS	based on another symbol's value
LIN	line (not a symbol)
PUB	public symbol
SYM	local symbol

SYMBOL field contains the name of the symbol or number of the line. If the SYMBOLCOLUMNS value is 1, this field is 40 characters wide. If the SYMBOLCOLUMNS value is 2 or more, then this field is 16 characters wide. If the symbol name is longer than the width of the entry, then the name is hyphenated and continued in the SYMBOL field on the next line.



```
system-id 8086 LOCATOR, Vx.y

INPUT FILE: pathname1
OUTPUT FILE pathname2
CONTROLS SPECIFIED IN INVOCATION COMMAND:

DATE:  MM/DD/YY    TIME:  HH:MM:SS

SYMBOL TABLE OF MODULE ROOT

BASE     OFFSET  TYPE  SYMBOL                BASE     OFFSET  TYPE  SYMBOL

0088H    016CH   PUB   BINDCONTROL           0088H    0052H   PUB   BNODEBASE
0088H    001EH   PUB   BUFBASE               0088H    001CH   PUB   BUFLEN
0088H    0054H   PUB   CLASHNODEBASE         0088H    0066H   PUB   COCONN
0088H    016DH   PUB   COMMENTSCONTROL       0088H    0177H   PUB   CURRENTOVERLAYNU
                                                                     -M
0088H    0179H   PUB   DEBUGTOGGLE          0088H    00ADH   PUB   DEFAULTPRTFILENA
                                                                     -ME
0088H    0068H   PUB   EXCEPTION             0088H    004EH   PUB   FANODEBASE
0088H    0074H   PUB   FBLOCKBASE            0088H    0070H   PUB   FBLOCKLISTHEAD
0088H    0072H   PUB   FBLOCKLISTTAIL        0088H    0143H   PUB   FBLOCKSEQUENCENU
                                                                     -MBER
0088H    0050H   PUB   FBNODEBASE            0088H    004CH   PUB   FDNODEBASE
0088H    004AH   PUB   FENODEBASE            0088H    0046H   PUB   FFNODEBASE
0088H    003AH   PUB   FIRSTBNODEP           0088H    002EH   PUB   FIRSTEXNODEP
0088H    0032H   PUB   FIRSTGRNODEP          00F8H    0022H   PUB   FIRSTNMNODEP
0088H    003EH   PUB   FIRSTOVNODEP          0088H    0056H   PUB   FIRSTRENAMEBLOCK
                                                                     -P
0088H    0026H   PUB   FIRSTSGNODEP          0088H    002AH   PUB   FIRSTTDNODEP
0088H    003EH   PUB   GRNODEBASE            0088H    0852H   PUB   HIGHESTDATALOCAT
                                                                     -ION


MODULE = ROOT

BASE     OFFSET  TYPE  SYMBOL                BASE     OFFSET  TYPE  SYMBOL

0088H    0FA0H   SYM   MEMORY                0088H    0006H   SYM   COPYRIGHT
0088H    001CH   SYM   BUFLEN                0088H    001EH   SYM   BUFBASE
0020H    00F7H   SYM   ERROR                 0020H    00FEH   SYM   WARNING
0088H    0020H   SYM   LASTNMNODEP           0088H    0022H   SYM   FIRSTNMNODEP
0088H    002AH   SYM   LASTSGNODEP           0088H    0026H   SYM   FIRSTSGNODEP
0088H    002AH   SYM   LASTTDNODEP           0088H    002AH   SYM   FIRSTTDNODEP
0088H    002EH   SYM   LASTEXNODEP           0088H    002EH   SYM   FIRSTEXNODEP
0088H    0032H   SYM   LASTGRNODEP           0088H    0032H   SYM   FIRSTGRNODEP
0088H    0036H   SYM   LASTOVNODEP           0088H    0036H   SYM   FIRSTOVNODEP
0088H    003AH   SYM   LASTBNODEP           0088H    003AH   SYM   FIRSTBNODEP
0088H    003EH   SYM   SGNODEBASE           0088H    003EH   SYM   GRNODEBASE
0088H    0040H   SYM   SYNODEBASE           0088H    0042H   SYM   NMNODEBASE
0088H    0044H   SYM   TDNODEBASE           0088H    0046H   SYM   FFNODEBASE
0088H    0048H   SYM   OVNODEBASE           0088H    004AH   SYM   FENODEBASE
0088H    004CH   SYM   FDNODEBASE           0088H    004EH   SYM   FANODEBASE
0088H    0050G   SYM   FBNODEBASE           0088H    0052H   SYM   BNODEBASE
0088H    0054H   SYM   CLASHNODEBASE        0088H    0056H   SYM   FIRSTRENAMEBLOCK
                                                                     -P


0088H    01AAH   SYM   SIGNONMSG            0020H    0174H   SYM   PRINTNAME
0020H    01A3H   SYM   INITIALIZEINPUT      0020H    01A8H   SYM   OPENFBLOCKFILE
0020H    01F6H   SYM   CLOSEFBLOCKFILE      0020H    00F7H   LIN      7
0020H    00FAH   LIN      10                0020H    00FEH   LIN     11
0020H    0101H   LIN      14                0020H    0105H   LIN     73
0020H    0108H   LIN      75                0020H    010FH   LIN     76
0020H    0118H   LIN      77                0020H    011DH   LIN     78
0020H    0126H   LIN      79                0020H    012AH   LIN     80
0020H    012DH   LIN      84                0020H    0136H   LIN     85
0020H    013DH   LIN      86                0020H    0144H   LIN     87
0020H    0151H   LIN      88                0020H    015AH   LIN     89
0020H    0168H   LIN      90                0020H    0170H   LIN     91
0020H    0174H   LIN      94                0020H    0177H   LIN     96
0020H    018EH   LIN      97                0020H    0198H   LIN     98
0020H    019FH   LIN      99                0020H    01A3H   LIN    100
0020H    01A6H   LIN     103                0020H    01A8H   LIN    105
```

Figure 5-2. LOC86 Symbol Table

The Memory Map

The memory map supplies useful information about segment placement and address assignment. Figure 5-3 shows LOC86's memory map.

The map consists of three parts:

- Header
- Segment map
- Group map

The header includes the input module name and the start address.

The segment map is a table with six columns. From left to right the columns show:

- the START address of the segment
- the STOP address of the segment
- the LENGTH of the segment
- the ALIGNMENT attribute of the segment
- the NAME of the segment
- the CLASS of the segment
- the OVERLAY of the segment

A "C" printed between the STOP and LENGTH columns indicates that two segments have overlapping memory locations; a warning message is also issued.

A segment may have any one of the following alignment attributes:

A	absolute
B	byte
G	paragraph
M	member of an LTL group
P	page
W	word
R	in-page

```
MEMORY MAP OF MODULE ROOT

MODULE START ADDRESS PARAGRAPH = 0020H OFFSET = 0002H
SEGMENT MAP

START     STOP     LENGTH ALIGN NAME       CLASS        OVERLAY

0020DH    00BB5H   09B6H   W   CODE       CODE
00BB6H    00BC6H   0016H   W   CONST      CONST
00BCCH    01703H   0B38H   W   DATA       DATA
01704H    01B45H   0442H   W   STACK      STACK
01B50H    01B50H   0000H   G   ??SEG
01B50H    01B50H   0000H   W   MEMORY     MEMORY



GROUP MAP

ADDRESS   GROUP OR SEGMENT NAME
0020DH    CGROUP
            CODE
00BB0H    DGROUP
            CONST
            DATA
            STACK
            MEMORY
```

Figure 5-3. LOC86 Memory Map

In-page alignment means that the entire segment must be resident within a single 256-byte page. The address of the first byte in any page has zeros in the first two hexadecimal digits (00H, 100H, 200H,... 0FFF00H).

The group map has two columns: the first is the physical address (five-digit hexadecimal number) of the beginning of the group; the second column is the group name, followed by the segments contained in that group. The segment names contained within a given group are listed in the same column but indented slightly.

Error and Warning Messages

The error and warning messages are listed consecutively as the error situations are encountered.

See the discussion of the interpretation of individual messages in Appendix F.

LOC86's Algorithm for Locating Segments

Assuming that there are no errors in the invocation line or input module, LOC86 locates an input module in three stages.

1. All absolute segments are removed from the list of segments contained in the module.
2. The remaining relocatable segments are ordered into a sequential list.
3. The relocatable segments are then given absolute addresses according to each segment's alignment, size, and memory attribute.

Absolute Segments

When LOC86 encounters an absolute segment, LOC86 removes the segment from a list of input segments and reserves the memory area used by that segment. LOC86 maintains a map of free memory. Each time an absolute segment is encountered, the memory space used by that segment is removed from the memory map. A segment can become absolute in one of three ways:

1. It may be assigned an absolute address by the translator.
2. It may be explicitly specified in an ADDRESSES control.
3. It may become absolute implicitly. If an absolute segment is specified in an ORDER control, then all other segments referred to in that control, either by segment name or by class name, are treated as absolute.

Segment Ordering

After all memory used by absolute segments has been removed from LOC86's free memory map and before LOC86 begins assigning addresses to the remaining relocatable segments, LOC86 prepares an ordered list of all relocatable segments.

All relocatable segments specified in an ORDER control are placed at the head of the list.

After all ORDER controls, if any, have been processed, LOC86 adds the relocatable segments that remain to the end of the list. If the first segment not previously used has a class name, then all other segments with the same class name are added to the list. After all segments of the class have been added to the list, then the next segment is added to the list.

This process continues until all segments have been added to the ordered list.

NOTE

Memory segments do not adhere to this process — a memory segment is always located at the top of memory, if possible. If an input module contains more than one memory segment, only the first is placed at the top of memory; the other segments are treated as any other relocatable segment.

Assigning Addresses to Relocatable Segments

Once LOC86 completes the ordered list of relocatable segments, it begins assigning addresses. LOC86 will never assign addresses that conflict with the location of absolute segments or the RESERVE control or between 00H and 200H, since that area is reserved for interrupt routines.

Starting at location 200H, LOC86 scans free memory to find an area in which the first segment will fit. When LOC86 finds a suitable address, it assigns it to the segment and removes that area from free memory. LOC86 then scans free memory for an area that will fit the next segment in the ordered list. LOC86 begins scanning at the end of the previous segment.

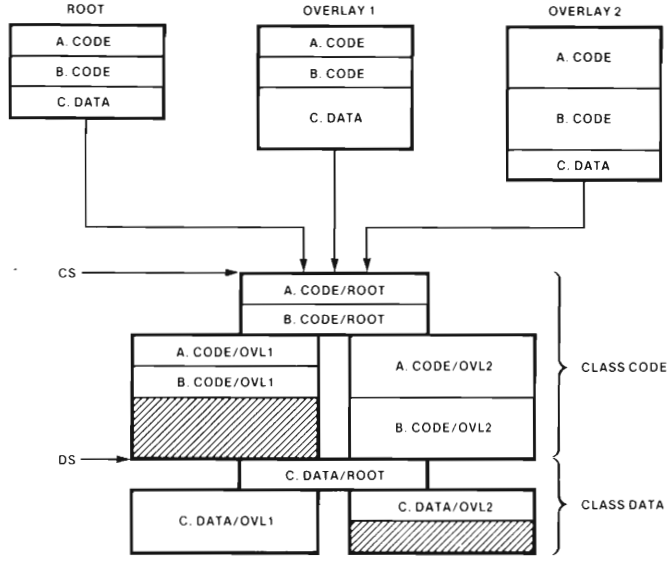
IF LOC86 reaches the end of memory and all of the relocatable modules have not been located, it makes an additional scan through free memory. The scanning process continues until all modules have been located.

LOC86's Algorithm for Locating Modules Containing Overlays

LOC86 locates programs with overlays in much the same way as it handles programs that do not contain overlays. However, there are some differences.

1. Segments contained in the root and each overlay are ordered separately.
2. Segments that are common to both the root and overlays (e.g.,STACK and MEMORY) are put at the end of the list of relocatable segments.
3. Segments in the root are located at the lowest available addresses in memory.
4. Segments contained in the overlays are located at the first available address above the root.
5. Segments common to the root and overlays are located immediately above the largest overlay in the file.

Figure 5-4 illustrates how LOC86 treats two PL/M-86 programs that use overlays. Figure 5-4a shows how segments are located when the modules are compiled with the LARGE model. Figure 5-4b shows how segments are located when the modules are compiled with the SMALL model of segmentation.



a. LARGE Model

121616-7

Figure 5-4. LOC86's Address Assignments for Overlays

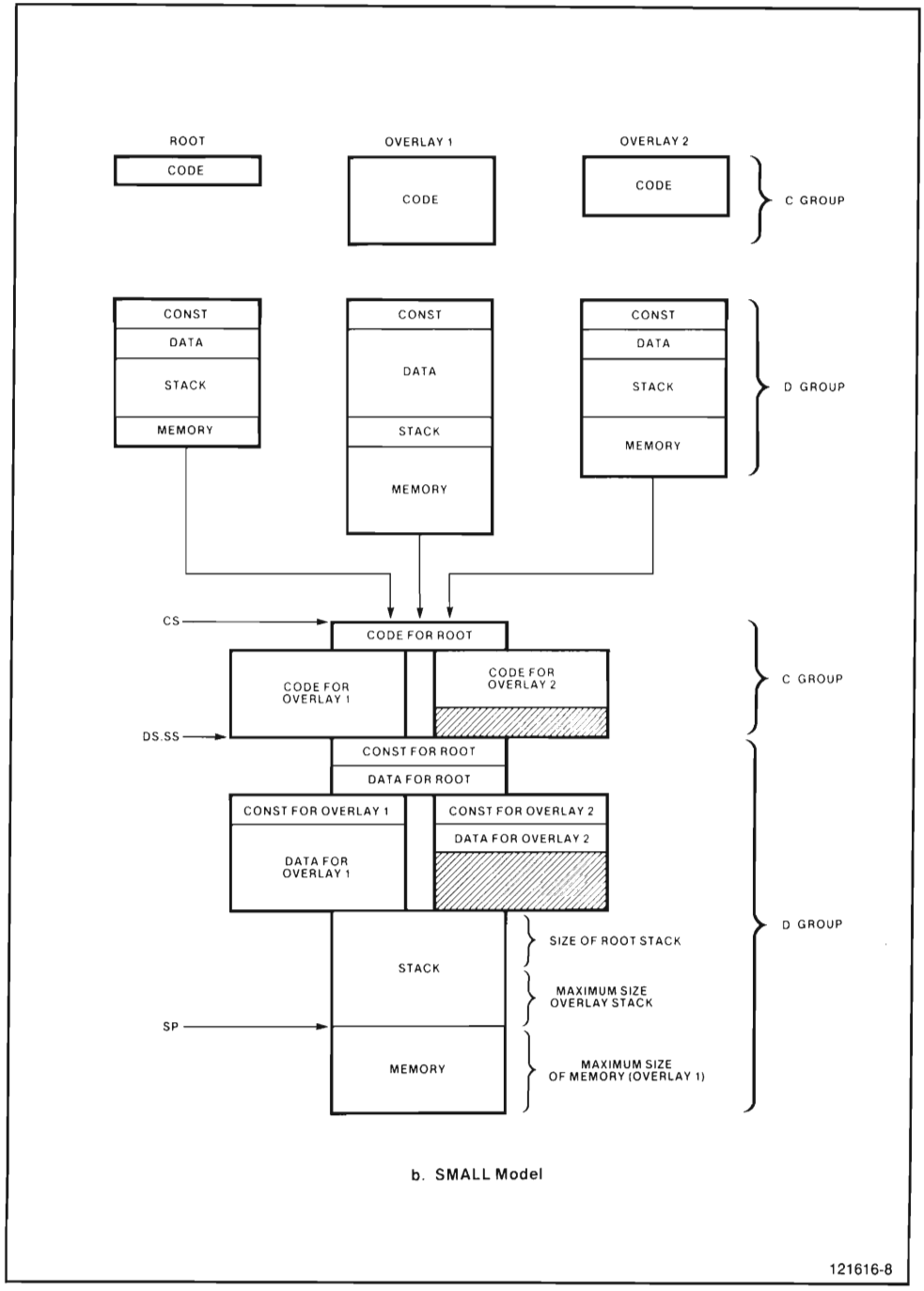


Figure 5-4. LOC86's Assignments for Overlays (Cont'd.)

OH86 6

OH86 converts 8086 absolute object modules to 8086 hexadecimal format. The input module must be in absolute format, and it may not contain overlays or register initialization records.

Figure 6-1 illustrates the object-to-hexadecimal conversion process. Any errors encountered during execution are displayed at the console output device.

For definition of file-naming conventions and syntax notation, refer to Notational Conventions following the Preface. For information on error and warning messages which may be produced, refer to Appendix H.

The general syntax for the invocation line is:

[*directory-name*]OH86 *input file*[TO *output file*]

The *input file* contains an 8086 absolute object module.

TO *output file* designates the file to receive the 8086 hexadecimal format. If *output file* is not specified, then output is directed to a file that has the same pathname as the input list, but its extension is HEX.

See Chapter 7 for operating system-specific examples of the OH86 invocation.

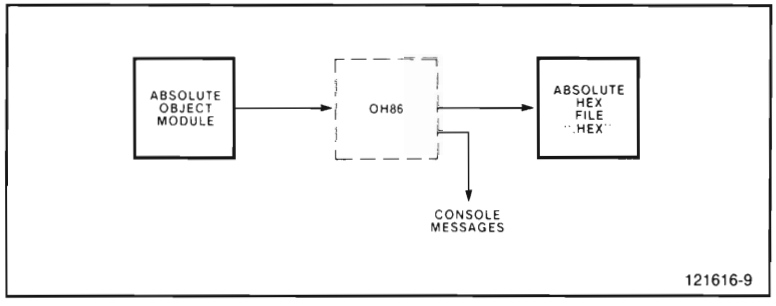


Figure 6-1. OH86 Input and Output Files

Using the iAPX 86,88 Utilities under DOS 7

This chapter contains information and examples for using the iAPX 86,88 utilities in a DOS environment to create executable code for an iRMX 86-based system.

Hardware/Software Environment

The iAPX 86,88 Utilities can run on an IBM XT or AT under the DOS operating system, version 3.0 (or later), and require at least 192K of memory. The code produced in this environment is executable on an iRMX 86-based system, as well as on various other Intel and non-Intel target systems. The examples in this chapter show how to produce code that is executable on an iRMX 86-based system.

Operating System Considerations

In this manual, the examples assume that the iAPX 86,88 utilities have already been installed in the directory named C:\intel. For this reason, the pathname is not specified in any of the examples. Refer to the *Disk Operating System* manual in the IBM Personal Computer Computer Language Series for information on changing the default directory and pathname as well as for further details on the DOS operating system.

Command Line

The general form of the command line used to invoke LINK86 is as follows:

```
C>LINK86 inputlist TO outputfile controls
```

where

C>	is the DOS prompt.
LINK86	invokes LINK86.
inputlist	is one or more modules to be linked together into a single object module.
outputfile	designates the file to receive the linked object module.
controls	are the LINK86 controls discussed in chapter 2.

Example

In this example, the LINK86 utility is invoked to link TEST.OBJ, SMALL.LIB, and USER.LIB with the BIND control in effect:

```
C>LINK86 PROG\TEST.OBJ, PROG\SMALL.LIB, & <cr>
>>USER.LIB TO PROG\TEST BIND <cr>
```

For general invocation examples of the other utilities (CREF86, LIB86, LOC86, and OH86), see the appropriate chapter for the utility. In addition, specific invocation examples are shown of each of the utilities at the end of this chapter.

Note that DOS places a 128 character limit on the length of the first line in the invocation sequence.

Continuation lines are necessary when a command or invocation will not fit on one line. The ampersand (&) should be used as the continuation character in this case.

Automating Program Invocation and Execution

DOS offers two ways of automatically invoking and executing multiple programs: batch files and command files. The two sections that follow provide examples that demonstrate how to use these files when using Intel software development tools.

DOS Batch Files

A DOS batch file is a file that contains one or more commands that DOS executes one at a time. A batch file can contain commands that are valid only within a batch file. All batch files must have the extension .BAT. See the *Disk Operating System* manual in the IBM Personal Computer Computer Language Series for information on how to create a DOS batch file.

You can pass parameters to a DOS batch file when the file executes. This way, the batch file can do similar work on a different program, or set of data, each time the batch file is executed. The following example illustrates how to use a batch file this way.

Example

In this example, the batch file PLM.BAT contains the command sequence that invokes the PL/M-86 compiler, and then invokes LINK86. LINK86 links the object module that resulted from the compilation to SMALL.LIB, in turn producing a bound object module with the extension .86. Each time this batch file is invoked, any PL/M sourcefile, with the extension .PLM, can be given as the parameter to be passed to the batch file. The percent signs and the following digit in the batch file will be replaced with the parameters that are specified on the command line that invokes the batch file.

1. Create a batch file named PLM.BAT that contains the following lines:

```
PLM86 %1.PLM
LINK86 %1.OBJ, SMALL.LIB TO %1.86 BIND
```

2. Invoke the batch file by typing the name of the batch file (without specifying the .BAT extension), followed by the name of the source file to be compiled. Do not specify the sourcefile extension.

```
PLM PROG1
```

   This executes the batch file named PLM.BAT which invokes the compiler to translate PROG1.PLM, in turn passing the resulting file PROG1.OBJ to LINK86. LINK86 then links PROG1.OBJ with SMALL.LIB to produce PROG1.86.

Other important characteristics of DOS batch files are listed below.

- Batch files accept control flow constructs such as IF and GOTO. For example,

```
IF ERRORLEVEL n GOTO label
```

allows the result of program execution in a batch file to determine which program in the batch file will be executed next. (The value of ERRORLEVEL is the error code returned by the last program executed.) If the error code returned is greater than or equal to the value specified for n, then control will be transferred to the line immediately after label. Thus, the batch file in the previous example, PLM.BAT, could contain the following lines:

```
PLM86 %1.PLM
IF ERRORLEVEL 1 GOTO STOP
LINK86 %1.OBJ, SMALL.LIB TO %1.86 BIND
:STOP
```

- Batch files cannot be nested. If a batch file references another batch file, control will be passed directly to the other batch file, but control will not return to the referring batch file.
- To process continuation lines in DOS batch files(not supported by DOS) you must redirect the input from a file that contains continuation lines to a batch file.

Example

In the example that follows, two files are created: LINKBIG.BAT is a batch file and LINKBIG.CON is a file containing continuation lines that must be redirected to the batch file upon execution. To redirect the file containing continuation lines (named LINKBIG.CON) to the batch file(LINKBIG.BAT), perform the following steps.

1. Create a batch file named LINK.BAT, and enter the following:

   ```
   LINK86 MODULE1.OBJ, MODULE2.OBJ, & < LINKBIG.CON
   ```

2. Create a text file named LINKBIG.CON that contains the following continuation lines:

   ```
   MODULE3.OBJ, MODULE4.OBJ, &
   PLM86.LIB, SMALL.LIB &
   TO BIGPROG.86
   ```

3. Execute the batch file by typing the following:

   ```
   LINKBIG
   ```

When LINKBIG.BAT is executed LINK86 is invoked, and MODULE1, MODULE2, PLM86.LIB and SMALL.LIB are linked; the resulting object module is placed in BIGPROG.86.

A batch file can contain several invocation lines, but each invocation line must fit on a single line. In the following example, the batch file GENBIG.BAT contains several invocation lines.

```
plm86 module1.plm
plm86 module2.plm
plm86 module3.plm
asm86 module4.asm
link86 module1.obj, module2.obj, & < linkbig.con
```

To execute GENBIG.BAT, type the following:

```
C>GENBIG <cr>
```

All of the modules in GENBIG.BAT will be compiled or assembled and then linked to produce the BIGPROG.86.

Command Files

Under DOS version 2.0 or later it is possible to invoke the DOS command line interpreter program, COMMAND.COM, with input that is redirected from a file (called a command file). This file can contain a sequence of DOS commands, as well as those that invoke programs such as the PL/M-86 compiler. This command file must contain the DOS EXIT command as its final line.

For example, if you create a command file named MAKEPROG.CMD that contains the following information:

```
PLM86 main.plm
PLM86 io.plm
PLM86 util.PLM
LINK86 main.obj, io.obj, util.obj, small.lib
to prog.86 bind
EXIT
```

It is now possible to redirect the commands in this file to the command line interpreter by typing the following:

```
C>COMMAND < MAKEPROG.CMD
```

COMMAND.COM will then invoke all commands listed in the file MAKEPROG.CMD.

The following considerations apply when invoking the command line interpreter (COMMAND.COM) with input that is redirected from a command file:

- This method of redirecting commands only works for a fixed sequence of commands; it is not possible to pass parameters to COMMAND.COM.
- The DOS batch file commands that allow conditional execution of portions of the command file (IF and GOTO) are not supported; commands are always executed sequentially.
- Command files can be nested by reinvoking COMMAND.COM from the primary command file with input redirected from a secondary command file. The secondary command file must contain an EXIT command as its final line. When the EXIT command is executed, control returns to the point in the primary file immediately following the point from which the secondary file was invoked.
- Command files, unlike DOS batch files, can contain continuation lines. For example, the following is a valid command file:

```
LINK86 &
  file1.obj, &
  file2.obj &
  to file12.lnk
LINK86 &
  file3.obj, &
  file4.obj &
  to file34.lnk
LINK86 &
  file12.lnk, &
  file34.lnk &
  to files.86 bind
EXIT
```

If a command file is invoked with output redirected to a file, that file will contain a complete log of all console output created during the execution of the command

file, including the invocation line for each program executed in the command file. For example, the following command would invoke the command file MAKEPROG.CMD, and would create a log file named MAKEPROG.LOG.

```
C>COMMAND < MAKEPROG.CMD > MAKEPROG.LOG
```

Work Files

The utilities create work files during processing, and delete them at the end of processing. These files are designated :WORK: and they do not conflict with any other files.

The DOS environment provides a mechanism for selecting the drive where :WORK: files are to be placed. The default drive is C:, but another drive can be selected as shown in the following example:

Example

```
C>SET :WORK:=d:\
```

In this example, the DOS SET command sets the current directory in which temporary files are created to be placed in the root directory d. This location is useful when the DOS VDISK.SYS device driver has been used to create a virtual disk in memory.

Generating Code to Run on an iRMX™ 86-Based System

To generate code that runs on an iRMX 86-based system, perform the following steps:

1. Translate the program into object code by using the appropriate compiler or assembler.
2. Use LINK86 to link the program with other routines or libraries as necessary. When doing this, remember the following:
   - If you wrote your program in FORTRAN or Pascal, or if you invoked specific universal development interface (UDI) calls, you must link your program to the iRMX 86 UDI library that corresponds to the model of segmentation for your program. These libraries are:

Library	Model of Segmentation
URXLRB.LIB	LARGE or MEDIUM
URXCOM.LIB	COMPACT
URXSML.LIB	SMALL

   - Do not use FASTLOAD control. Currently, the iRMX 86 Operating System cannot load programs linked with this control.
   - To produce LTL code, use the BIND control. In this case, also specify the MEMPOOL and SEGSIZE controls to allocate memory for the memory pool and stack. If you do not use BIND, you must specify SEGSIZE with the LOC86 command.
3. If you did not specify the BIND control in the LINK86 command, use LOC86 to assign absolute addresses to your program. In order to run this program in an

iRMX 86 environment, you must also reserve the program's memory locations during iRMX 86 configuration.

4. To invoke the program from a terminal, enter the pathname of the file that contains the program's linked (if LTL code) or located object code.

Program Development Examples

The following examples are programming problems solved by using one or more of the iAPX 86,88 utilities under the DOS operating system.

Example 1: Using CREF86

Figure 7-1 illustrates a CREF86 cross-reference listing for an input list of 15 files, one of which contains several modules. The output print file pathname OUT and a title for the listing were specified in the controls. Although PAGEWIDTH(PW) and PAGELENGTH (PL) specifications were also noted in the controls, the numbers specified are the same as those provided by default.

Example 2: Building and Using Library Files

A library is a file that contains object modules. Libraries allow you to collect commonly-used pieces of software into one file. The library file can be included in a LINK86 invocation, and LINK86 will use the modules to resolve references.

You can add the output from a translator, LINK86, or LOC86 to a library. The modules added may be relocatable or absolute; they can have unresolved references or be completely linked.

Consider the following example—you have created six routines (SINE, COSINE, TANGENT, COSECANT, SECANT, and COTANGENT). You want to create a library file that will allow each routine to be linked to programs separately.

The first step necessary to create the library is to translate each routine separately. If you were to put them in a single source module, the translator would translate them into one module with six public symbols. You could add this module to a library, but when you tried to link one of the routines into a program all six would be included.

Once the routines are translated, the interactive utility LIB86 can be used to create a library file and add modules. The LIST command is used to display the contents of the library and the publics contained within it.

```
C>LIB86<cr>

DOS 8086 LIBRARIAN Vx.y

*CREATE LIBRARY\TRIG.LIB<cr>
*ADD IN.OBJ, COS.OBJ TO LIBRARY\TRIG.LIB<cr>
*LIST LIBRARY\TRIB.LIB PUBLICS<cr>

  LIBRARY\TRIG.LIB
    SIN
      SINE
    COS
      COSINE
```

```
*ADD SEC.LNK, CSC.LNK, COT.LNK, &<cr>
**TAN.LNK TO LIBRARY\TRIG.LIB <cr>
*LIST LIBRARY\TRIG.LIB PUBLICS<cr>

  LIBRARY\TRIG.LIB
    SIN
      SINE
    COS
      COSINE
    SEC
      SECANT
    CSC
      COSECANT
    COT
      COTANGENT
    TAN
      TANGENT

*EXIT<cr>
```

Example 3: Linking and Locating Programs with Overlays Using OVERLAY Control

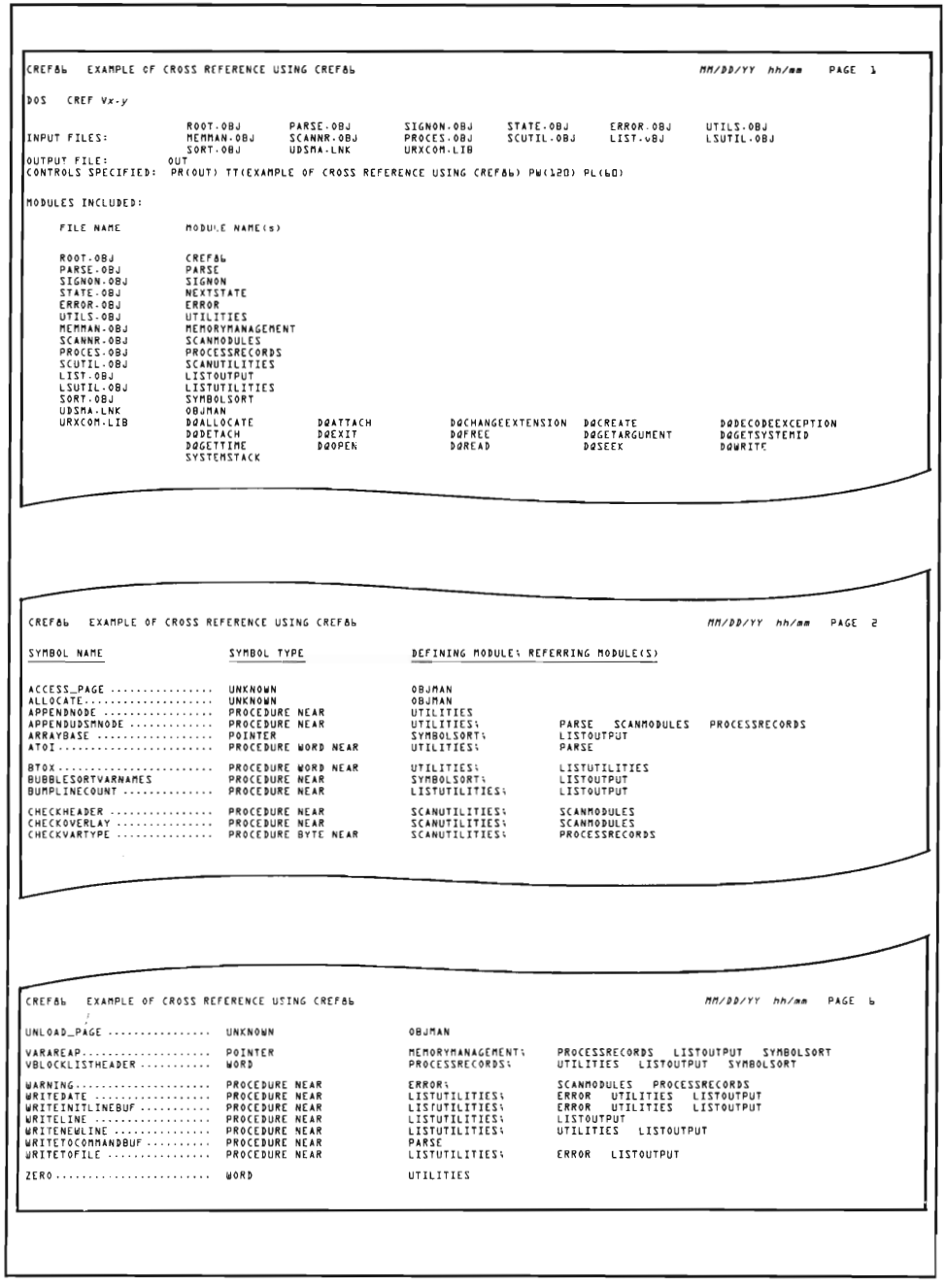
The easiest way to build an 8086 program that contains overlays is with LINK86's OVERLAY control. Overlay modules built with this control reside in the same file as the root. The target operating system (iRMX 86, in this case) supplies routines that will load the overlays constructed in this way. See the *iRMX™ 86 Loader Reference Manual* or the *Run-Time Support Manual for iAPX 86,88 Applications.*

After the program modules that will constitute the root and its overlays are translated, each of the overlays and the root must be linked separately. Then the root and all of the overlays are linked together.

The example following shows the first step toward linking overlays—linking all of the modules that will constitute each overlay and the root separately:

```
C>LINK86 OV1.OBJ, OV1A.OBJ, OV1B.OBJ &<cr>
>>OVERLAY(OVERLAY1)<cr>

C>LINK86 OV2.OBJ, OV2B.OBJ, OV2C.OBJ &<cr>
>>OVERLAY(OVERLAY2)<cr>

C>LINK86 OV3.OBJ, OV3A.OBJ OVERLAY(OVERLAY3)<cr>

C>LINK86 OV4.OBJ, OV4A.OBJ OVERLAY(OVERLAY)<cr>

C>LINK86 ROOT.OBJ, ROOTA.OBJ, ROOTB.OBJ, &<cr>
>>URXSML.LIB OVERLAY(ROOT)<cr>
```

Notice that all of the modules, including the root, are linked with the OVERLAY and NOBIND controls. The overlay name for the root is not as critical as for the overlays, since the overlay name is used when calling the loader.



```
CREF86  EXAMPLE OF CROSS REFERENCE USING CREF86                    MM/DD/YY hh/mm   PAGE 1

DOS   CREF Vx-y

                    ROOT.OBJ      PARSE.OBJ      SIGNON.OBJ     STATE.OBJ      ERROR.OBJ      UTILS.OBJ
INPUT FILES:        MEMMAN.OBJ    SCANNR.OBJ     PROCES.OBJ     SCUTIL.OBJ     LIST.OBJ       LSUTIL.OBJ
                    SORT.OBJ      UDSMA.LNK      URXCOM.LIB
OUTPUT FILE:        OUT
CONTROLS SPECIFIED: PR(OUT) TT(EXAMPLE OF CROSS REFERENCE USING CREF86) PW(120) PL(60)

MODULES INCLUDED:

     FILE NAME          MODULE NAME(S)

     ROOT.OBJ           CREF86
     PARSE.OBJ          PARSE
     SIGNON.OBJ         SIGNON
     STATE.OBJ          NEXTSTATE
     ERROR.OBJ          ERROR
     UTILS.OBJ          UTILITIES
     MEMMAN.OBJ         MEMORYMANAGEMENT
     SCANNR.OBJ         SCANMODULES
     PROCES.OBJ         PROCESSRECORDS
     SCUTIL.OBJ         SCANUTILITIES
     LIST.OBJ           LISTOUTPUT
     LSUTIL.OBJ         LISTUTILITIES
     SORT.OBJ           SYMBOLSORT
     UDSMA.LNK          OBJMAN
     URXCOM.LIB         D@ALLOCATE          D@ATTACH          D@CHANGEEXTENSION  D@CREATE          D@DECODEEXCEPTION
                        D@DETACH            D@EXIT            D@FREE             D@GETARGUMENT     D@GETSYSTEMID
                        D@GETTIME           D@OPEN            D@READ             D@SEEK            D@WRITE
                        SYSTEMSTACK
```

```
CREF86  EXAMPLE OF CROSS REFERENCE USING CREF86                    MM/DD/YY hh/mm   PAGE 2

SYMBOL NAME                 SYMBOL TYPE                 DEFINING MODULE\ REFERRING MODULE(S)

ACCESS_PAGE ..............  UNKNOWN                     OBJMAN
ALLOCATE..................  UNKNOWN                     OBJMAN
APPENDNODE ...............  PROCEDURE NEAR              UTILITIES
APPENDUDSMNODE ...........  PROCEDURE NEAR              UTILITIES\          PARSE   SCANMODULES   PROCESSRECORDS
ARRAYBASE ................  POINTER                     SYMBOLSORT\         LISTOUTPUT
ATOI......................  PROCEDURE WORD NEAR         UTILITIES\          PARSE

BTOX......................  PROCEDURE WORD NEAR         UTILITIES\          LISTUTILITIES
BUBBLESORTVARNAMES........  PROCEDURE NEAR              SYMBOLSORT\         LISTOUTPUT
BUMPLINECOUNT ............  PROCEDURE NEAR              LISTUTILITIES\      LISTOUTPUT

CHECKHEADER ..............  PROCEDURE NEAR              SCANUTILITIES\      SCANMODULES
CHECKOVERLAY .............  PROCEDURE NEAR              SCANUTILITIES\      SCANMODULES
CHECKVARTYPE .............  PROCEDURE BYTE NEAR         SCANUTILITIES\      PROCESSRECORDS
```

```
CREF86  EXAMPLE OF CROSS REFERENCE USING CREF86                    MM/DD/YY hh/mm   PAGE 6

UNLOAD_PAGE ..............  UNKNOWN                     OBJMAN

VARAREAP..................  POINTER                     MEMORYMANAGEMENT\   PROCESSRECORDS   LISTOUTPUT   SYMBOLSORT
VBLOCKLISTHEADER..........  WORD                        PROCESSRECORDS\     UTILITIES   LISTOUTPUT   SYMBOLSORT

WARNING...................  PROCEDURE NEAR              ERROR\              SCANMODULES   PROCESSRECORDS
WRITEDATE ................  PROCEDURE NEAR              LISTUTILITIES\      ERROR   UTILITIES   LISTOUTPUT
WRITEINITLINEBUF..........  PROCEDURE NEAR              LISTUTILITIES\      ERROR   UTILITIES   LISTOUTPUT
WRITELINE ................  PROCEDURE NEAR              LISTUTILITIES\      LISTOUTPUT
WRITENEWLINE .............  PROCEDURE NEAR              LISTUTILITIES\      UTILITIES   LISTOUTPUT
WRITETOCOMMANDBUF ........  PROCEDURE NEAR              PARSE
WRITETOFILE ..............  PROCEDURE NEAR              LISTUTILITIES\      ERROR   LISTOUTPUT

ZERO......................  WORD                        UTILITIES
```

Figure 7-1. CREF86 Cross-Reference Listing

Finally, the overlays and root must be linked together. Since any one of the files could be the root, LINK86 requires for the final link the file containing the root must be first in the input list. During this final link, the OVERLAY control is not used:

```
C>LINK86 ROOT.LNK, OV1.LNK, OV2.LNK, OV3.LNK &<cr>
>>OV4.LNK TO PROG.86 BIND<cr>
```

In the invocation, the BIND control is specified. The resulting object file is executable on an iRMX 86-based system.

Figure 7-2 shows the LINK86 print file listing for the previous invocation.

There is nothing special about the invocation line to LOC86 when locating a file that contains overlays or that has been bound:

```
C>LOC86 PROG.86 RESERVE (0H TO 77FFH, &<cr>
>>0FC000H TO 0FFFFFH)<cr>
```

The RESERVE control prevents LOC86 from assigning memory addresses reserved for the operating system. However, the values you enter with the RESERVE control must depend on the size and location of your operating system and other application software. Figure 7-3 illustrates the printout from this invocation.

Example 4: Linking and Locating Programs with Overlays Without OVERLAY Control

It is harder to produce overlay modules without using the OVERLAY control. However, sometimes it is necessary to build programs in this way, for example, building a program for running under an operating system that does not support overlay modules contained in the same file as the root module.

Regardless of the above reason, building overlays in this fashion places an extra burden on the programmer. He must do some of the work that would be left to LINK86 (and LOC86) if he were to use the OVERLAY control. The following example shows the preparation of a root and two overlay modules in separate files.

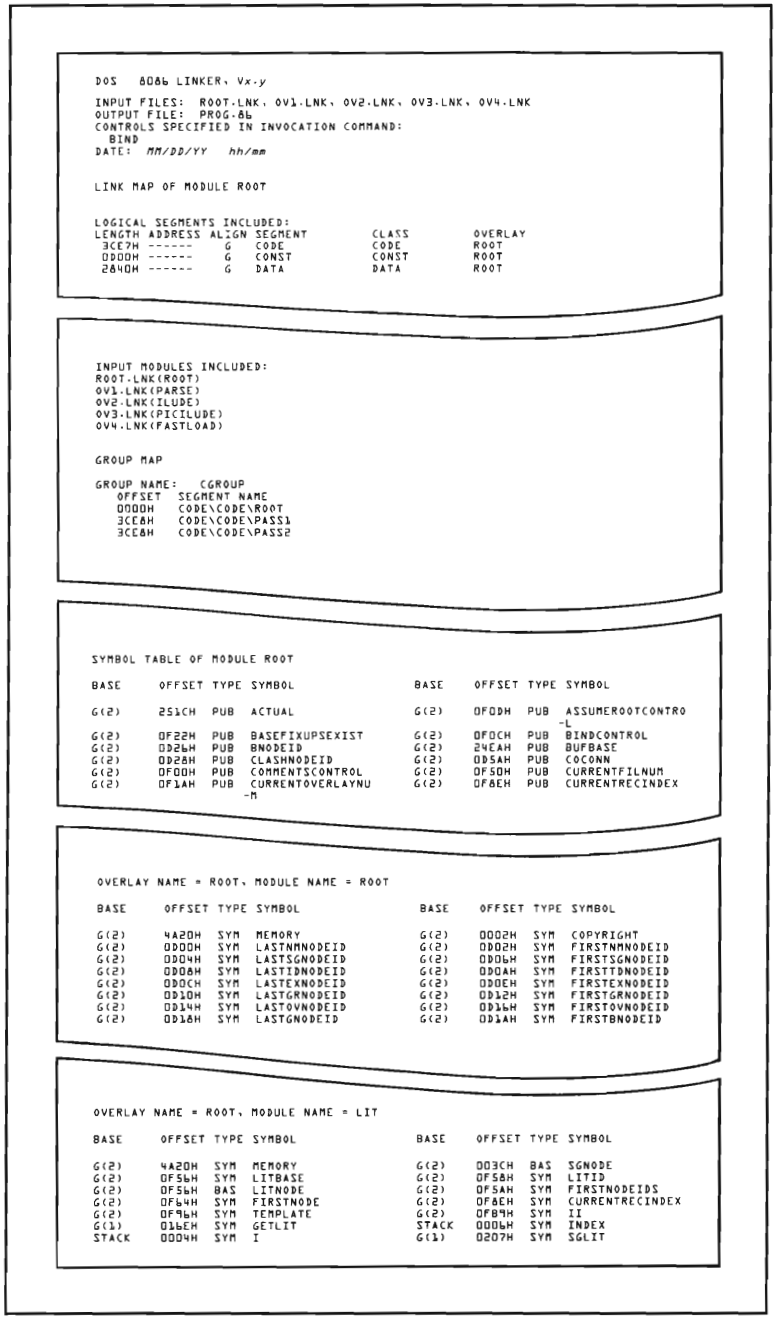
First you must compile all modules. Examples of the invocation lines are shown below:

```
C>PLM86 ROOT.SRC SMALL<cr>

C>PLM86 OV1.SRC SMALL<cr>

C>PLM86 OV2.SRC SMALL<cr>
```

In the next step we must link the root module to resolve external symbols with a library and to obtain a link map:

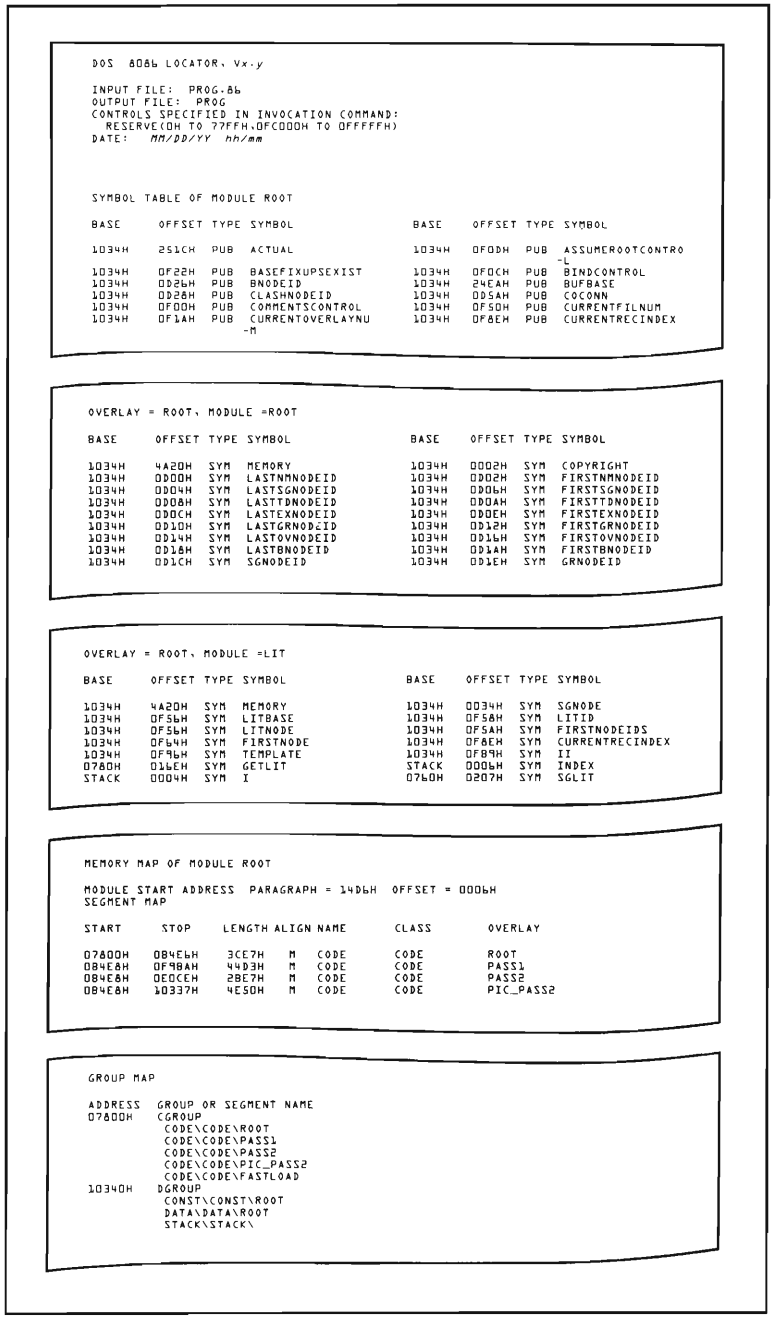
```
C>LINK86 ROOT.OBJ,USER.LIB MAP<cr>
```

You will need the link map for locating purposes. The link map, shown in figure 7-4, shows the size of each segment in the root. Since the overlays are self-contained except for references to the root, you do not need a link map for them. The PL/M-86 listing files will show the size of each overlay's segments, as illustrated in figure 7-5.



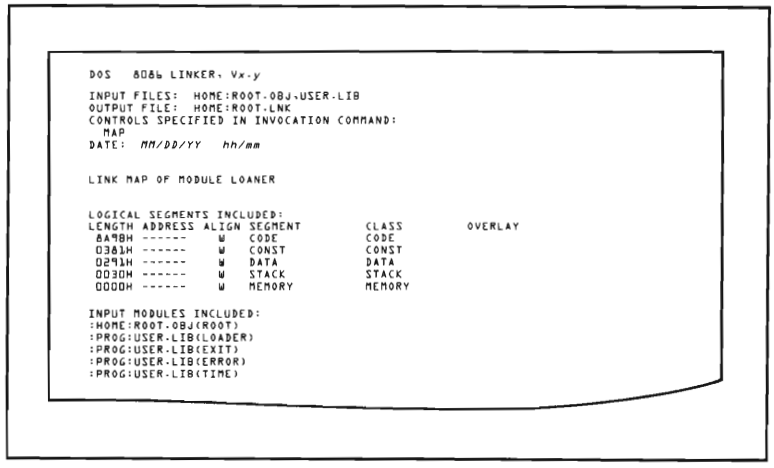
```
DOS   8086 LINKER, Vx.y

INPUT FILES:  ROOT.LNK, OV1.LNK, OV2.LNK, OV3.LNK, OV4.LNK
OUTPUT FILE:  PROG.86
CONTROLS SPECIFIED IN INVOCATION COMMAND:
  BIND
DATE:  MM/DD/YY   hh/mm

LINK MAP OF MODULE ROOT


LOGICAL SEGMENTS INCLUDED:
LENGTH ADDRESS ALIGN SEGMENT         CLASS         OVERLAY
 3CE7H ------    G   CODE            CODE          ROOT
 0D00H ------    G   CONST           CONST         ROOT
 2840H ------    G   DATA            DATA          ROOT


INPUT MODULES INCLUDED:
ROOT.LNK(ROOT)
OV1.LNK(PARSE)
OV2.LNK(ILUDE)
OV3.LNK(PICILUDE)
OV4.LNK(FASTLOAD)

GROUP MAP

GROUP NAME:   CGROUP
   OFFSET   SEGMENT NAME
   0000H    CODE\CODE\ROOT
   3CE8H    CODE\CODE\PASS1
   3CE8H    CODE\CODE\PASS2


SYMBOL TABLE OF MODULE ROOT

BASE     OFFSET TYPE SYMBOL                BASE     OFFSET TYPE SYMBOL

G(2)     251CH  PUB  ACTUAL                G(2)     0F0DH  PUB  ASSUMEROOTCONTRO
                                                                 -L
G(2)     0F22H  PUB  BASEFIXUPSEXIST       G(2)     0F0CH  PUB  BINDCONTROL
G(2)     0D26H  PUB  BNODEID               G(2)     24EAH  PUB  BUFBASE
G(2)     0D28H  PUB  CLASHNODEID           G(2)     0D5AH  PUB  COCONN
G(2)     0F00H  PUB  COMMENTSCONTROL       G(2)     0F50H  PUB  CURRENTFILNUM
G(2)     0F1AH  PUB  CURRENTOVERLAYNU      G(2)     0F8EH  PUB  CURRENTRECINDEX
                     -M


OVERLAY NAME = ROOT, MODULE NAME = ROOT

BASE     OFFSET TYPE SYMBOL                BASE     OFFSET TYPE SYMBOL

G(2)     4A20H  SYM  MEMORY                G(2)     0002H  SYM  COPYRIGHT
G(2)     0D00H  SYM  LASTNMNODEID          G(2)     0D02H  SYM  FIRSTNMNODEID
G(2)     0D04H  SYM  LASTSGNODEID          G(2)     0D06H  SYM  FIRSTSGNODEID
G(2)     0D08H  SYM  LASTIDNODEID          G(2)     0D0AH  SYM  FIRSTTDNODEID
G(2)     0D0CH  SYM  LASTEXNODEID          G(2)     0D0EH  SYM  FIRSTEXNODEID
G(2)     0D10H  SYM  LASTGRNODEID          G(2)     0D12H  SYM  FIRSTGRNODEID
G(2)     0D14H  SYM  LASTOVNODEID          G(2)     0D16H  SYM  FIRSTOVNODEID
G(2)     0D18H  SYM  LASTGNODEID           G(2)     0D1AH  SYM  FIRSTBNODEID


OVERLAY NAME = ROOT, MODULE NAME = LIT

BASE     OFFSET TYPE SYMBOL                BASE     OFFSET TYPE SYMBOL

G(2)     4A20H  SYM  MEMORY                G(2)     0D3CH  BAS  SGNODE
G(2)     0F56H  SYM  LITBASE               G(2)     0F58H  SYM  LITID
G(2)     0F56H  BAS  LITNODE               G(2)     0F5AH  SYM  FIRSTNODEIDS
G(2)     0F64H  SYM  FIRSTNODE             G(2)     0F8EH  SYM  CURRENTRECINDEX
G(2)     0F96H  SYM  TEMPLATE              G(2)     0FB9H  SYM  II
G(1)     016EH  SYM  GETLIT                STACK    0006H  SYM  INDEX
STACK    0004H  SYM  I                     G(1)     0207H  SYM  SGLIT
```

Figure 7-2. LINK86 Listing for Program with Overlays



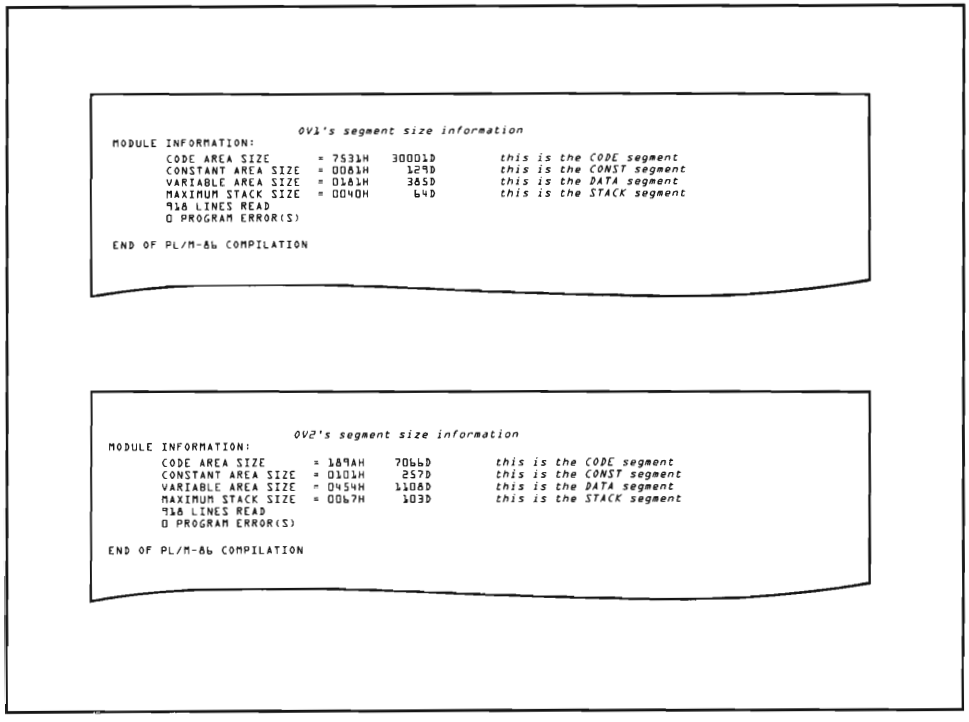
```
DOS  8086 LOCATOR, Vx.y

INPUT FILE:  PROG.86
OUTPUT FILE:  PROG
CONTROLS SPECIFIED IN INVOCATION COMMAND:
  RESERVE(0H TO 7FFFH,0FC000H TO 0FFFFFH)
DATE:   MM/DD/YY  hh/mm



SYMBOL TABLE OF MODULE ROOT

BASE     OFFSET TYPE SYMBOL                BASE     OFFSET TYPE SYMBOL

1034H    251CH  PUB  ACTUAL                1034H    0F0DH  PUB  ASSUMEROOTCONTRO
                                                                 -L
1034H    0F22H  PUB  BASEFIXUPSEXIST       1034H    0F0CH  PUB  BINDCONTROL
1034H    0D26H  PUB  BNODEID               1034H    24EAH  PUB  BUFBASE
1034H    0D28H  PUB  CLASHNODEID           1034H    0D5AH  PUB  COCONN
1034H    0F00H  PUB  COMMENTSCONTROL       1034H    0F50H  PUB  CURRENTFILNUM
1034H    0F1AH  PUB  CURRENTOVERLAYNU      1034H    0F8EH  PUB  CURRENTRECINDEX
                     -M


OVERLAY = ROOT, MODULE =ROOT

BASE     OFFSET TYPE SYMBOL                BASE     OFFSET TYPE SYMBOL

1034H    4A20H  SYM  MEMORY                1034H    0002H  SYM  COPYRIGHT
1034H    0D00H  SYM  LASTNMNODEID          1034H    0D02H  SYM  FIRSTNMNODEID
1034H    0D04H  SYM  LASTSGNODEID          1034H    0D06H  SYM  FIRSTSGNODEID
1034H    0D08H  SYM  LASTTDNODEID          1034H    0D0AH  SYM  FIRSTTDNODEID
1034H    0D0CH  SYM  LASTEXNODEID          1034H    0D0EH  SYM  FIRSTEXNODEID
1034H    0D10H  SYM  LASTGRNODEID          1034H    0D12H  SYM  FIRSTGRNODEID
1034H    0D14H  SYM  LASTOVNODEID          1034H    0D16H  SYM  FIRSTOVNODEID
1034H    0D18H  SYM  LASTBNODEID          1034H    0D1AH  SYM  FIRSTBNODEID
1034H    0D1CH  SYM  SGNODEID              1034H    0D1EH  SYM  GRNODEID


OVERLAY = ROOT, MODULE =LIT

BASE     OFFSET TYPE SYMBOL                BASE     OFFSET TYPE SYMBOL

1034H    4A20H  SYM  MEMORY                1034H    0034H  SYM  SGNODE
1034H    0F5BH  SYM  LITBASE               1034H    0F58H  SYM  LITID
1034H    0F5BH  SYM  LITNODE               1034H    0F5AH  SYM  FIRSTNODEIDS
1034H    0F64H  SYM  FIRSTNODE            1034H    0F8EH  SYM  CURRENTRECINDEX
1034H    0F96H  SYM  TEMPLATE              1034H    0F89H  SYM  II
0780H    016EH  SYM  GETLIT                STACK    0006H  SYM  INDEX
STACK    0004H  SYM  I                     0760H    0207H  SYM  SGLIT


MEMORY MAP OF MODULE ROOT

MODULE START ADDRESS  PARAGRAPH = 14D6H  OFFSET = 0006H
SEGMENT MAP

START     STOP     LENGTH ALIGN NAME      CLASS       OVERLAY

07800H    0B4E6H    3CE7H   M   CODE      CODE        ROOT
0B4E8H    0F98AH    4A03H   M   CODE      CODE        PASS1
0B4E8H    0EDCEH    2BE7H   M   CODE      CODE        PASS2
0B4E8H    10337H    4E50H   M   CODE      CODE        PIC_PASS2


GROUP MAP

ADDRESS   GROUP OR SEGMENT NAME
07800H    CGROUP
           CODE\CODE\ROOT
           CODE\CODE\PASS1
           CODE\CODE\PASS2
           CODE\CODE\PIC_PASS2
           CODE\CODE\FASTLOAD
1034OH    DGROUP
           CONST\CONST\ROOT
           DATA\DATA\ROOT
           STACK\STACK\
```

Figure 7-3. LOC86 Listing for Program with Overlays



```
DOS   8086 LINKER, Vx.y

INPUT FILES:  HOME:ROOT.OBJ,USER.LIB
OUTPUT FILE:  HOME:ROOT.LNK
CONTROLS SPECIFIED IN INVOCATION COMMAND:
  MAP
DATE:  MM/DD/YY   hh/mm

LINK MAP OF MODULE LOANER


LOGICAL SEGMENTS INCLUDED:
LENGTH ADDRESS ALIGN SEGMENT          CLASS         OVERLAY
 8A9BH ------    W   CODE             CODE
 0381H ------    W   CONST            CONST
 0291H ------    W   DATA             DATA
 0030H ------    W   STACK            STACK
 0000H ------    W   MEMORY           MEMORY

INPUT MODULES INCLUDED:
:HOME:ROOT.OBJ(ROOT)
:PROG:USER.LIB(LOADER)
:PROG:USER.LIB(EXIT)
:PROG:USER.LIB(ERROR)
:PROG:USER.LIB(TIME)
```

Figure 7-4. LINK86 Map for Root 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      = 189AH    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
```

Figure 7-5. Module Information for Overlays

Note that the length of the root's code segment and OV1's code segment must 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, you can locate the root module. The resulting file will not be executable, but it allows you 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 0FFCEH, and the CS register is initialized to 0.) The STACK and MEMORY segments will be located above OV2's DATA segment:

```
C>LOC86 ROOT.LNK &<cr>
>>ADDRESSES(GROUPS(CGROUP)0H),DEGROUP)0FFCEH)), &<cr>
>>          SEGMENTS(CODE(0H0,CONST(0FFCEH),STACK(10B34H000 &<cr>
>>ORDER(SEGMENTS(CODE,CONST,DATA,STACK,MEMORY)) &<cr>
>>SEGSIZE(STACK(100H))<cr>
```

Once the root is located, you 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 you use the located overlays to resolve external references in the root:

```
C>LINK86 OV1.OBJ,PUBLICSONLY(ROOT) &<cr>
>>NOPUBLICS EXCEPT(OV1CODE, OV1DAT)<cr>

C>LINK86 OV2.OBJ,PUBLICSONLY(ROOT) &<cr>
>>NOPUBLICS EXCEPT(OV2CODE, OV2DATA)<cr>
```

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 you first located the root. In this case the STACK and MEMORY segments must be the same for the overlays and the root:

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

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

The CGROUP and DGROUP base address must be specified in order to compute offset information. The final base address assigned to DGROUP by LOC86 will be rounded down 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.

```
C>LINK86 ROOT.OBJ,USER.LIB, &<cr>
>>PUBLICSONLY(OV1,OV2)<cr>

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

The executable forms of the root and its overlay files are contained in ROOT, OV1, and OV2. Figure 7-6 shows the resulting layout of memory.

Invocation Examples

The remainder of this chapter consists of a list of invocation examples for each of the iAPX 86,88 Family Utilities with each of the controls for that utility.

These examples should be used in conjunction with the syntax specifications given in the chapter for the specific utility. (Chapter 2 for LINK86, Chapter 3 for CREF86, Chapter 4 for LIB86, Chapter 5 for LOC86, and Chapter 6 for OH86.)

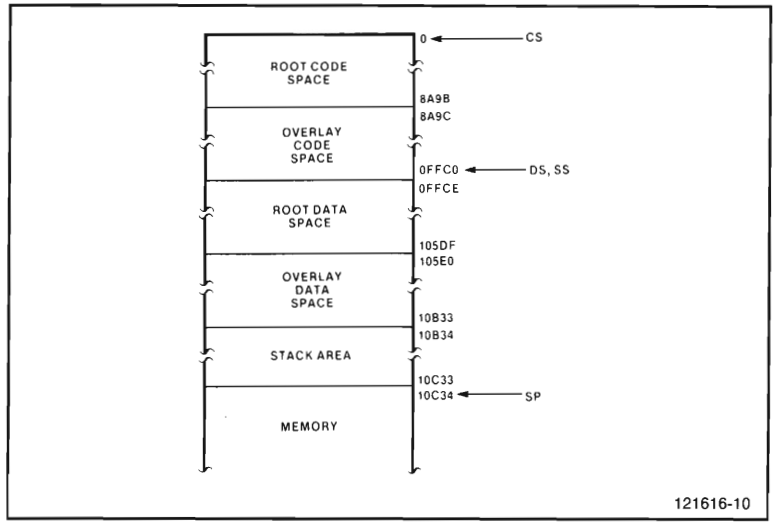


Figure 7-6. Memory Organization for Example 4

LINK86 Examples

ASSIGN

```
C>LINK86 FILE1, FILE2, FILE3 & <cr>

>>ASSIGN (VARONE(50H), & <cr>

>>VARTWO(2000H))<cr>
```

The above example defines two public symbols, VARONE and VARTWO, with absolute addresses 50H and 2000H, respectively.

ASSUMEROOT

```
C>LINK86 OV1.OBJ, OV2.OBJ, & <cr>
>>LIB1, LIB2 TO OVL1 OVERLAY (OVL1) & <cr>
>>ASSUMROOT (RTFILE) <cr>
```

In the above example, the root file is RTFILE, and LIB1 and LIB2 are library files.

BIND/NOBIND

```
C>LINK86 TEST.OBJ, & <cr>

>>LIBRARY\USER.LIB & <cr>

>>BIND PRINT<cr>
```

The above example creates an LTL module. The output object file is TEST with no extension.

```
C>LINK86 GENERAL.OBJ & <cr>
>>NOBIND<cr>
```

The above example specifies default to avoid ambiguity.

COMMENTS/NOCOMMENTS

```
C>LINK86 SYSTEM\PROG.OBJ &<cr>
>>TO SYSTEM\TEMP.TST &<cr>
>>COMMENTS<cr>
```

FASTLOAD/NOFASTLOAD

```
C>LINK86 PROG.OBJ, LIB1, LIB2 &<cr>
>>BIND FASTLOAD <cr>
```

Do not use the FASTLOAD control when producing code for an iRMX environment.

GROUPOVERLAYS/NOGROUPOVERLAYS

```
C>LINK86 PROG.OBJ TO PROG &<cr>
>>BIND GROUPOVERLAYS <cr>
```

The segments in PROG.OBJ will be grouped by default for optimization of static memory usage.

INITCODE

```
C>LINK86 MYPROG INITCODE<cr>
```

LINES/NOLINES

```
C>LINK86 TEST\RN.OBJ NOLINES<cr>

C>LINK86 TEST\RN.OBJ LINES<cr>
```

LINES is the default, so it need not be specified.

MAP/NOMAP

```
C>LINK86 TESTER.OBJ MAP<cr>

C>LINK86 MAIN.OBJ, &<cr>
>>USER.OBJ, &<cr>
>>PUBLICSONLY(8087.LOC) &<cr>
>>NOMAP<cr>
```

MEMPOOL

```
C>LINK86 TEST.OBJ, &<cr>
>>USER.LIB, PASCAL.LIB BIND &<cr>
>>MEMPOOL(+20H)<cr>
```

The above MEMPOOL example will increase the minimum dynamic memory requirements by 20H bytes, and by default the maximum size will increase, if necessary, to equal the minimum.

```
C>LINK86 USER\TEST.OBJ &<cr>
>>MEMPOOL(100H, +200H) BIND<cr>
```

The minimum dynamic memory requirement is 100H. The maximum dynamic memory requirement is 300H.

NAME

```
C>LINK86 TOM.LBJ, &<cr>
>>SYS.LIB NAME &<cr>
>>('THIS IS A VERY LONG MODULE@NAME')<cr>
```

The LINK86 output module in the above example will have the name specified in parentheses in the control.

OBJECTCONTROLS

```
C>LINK86 FINAL, &<cr>
>>USER.LIB, SYS.LIB &<cr>
>>OBJECTCONTROLS (PURGE)<cr>
```

The above example removes all debug and public records from the object file.

```
C>LINK86 PASCL1.OBJ &<cr>
>>OBJECTCONTROLS(PURGE), &<cr>
>>NOPUBLICS EXCEPT(START, &<cr>
>>DATA1, DATA2)<cr>
```

The EXCEPT in the NOPUBLICS overrides the PURGE.

ORDER

```
C>LINK86 PLMPRG.OBJ, &<cr>
>>PLM.LIB, URXSML.LIB, &<cr>
>>USER.LIB ORDER (DGROUP(SEG1, SEG2), & <cr>
>>CGROUP (CSEG1, CSEG2, CSEG3)) <cr>
```

This use of ORDER specifies the order of segments for two groups.

OVERLAY/NOOVERLAY

```
C>LINK86 FILE1, FILE2, FILE3 &<cr>
>>TO OV1.LNK & <cr>
>>OVERLAY(OVERLAY1)<cr>
```

The above example will create an overlay record. The name of the overlay will be OVERLAY1.

PRINT/NOPRINT

```
C>LINK86 USER\PROG.OBJ &<cr>
>>TO USER\TEMP1.TST &<cr>
>>PRINT<cr>
```

The print file in the above example is: USER\TEMP1.MP1.

```
C>LINK86 PROG.OBJ<cr>
```

The print file in the above example is PROG.MP1

```
C>LINK86 PROG.OBJ, & <cr>
>>USER.LIB PRINT &<cr>
>>(THE.MAP)<cr>
```

The print file in the above example is THE.MAP.

PRINTCONTROLS

```
C>LINK86 TEMP.OBJ BIND &<cr>
>>PRINTCONTROLS(NOLINES, &<cr>
>>NOCOMMENTS, NOSYMBOLS) <cr>
```

The above example removes information about line numbers, local symbols, and comments from the print file.

```
C>LINK86 PASCL1.OBJ &<cr>
>>PRINTCONTROLS (PURGE)<cr>
```

The above statement removes all but the segment information and error messages from the print file.

PUBLICS/NOPUBLICS

```
C>LINK86 TEST.OBJ, &<cr>
>>USER.LIB NOPUBLICS EXCEPT &<cr>
>>(DATA1, DATA2, LABEL3, PROC4)<cr>
```

Public information concerning only DATA1, DATA2, LABEL3, and PROC4 is placed in the object file.

```
C>LINK86 TEMP.OBJ, &<cr>
>>URXSML.LIB, USER.LIB PUBLICS<cr>
```

All public symbol information will be included in the print file and output file.

PUBLICSONLY

```
C>LINK86 PUBLICSONLY & <cr>
>>(8087.LOC)<cr>
```

The above example will produce a file containing only the absolute public symbol records from 8087.LOC. The object file will be 8087.LNK.

```
C>LINK86 ROOT.OBJ, &<cr>
>>PUBLICSONLY(OV1, OV2)<cr>
```

The above example will resolve the references in ROOT.OBJ to absolute public symbols in the separately linked and located overlays OV1 and OV2.

PURGE/NOPURGE

```
C>LINK86 INDEX.OBJ PURGE<cr>
```

The above example produces an object file containing no debug or public information.

```
C>LINK86 FINAL.OBJ &<cr>
>>PRINTCONTROLS(NOPURGE)<cr>
```

The above example confirms that the line and symbol information should be kept in the print file.

RENAMEGROUPS

```
C>LINK86 PLMPRG.OBJ & <cr>
>>RENAMEGROUPS(CGROUP TO <cr>
>>THE@CODE)<cr>
```

The above example will change the translator-assigned name CGROUP to THE@CODE. A subsequent linkage would not merge THE@CODE with a group named CGROUP.

```
C>LINK86 ASMPRG.OBJ &<cr>
>>RENAMEGROUPS(CODE TO CGROUP)<cr>
```

The above example changes the group name CODE to CGROUP.

SEGSIZE

```
C>LINK86 GEORGE.OBJ &<cr>
>>USER.LIB, SYSTEM.LIB BIND &<cr>
>>SEGSIZE(MEMORY (15FFH, &<cr>
>>+2000H))<cr>
```

The above example tells the loader that 15FFH bytes of code is the minimum requirement for MEMORY. The new maximum size of MEMORY is 35FFH.

```
C>LINK86 PROJECT.OBJ, &<cr>
>>REST.LIB SEGSIZE &<cr>
>>(MEMORY(+1FF,+1FF))<cr>
```

In the above example, MEMORY's minimum size is incremented by 1FFH bytes. The maximum size of MEMORY is equal to the old minimum size plus 3FEH.

SYMBOLS/NOSYMBOLS

```
C>LINK86 TEMP.OBJ, &<cr>
>>USER.LIB NOSYMBOLS &<cr>
>>SYMBOLS<cr>
```

In the above example the local symbol records will be included in the object file.

```
C>LINK86 TEST.OBJ, &<cr>
>>USER.LIB &<cr>
>>PURGE <cr>
```

PURGE is a shorthand for NOSYMBOLS, NOCOMMENTS, NOPUBLICS, NOTYPE, and NOLINES.

SYMBOLCOLUMNS

```
C>LINK86 TEST.OBJ &<cr>
>>SYMBOLCOLUMNS(1)<cr>
```

In the above example, SYMBOLCOLUMNS has no effect, since BIND was not specified.

```
C>LINK86 ROOT.LNK, &<cr>
>>OV1.LNK, OV2.LNK, &<cr>
>>PUBLICSONLY(8087)&<cr>
>>SYMBOLCOLUMNS(4) BIND &<cr>
>>PRINT<cr>
```

In the above example, the symbol table will be printed on a line printer with 4 columns on each line.

TYPE/NOTYPE

```
C>LINK86 LIBMOD.OBJ TYPE<cr>
```

In the above example, LIBMOD will retain its type information.

CREF86 Examples

PAGELENGTH

```
C>CREF86 FILE1, FILE.LIB & <cr>
>>PAGELENGTH(35) <cr>
```

The cross-reference listing produced from the above example will have 35 lines on each page.

PAGEWIDTH

```
C>CREF86 PROGRAM, & <cr>
>>PROG.LIB(MOD1) & <cr>
>>PAGEWIDTH (100 <cr>
```

The cross-reference listing produced from the above example will be 100 characters wide, maximum, per page.

PRINT

```
C>CREF86 MYPROG, HISPROG, HERPROG &<cr>
>>PRINT (MYFILE) <cr>
```

The pathname of the print file in the above example will be MYFILE.

TITLE

```
C>CREF86 MYPROG, HISPROG, HERPROG, &<cr>
>>MYLIB, HISLIB, HERLIB TITLE &<cr>
>>('A CROSS-REFERENCE') & <cr>
>>PAGEWIDTH(105) <cr>
```

In the above example, the message in the TITLE control must be placed on one line. If the message contains special characters, it must be enclosed in single quotes (').

LIB86 Examples

ADD

```
*ADD SIN, COS, TAN TO & <cr>
**USER.LIB <cr>
```

In the above example, three object files are added to the USER.LIB.

```
*ADD LIB.ABC(MOD1, MOD2, MOD3) & <cr>
**TO PROJ.TOM <cr>
```

Three modules from the LIB.ABC are added to PROJ.TOM in the above example.

CREATE

```
*CREATE SYSTEM\TOMS.LIB <cr>
```

The example shown above will produce an empty library file called TOMS.LIB.

```
*CREATE USER.LIB <cr>
```

DELETE

```
*DELETE USER.LIB(TEMP1, & <cr>
**TEMP3, TEM_TMP, TEST?) <cr>

*DELETE IO.LIB(FLOPPY, CRT, & <cr>
**PAPER, TAPE) <cr>
```

In the above examples, four modules are deleted from the library USER.LIB.

EXIT

```
*EXIT <cr>
```

LIST

```
*LIST USER.LIB <cr>
  USER.LIB
    TEMP
    TEST
    EXEC
    MAIN
    LOOP
*LIST USER.LIB(TEMP, TEST)<cr>
  USER.LIB
    TEMP
    TEST
```

```
*LIST USER.LIB,TEMP.LIB<cr>
  USER.LIB
    TEMP
    TEST
    EXEC
    MAIN
    LOOP
  TEMP.LIB
    MODULE1
    MODULE3
    MODULETC
```

LOC86 Examples

ADDRESS

```
C>LOC86 COME.LNK TO WENT & <CR>
>>ADDRESSES(SEGMENTS(SEG1(15FFH), &
>>SEG2(4F5AH)) <cr>
```

In the above example, if SEG1 is byte alignable, it will be located at 15FFH. If SEG2 is byte or word alignable, it will be at 4F5AH.

Address assignment of groups, segments, and classes can be in any order, as long as addresses do not conflict with existing absolute addresses.

BOOTSTRAP

```
C>LOC86 USER\TEST.LNK & <cr>
>>START(GO) BOOTSTRAP <cr>
```

A long jump to GO will be placed at location 0FFFF0H.

COMMENTS/NOCOMMENTS

```
C>LOC86 SOURCE.LNK NOCOMMENTS <cr>
```

```
C>LOC86 TEMP.LNK COMMENTS <cr>
```

INITCODE/NOINITCODE

```
C>LOC86 FORK.LNK & <cr>
>>INITCODE (32768) <cr>
```

The initialization code is placed at address 32768 decimal (8000H).

```
C>LOC86 TEST.LNK NOINITCODE <cr>
```

No initialization code will be produced.

LINES/NOLINES

```
C>LOC86 RUN.LNK NOLINES <cr>

C>LOC86 TEST.LNK <cr>
```

LINES is the default, so it need not be specified.

MAP/NOMAP

```
C>LOC86 TESTER.LNK MAP <cr>
```

The map is placed in the file named TESTER.MP2.

```
C>LOC86 GONE.LNK TO & <cr>
>>HERMAP.OVY NOMAP <cr>
```

NAME

```
C>LOC86 SHORT.LNK NAME &<cr>
>>('THIS IS A VERY LONG MODULE') <cr>
```

OBJECTCONTROLS

```
C>LOC86 UPWARD.LNK & <cr>
>>OBJECTCONTROLS (NOLINES, & <cr>
>>NOCOMMENTS, NOSYMBOLS) <cr>
```

The statement in the above example removes all debug records from the object file, but keeps the information in the print file.

```
C>LOC86 PASCAL1.LNK & <cr>
>>OBJECTCONTROLS (PURGE, PUBLICS) <cr>
```

NOPUBLICS is implied by PURGE, but PUBLICS overrides it.

ORDER

```
C>LOC86 SPCSEG.LNK ORDER & <cr>
>>(CLASSES (CLASS1 (SEG@A, SEG@B), & <cr>
>>CLASS2), SEGMENTS & <cr>
>>(SEG1, SEG22, SEG10)) <cr>
```

SEG@A of CLASS1 will be the first relocatable segment located. SEG@B will be next, followed immediately by any other segments contained within CLASS1. The extra segments in CLASS1 (and all of the segments in CLASS2) are located in the order in which they are encountered. Finally, the list in the SEGMENTS subcontrol is handled.

PRINT/NOPRINT

```
C>LOC86 PROG.LNK TO & <cr>
>>TEMP1.TST PRINT <cr>
```

The print file for the above example is TEMP1.MP2.

```
C>LOC86 INTRUPT.LNK <cr>
```

The print file for the above example is INTRUPT.MP2.

```
C>LOC86 PROG.LNK PRINT(MAP)<cr>
```

The print file for the above example is MAP.

PRINTCONTROLS

```
C>LOC86 LINEAR.LNK & <cr>
>>PRINTCONTROLS(NOLINES) <cr>
```

Information about line numbers is removed from the print file.

```
C>LOC86 DIR1\SUBDIR\PR.LNK & <cr>
>>PRINTCONTROLS (PURGE) <cr>
```

All but the segment information is removed from the print file.

PUBLICS/NOPUBLICS

```
C>LOC86 PRIVATE.LNK NOPUBLICS<cr>
```

No public information is included in the output file and the print file.

```
C>LOC86 TEXT.LNK & <cr>
>>PUBLICS <cr>
```

All public information will be included in both the print file and the output file.

PURGE/NOPURGE

```
C>LOC86 PROJ5.LNK PURGE <cr>
```

The object file contains no public or debug information, and the symbol table does not appear in the print file.

```
C>LOC86 B0209.LNK PURGE & <cr>
>>PRINTCONTROLS (NOPURGE)<cr>
```

The line and symbol information will be kept in the print file.

RESERVE

```
C>LOC86 LOWMEM.LNK RESERVE & <cr>
>>(0F0000H TO 0FFFFFH)  <cr>
```

The control in the above example reserves a high-order 64K of memory.

```
C>LOC86 HUGOS.LNK RESERVE & <cr>
>>(00H TO 0200H, 0FFF00H TO 0FFFFFH) <cr>
```

A 200H and a 100H section of memory at the top and bottom of memory are reserved.

SEGSIZE

```
C>LOC86 GROW.LNK SEGSIZE & <cr>
>>(MEMORY(+2000))<cr>
```

The size of segment MEMORY will be increased by 2000 bytes.

```
C>LOC86 SEGPROB.LNK SEGSIZE & <cr>
>>(MYSELF(-1FFH))<cr>
```

The size of segment MYSEG will be decreased by 511 bytes.

```
C>LOC86 RPLACE.LNK SEGSIZE & <cr>
>>(XENDA(7770))<cr>
```

The new segment size for XENDA is 7770 bytes.

START

```
C>LOC86 AUTO.LNK START(IGNITION) <cr>
```

Execution of AUTO will start at IGNITION.

```
C>LOC86 HALTS.LNK START & <cr>
>>(00H, 200H) <cr>
```

HALTS will start at location 200H.

SYMBOLS/NOSYMBOLS

```
C>LOC GESHTA.LNK SYMBOLS <cr>
```

This statement will include the local symbol records in the object file and the symbol information in the print file.

```
C>LOC86 TEST.LNK PURGE <cr>
```

PURGE is a shorthand for NOSYMBOLS, NOCOMMENTS, NOPUBLICS, and NOLINES.

SYMBOLCOLUMNS

```
C>LOC86 TEST.LNK & <cr>
>>SYMBOLCOLUMNS(1) <cr>
```

```
C>LOC86 LINKED.LNK & <cr>
>>SYMBOLCOLUMNS(4) PRINT <cr>
```

The symbol table will be printed on a line printer. Each line will contain 4 columns.

OH86

```
C>OH86 FINALPRG TO FINISH.HEX <cr>
```

iAPX 86,88 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 that 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—denotes a module created by a translator, such as PL/M-86 or ASM86, and

L-MODULE—denotes a module created by LINK86 from one or more constituent modules. (Note that modules are not "created" in this sense by the iAPX86,88 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 iAPX86,88 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 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) or an R-MODULE HEADER RECORD (RHEADR). In the LHEADR or RHEADER record a name is also provided. This name is available for use to refer 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 | omod.
tmod                    =THEADR |REGINT|content  def mod_tail.
lmod                    =LHEADR |REGINT|t  component mod_tail.
omod                    =RHEADR {OVLDEF}|REGINT| o_component
                          {OVLDEF} mod  tail.
o_component             =t_component ENDREC.
t_component             =|THEADR| content_def
content_def             =PEDATA | PIDATA.
mod_tail                =|REGINT|  MODEND.
```

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.

One module may not contain more than one REGINT record and more than one OVLDEF sequence. If a REGINT record and an OVLDEF sequence exist, the REGINT record must immediately follow the OVLDEF sequence.

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, containing information not pertinent to an absolute loader. The record numbers in this category are:

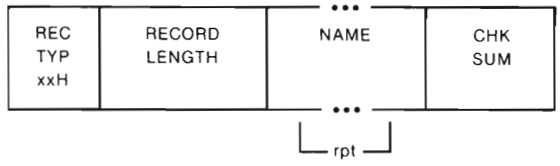
   72H, 74H, 7AH, 7CH, 7EH, 88H, 8CH, 8EH, 90H, 92H, 94H, 96H, 98H, 9AH, 9CH

2. Erroneous, containing information about relocation, indicating that the object module is not yet in absolute form or that erroneous record types exist. 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 four vertical bars in the top and bottom represent 4-byte fields. 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 two 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.

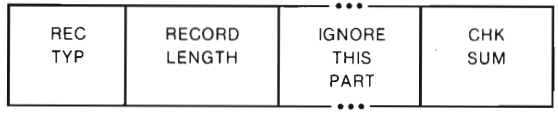
Chk Sum

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

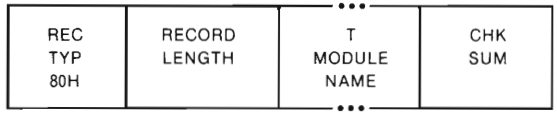
Descriptions of contents of fields will sometimes get down to the bit level. Boxes with vertical lines drawn through them represent bytes or words; the vertical lines indicate bit boundaries; thus this byte has three bit-fields of three, one, and four bits:

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 72H, 74H, 7AH, 7CH, 7EH, 88H, 8CH, 8EH, 90H, 92H, 94H, 96H, 98H, 9AH, 9CH.

T-Module Header Record (THEADR)



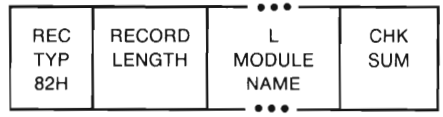
Every module output from a translator must have a T-MODULE HEADER RECORD. Its purpose is to provide the identity of the original defining module for all debug information encountered in the module up to the following T-MODULE HEADER RECORD or MODULE END RECORD.

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

T-Module Name

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

L-Module Header Record (LHEADR)



A module created by LINK86 and LOC86 may have an L-MODULE HEADER RECORD. This record serves only to identify a module that has been processed (output) by LINK86 and/or LOC86. 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.

R-MODULE HEADER RECORD (RHEADR)

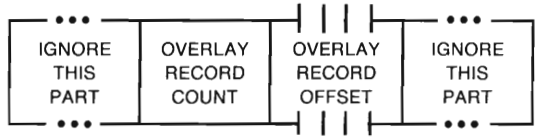
Every module with overlays created by LINK86/LOC86 will have an R-MODULE HEADER RECORD. This record serves to identify a module that has been processed (output) by LINK86/LOC86. It also specifies the overlay count and the location of the Overlay Definition records. When several modules are linked to form another module, the new module requires a name, perhaps unique from those of the linked modules, by which it can be referred to.

R-MODULE NAME

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

OVERLAY INFO

The OVERLAY INFO field provides information on overlays in the module and has the following format:



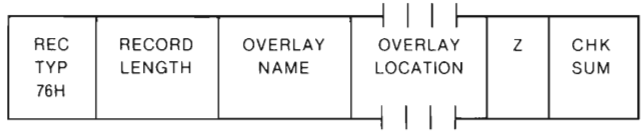
The first subfield is a 5-byte field that should be ignored.

The OVERLAY RECORD COUNT subfield indicates the number of Overlay Definition Records in the module.

The OVERLAY RECORD OFFSET subfield is a 4-byte field. It contains a 32-byte unsigned number indicating the location in bytes, relative to the start of the object file, of the first Overlay Definition Record in the Module.

The last subfield is a 16-byte field that should be ignored.

OVERLAY DEFINITION RECORD (OVLDEF)



This Record provides the overlay name, the location of the overlay in the object file.

A loader may use this record to locate the data records of the overlay in the object file.

OVERLAY NAME

The OVERLAY NAME field provides a name by which a collection of data records may be referenced for loading.

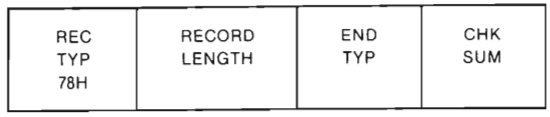
OVERLAY LOCATION

The OVERLAY LOCATION is a 4-byte field which gives the location in bytes relative to the start of the file of the first byte of the records in the overlay.

Z

The Z field is a reserved field. This field is required to be zero.

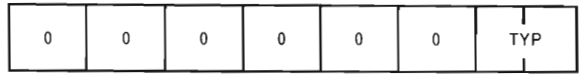
END RECORD (ENDREC)



This record is used to denote the end of a set of records such as records in an overlay.

END TYP

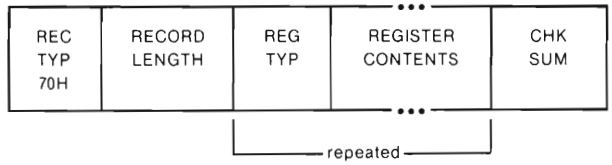
This field specifies the type of the set. It has the following format:



TYP is a two bit subfield that specifies the following types of ends:

TYP	TYPE OF END
0	End of overlay
1	(Reserved)
2	(Illegal)
3	(Illegal)

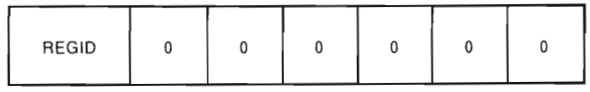
REGISTER INITIALIZATION RECORD (REGINT)



This record provides information about the 8086 registers/register-pairs: CS and IP, SS and SP, DS and ES. The purpose of this information is for a loader to set the necessary registers for initiation of execution.

REG TYP

The REG TYP field provides the register/register-pair name. It has the following format:

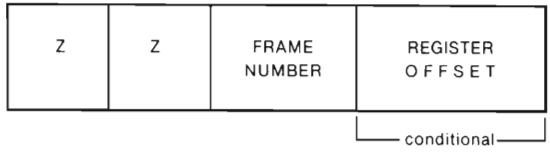


REGID is a two bit subfield that specifies the name of the registers/register-pairs as follows:

REGID	REGISTER/REGISTERPAIR
0	CS and IP
1	SS and SP
2	DS
3	ES

REGISTER CONTENTS

The REGISTER CONTENTS field has the following format:

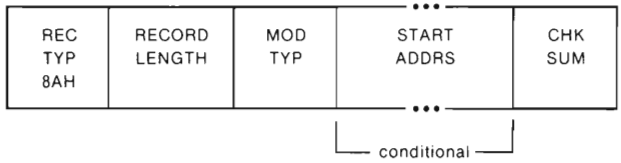


The Z fields are reserved fields. They are required to be zero.

The FRAME NUMBER field specifies a frame number that must be used to initialize the base register indicated by the REGID value.

The REGISTER OFFSET field, present only if REGID <= 1, specifies an offset relative to the FRAME. This value is appropriate for the initialization of either the IP register (REGID = 0) or the SP register (REGID = 1).

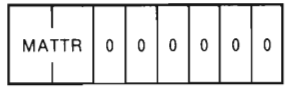
Module End Record (MODEND)



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:

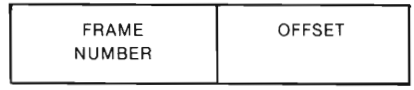


MATTR is a two-bit subfield that specifies the following module attributes:

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

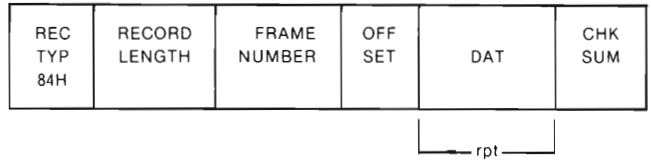
The START ADRS field has the following format:



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 that 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.

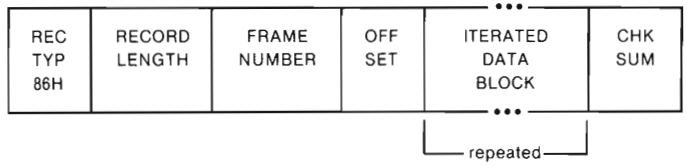
Offset

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

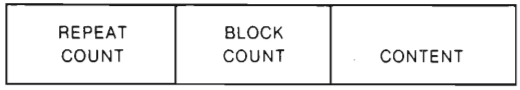
This field specifies a frame number relative to which the data bytes will be loaded.

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 than 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

RECD MARK ':'	REC LEN '02'	ZEROES '0000'	REC TYP '02'	USBA	CHK SUM

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'.

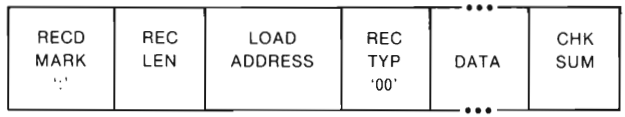
USBA

The USBA field contains four ASCII hexadecimal 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

RECD MARK ':'	REC LEN '04'	ZEROES '0000'	REC TYP '03'	CS	IP	CHK SUM

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.

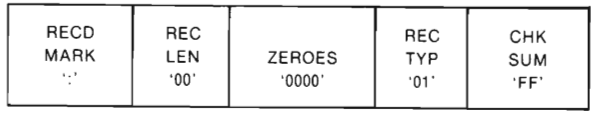
IP

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

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 defines a contiguous memory image from 00200H to 00215H.
4	PEDATA record defines a contiguous memory image from 00360H to 00377H.
5	PEDATA record defines a contiguous memory image from 00415H to 0042BH.
6	PEDATA record defines a contiguous emory image from 051620H to 0516633H.
7	PIDATA record 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 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.

Sample Absolute Hexadecimal Object File

The following is the hexadecimal object file representation of the object file given in the example above:

```
:020000020020DC
:10000000004992DB246DB6FF4891DA236CB5FE47B8
:0600100090D9226BB4FD43
:020000020036C6
:100000000062C42688EA4CAE1072D43698FA5CBE00
:080010002082E446A80A6CCE30
:020000020041BB
:10000500001D3A577491AECBE805223F5C7996B353
:07001500D0ED0A2744617ED3
:02000002516249
:1000000000850A8F14991EA328AD32B73CC146CB98
:0400100050D55ADF8E
:0200000251B0FB
:1000000041424341424341424344444444454545BF
:0E00100044444444454545444444445454524
:040000035162000541
:00000001FF
```

Hexadecimal-Decimal Conversion

B

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

The Effect of Available Memmory on LINK86, CREF86, LIB86, and LOC86

C

The system resources required by LINK86, CREF86, LIB86, or LOC86 depend on the number of symbols, modules, or segments in the input file(s). The greater the number of symbols in the input, the greater the memory requirements.

LINK86, CREF86, AND LIB86

These utilities can take advantage of up to 512K of available memory space. When the number of symbols in the input list requires more memory than is available, these utilities use disk resources to accommodate the remainder. *Available memory* means RAM which the utilities have available to them exclusively. Once a utility has run out of memory and has to use disk, performance will become impaired.

The following table defines the number of symbols or modules which these utilities may process without performance degradation, given several levels of available memory. The available memory depends on the hardware and software environment under which the utilities are running on your system. Note that the relationship between number of symbols or modules and the amount of available memory is linear, up to a maximum. The following assumptions were used to calculate the figures provided:

- Variable and module names average 10 characters.
- Each symbol has five references (CREF86).
- Each module has 1.4 public names (LIB86).
- A symbol as used here is an abstract representation of an 8086 object module format record:

	LINK86	CREF86	LIB86
Maximum number of symbols or modules which can be processed without performance penalty:			
With 100K available memory	1,700 symbols	1,900 symbols	450 modules
With 164K available memory	2,900 symbols	3,300 symbols	1,000 modules
With 228K available memory	4,200 symbols	4,700 symbols	1,700 modules
With 484K available memory	10,000 symbols	11,000 symbols	4,000 modules
Theoretical maximum number of symbols or modules, regardless of available memory:	10,000 symbols	11,000 symbols	4,000 symbols

LOC86

With 96K of available memory, LOC86 will support up to 900 segments.

LINK86 Controls and Error Messages

D

Table D-1 lists all of LINK86's control syntax, abbreviations, and default settings.

Table D-1. Summary of LINK86 Controls

Control	Abbrev.	Default
ASSIGN({ *variable*(address) } [,...])	AS	Not applicable
ASSUMEROOT(*pathname*)	AR	Not applicable
BIND	BI	NOBIND
NOBIND	NOBI	
COMMENTS	CM	COMMENTS
NOCOMMENTS	NOCM	
FASTLOAD	FL	NOFASTLOAD
NOFASTLOAD	NOFL	
GROUPOVERLAYS	GO	GROUPOVERLAYS
NOGROUPOVERLAYS	NOGO	
INITCODE	IC	Not applicable
LINES	LI	LINES
NOLINES	NOLI	
MAP	MA	MAP
NOMAP	NOMA	
MEMPOOL(*min-size* [, *max-size*])	MP	Not applicable
NAME(*module name*)	NA	Not applicable
OBJECTCONTROLS({ LINES | NOLINES | COMMENTS | NOCOMMENTS | SYMBOLS | NOSYMBOLS | PUBLICS [EXCEPT(*symbol* [,...])] | NOPUBLICS [EXCEPT (*symbol* [,...])] | TYPE | NOTYPE | PURGE | NOPURGE } [,...])	OC	Not applicable
ORDER({ *group*({ *segment* [\ *class* [\ *overlay*]] } [,...]) } [,...])	OD	Not applicable
OVERLAY[(*overlay*)]	OV	NOOVERLAY
NOOVERLAY	NOOV	
PRINT[(*pathname*)]	PR	PRINT(*object file*.MP1)
NOPRINT	NOPR	

Table D-1. Summary of LINK86 Controls (Cont'd.)

Control	Abbrev.	Default
PRINTCONTROLS({ LINES | NOLINES | COMMENTS | NOCOMMENTS | SYMBOLS | NOSYMBOLS | PUBLICS[EXCEPT(*symbol*[,...])] | NOPUBLICS [EXCEPT(*symbol* [,...])] | TYPE | NOTYPE | PURGE | NOPURGE } [,...])	PC	Not applicable
PUBLICS [EXCEPT(*symbol* [,...])]	PL [EC]	PUBLICS
NOPUBLICS [EXCEPT(*symbol* [,...])]	NOPL[EC]	
PUBLICSONLY(*pathname*[,...])	PO	Not applicable
PURGE	PU	NOPURGE
NOPURGE	NOPU	
RENAMEGROUPS({ *group* TO *group* } [,...])	RG	Not applicable
SEGSIZE({ *segment*[\ *class*[\ *overlay*]] (*min-size*[,[*max-size*]]) } [,...])	SS	Not applicable
SYMBOLS	SB	SYMBOLS
NOSYMBOLS	NOSB	
SYMBOLCOLUMNS({ 1 | 2 | 3 | 4 })	SC	SYMBOLCOLUMNS(2)
TYPE	TY	TYPE
NOTYPE	NOTY	

The following are descriptions of all LINK86 error and warning messages. The description of each message has up to four parts:

- Meaning—how to interpret the message
- Cause—the usual reason for the error or warning condition
- Effect—the state of LINK86 and the object file(s) after the message is issued
- User Action—what you can do to correct the condition

Not all these parts are given for each message. However, parts excluded are self-explanatory.

Error messages are always fatal, but warning messages are not. In the event of a warning, read the EFFECT of the warning carefully to determine whether the resulting code is valid.

Error and warning messages are displayed at the console device, but printed only if a listing would otherwise be printed.

```
ERROR 1:  I/O ERROR
  operating system error message
  FILE: pathname

ERROR 2:  I/O ERROR
  operating system error message
  FILE: pathname
```

```
ERROR 3:  I/O ERROR
  operating system error message
  FILE: pathname
```

```
ERROR 4:  CONSOLE I/O ERROR
  operating system error message
  FILE: pathname
```

Meaning

An I/O error was detected by the operating system. The error number identifies the file that caused the error:

1. The input file
2. The print file
3. The object file
4. The console file (usually the console)

Refer to the documentation for your operating system for a complete list of all possible messages.

Effect

LINK 86 immediately terminates processing, all open files are closed. The contents of the print and object files are undefined.

User Action

Correct the error and restart LINK86.

```
ERROR 5:  INPUT PHASE ERROR
  FILE: pathname
  MODULE: module name
```

Meaning

LINK86 encountered a record during the second phase of linkage that does not agree with information gathered during the first phase of linkage.

Cause

This error is caused by a data transmission error or a LINK86 error.

Effect

LINK86 immediately terminates processing and closes all open files. The contents of the print file and the object file are undefined.

User Action

Contact Intel immediately. Forward a copy of the object file, the LINK86 invocation line, and your version of LINK86.

```
ERROR 6:  CHECK SUM ERROR
  FILE: pathname
  MODULE: module name
```

Meaning

The check sum field at the end of one of the object module records indicates a transcription error.

Cause

Any one of many possible data encoding or communication errors could be at fault.

Effect

LINK86 immediately terminates processing and closes all open files. The contents of the print file and the object file are undefined.

User Action

Retranslate the source that produced the specified module and relink.

```
ERROR 7:  COMMAND INPUT ERROR
```

Meaning

LINK86 encountered an error while attempting to read the complete invocation line.

Cause

Possibly an end-of-file while reading from the console input device.

Effect

LINK86 immediately terminates processing and closes all open files. The contents of the print file and the object file are undefined.

User Action

Examine the invocation line, and reinvoke LINK86 correctly.

```
WARNING 8:  SEGMENT COMBINATION ERROR
  FILE: pathname
  MODULE: module name
  SEGMENT: segment name
  CLASS: class name
```

Meaning

Two segments with the same name have been found to be uncombinable.

Cause

The specified segments have different combination attributes or incompatible alignment attributes.

Effect

Although LINK86 will continue processing pass 1, pass 2 will not be started. The object file will be useless and the print file will contain limited information.

User Action

Retranslate the source that produced the specified file and module.

```
WARNING 9:  TYPE MISMATCH
  FILE: pathname
  MODULE: module name
  SYMBOL: symbol name
```

Meaning

LINK86 has found a public/external symbol pair for which the type definitions do not agree.

Effect

LINK86 continues processing using the first definition only. The object file and the print file should be valid, except the second definition is ignored.

User Action

Modify the public or external declaration and recompile and relink the source file.

```
WARNING 10:  DIFFERENT VALUES FOR
  FILE: pathname
  MODULE: module name
  SYMBOL: symbol name
```

Meaning

LINK86 encountered the same symbol declared public in two different modules. The specified file and module contains the second definition encountered.

Cause

Two modules have used the same symbol name for different public definitions.

Effect

LINK86 continues processing using the value of the first public definition; the second definition is ignored. Both the print file and the object file will be valid.

User Action

Change the name of the symbol in either the specified file or the file containing the earlier definition.

```
ERROR 11:  INSUFFICIENT MEMORY
  FILE: pathname
  MODULE: module name
```

Meaning

There is insufficient memory in your system for LINK86 to build its internal tables and data structures.

Cause

You are using too many public symbols.

Effect

LINK86 immediately terminates processing and closes all open files. The contents of the print file and the object file are undefined.

User Action

If expanding system memory is not possible, try incremental linkage (i.e., link smaller sets of files together using the NOPUBLICS control, then link the resulting composite modules together).

```
WARNING 12:  UNRESOLVED SYMBOLS
  FILE: pathname
  MODULE: module name
```

Meaning

There are declarations of external symbols that were not resolved during this linkage.

Cause

This is very common when performing an incremental linkage.

Effect

The print file is valid. The object file must be linked to resolve the external references.

User Action

Link object file to a file that will resolve the external references.

```
WARNING 13:  IMPROPER FIXUP
  FILE: pathname
  MODULE: module name
```

Cause

The external reference makes assumptions about the segment register that do not agree with the assumption made for the public definition.

Effect

LINK86 continues processing. The object file will not be usable, but the print file will be complete and accurate.

User Action

Depending on the cause of the error: change your ORDER control, recompile with a different model of segmentation, or change the source and reassemble.

```
WARNING 14:  GROUP ENLARGED
  FILE: pathname
  GROUP: group name
  MODULE: module name
```

Meaning

The specified group name has been defined twice in two different modules. The segments contained in the two definitions are different.

Effect

The two groups are combined into one. All segments that were in either group are included in the resulting group. Segments with the same segment name, class name, and overlay name are combined. LINK86 continues processing. Both the print file and object file are valid.

User Action

No user action should be necessary.

```
ERROR 15:  LINK86 ERROR
  FILE: pathname
  MODULE: module name
```

User Action

Contact Intel immediately. Forward a copy of the object file, the LINK86 invocation line, and your version of LINK86.

```
ERROR 16:  STACK OVERFLOW
  FILE: pathname
  MODULE: module name
```

Meaning

LINK86's run time stack used for type matching has overflowed.

Cause

The type definition of one of your symbols is overly complex.

Effect

LINK86 immediately terminates processing and closes all open files. The contents of the print file and the object file are undefined.

User Action

Try incremental linkage — if error persists, contact Intel.

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

Meaning

The combination of two or more segments has resulted in a segment that exceeds 64K.

Effect

LINK86 continues processing during the current pass, but the print and object files are not useable.

User Action

Reorganize your segments and reassemble.

```
WARNING 18:  IMPROPER START ADDRESS
  FILE: pathname
  MODULE: module name
```

Meaning

A start address was found in one of the overlay modules, and none was found in the root module.

Cause

This error is often caused by misordering the input modules in the input list.

Effect

LINK86 ignores the start address in the specified overlay module and continues processing.

User Action

If you want the module containing the start address to be the root, relink with that module first in the input list.

```
ERROR 19:  TYPE DESCRIPTION TOO LONG
  FILE: pathname
  MODULE: module name
```

Meaning

The type definition is too long to fit in LINK86's symbol table.

Effect

LINK86 immediately terminates processing and closes all open files. The contents of the print file and the object file are undefined.

User Action

Contact Intel immediately. Forward a copy of the object file, the LINK86 invocation line, and your version of LINK86.

```
WARNING 20:  NO SUCH GROUP
  NAME: group name
```

Cause

You have attempted to rename a nonexistent group.

Effect

LINK86 ignores the RENAME control and continues processing.

User Action

Reinvoke LINK86 with the correct invocation line.

```
WARNING 21:  RENAME ERROR
  NAME: name
```

Meaning

The new group name specified is the same as an existing group.

Effect

The group is not renamed. LINK86 continues as if the rename control was not given.

User Action

Reinvoke LINK86 with the correct invocation line.

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

Cause

This is usually the result of a typo in the invocation line. The partial command tail up to the point where the error was detected is printed.

Effect

LINK86 terminates processing and closes all open files. The contents of the print file and the object file are undefined.

User Action

Reinvoke LINK86 more carefully this time.

```
ERROR 23:  BAD OBJECT FILE
  FILE: pathname
  MODULE: module name
```

Meaning

LINK86 has discovered an inconsistency in the fields of a record in the specified input file.

Cause

This could be an error by the translator or a data transmission error.

Effect

LINK86 immediately terminates processing and closes all open files. The contents of the print file and the object file are undefined.

User Action

Retranslate the source file. If the problem persists contact Intel.

```
WARNING 24:  CANNOT FIND MODULE
  FILE: pathname
  MODULE: module name
```

Meaning

The specified module cannot be found in the specified library file.

Effect

LINK86 continues processing as if the specified module was not in the list.

User Action

If the module is important, you can link it into the output object file later.

```
WARNING 25:  EXTRA START ADDRESS IGNORED
  FILE: pathname
  MODULE: module name
```

Meaning

LINK86 has encountered a start address in more than one module.

Cause

This will occur any time you specify more than one main module in the input list.

Effect

LINK86 uses the start address encountered earlier and ignores the start address in the specified module. LINK86 continues processing with no other side effects.

User Action

None, if the start address in the specified module was intended to be ignored.

```
ERROR 26:  NOT AN OBJECT FILE
  FILE: pathname
```

Meaning

The specified file is not an object file.

Cause

This is usually the result of a typo when entering. However, certain data transmission errors can also cause this error.

Effect

LINK86 terminates processing and closes all open files.

User Action

Reinvoke LINK86 typing the line more carefully. If error resulted from a data transmission error, retranslate and then relink.

```
ERROR 27:  OPERATING SYSTEM INTERFACE ERROR
  FILE: pathname
```

Effect

LINK86 immediately terminates processing and closes all open files. The contents of the print file and the object file are undefined.

User Action

Refer to the documentation for your operating system. If you cannot correct the error condition, contact Intel; forward a copy of the object file, the LINK86 invocation line, and your version of LINK86.

```
WARNING 28:  POSSIBLE OVERLAP
  FILE: pathname
  MODULE: module name
  SEGMENT: segment name
  CLASS: class name
```

Meaning

LINK86 issues this warning when it combines two absolute segments.

Effect

LINK86 continues processing with no side effects.

User Action

If there is a conflict LOC86 or the loader will detect the overlap.

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

Meaning

This error occurs if the public symbol corresponding to an external reference has been specified by its absolute address, and the address does not reside in any segment.

Effect

LINK86 continues processing and the print and object files will be valid except the external reference has not been properly resolved.

User Action

Either remove the reference to the public symbol or do not allow the symbol to be absolute.

```
ERROR 30:  LIBRARY IS NOT ALLOWED WITH PUBLICSONLY CONTROL
  FILE: pathname
```

Meaning

The specified file is a library and libraries are not allowed in a PUBLICSONLY control.

Effect

LINK86 immediately terminates processing and closes all open files. The contents of the print file and the object file are undefined.

User Action

Remove library file from PUBLICSONLY argument list and reinvoke LINK86.

```
WARNING 31:  REFERENCED LOCATION OFFSET UNDERFLOW
  FILE: pathname
  MODULE: module name
```

Meaning

While computing the offset for an 8089 self relative reference, LINK86 had a negative result.

Cause

Either with the ORDER control or the order of files in the input list, the reference was separated from its target, or the 8089 segment is too large.

Effect

LINK86 continues processing; however, the invalid offset computation is used.

User Action

Examine the ORDER control in the invocation line and modify its arguments. Reinvoke LINK86 carefully.

```
WARNING 32:  EXTRA REGISTER INITIALIZATION RECORD IGNORED
  FILE: pathname
  MODULE: module name
```

Cause

You have included two main modules in your input list.

Effect

LINK86 uses the first register initialization record and ignores the second. Processing continues.

User Action

If the register initialization information in the specified file and module should be used, then modify your input list; otherwise, no user action is necessary.

```
ERROR 33:  ILLEGAL USE OF OVERLAY CONTROL
  FILE: pathname
  MODULE: module name
```

Meaning

LINK86 has found an overlay definition in the specifed file and module, while processing input modules for an overlay.

Effect

LINK86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Remove the specified file from the input list and relink.

```
ERROR 34:  TOO MANY OVERLAYS IN INPUT FILE
  FILE: pathname
  MODULE: module name
```

Meaning

The specified file and module above contains more than one overlay definition.

Effect

LINK86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Remove the specified file from the input list and relink.

```
ERROR 35:  SAME OVERLAY NAME IN TWO OVERLAYS
  FILE: pathname
  MODULE: module name
  NAME: name
```

Meaning

The specified file contains an overlay that has the same name as an overlay encountered in the input list.

Effect

LINK86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Remove one of the duplicate names from the input list and relink. If both overlays are necessary, relink one overlay specifying a different overlay name.

```
ERROR 36:  ILLEGAL OVERLAY CONSTRUCTION
  FILE: pathname
  MODULE: module name
```

Meaning

Some of the modules in the input list contain overlay definitions while others do not. This is not permitted — all modules in the input list must be the same with respect to overlays.

Effect

LINK86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Remove the non-overlay files and relink.

```
WARNING 37:  DIFFERENT PUBLICS FOR EXTERNAL IN ROOT
  FILE: pathname
  MODULE: module name
```

Meaning

LINK86 has found two symbol definitions in the overlay modules that resolve an external symbol definition in the root.

Effect

LINK86 ignores the definition in the specified file and module, and continues processing with no side effects.

User Action

Remove the unwanted symbol definition and relink.

```
ERROR 38:  INVALID OVERLAPPING GROUPS
  FILE: pathname
  MODULE: module name
  SEGMENT: segment name
  GROUP: group name
```

Meaning

While binding the input list LINK86 found a segment that was defined to be within two groups.

Effect

LINK86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Either modify the source to remove the segment from one of the groups or do not link with the BIND control.

```
ERROR 39:  SPECIFIED GROUP NOT FOUND IN INPUT MODULE
  GROUP: group name
```

Cause

Often this is the result of a typographical error in the invocation line.

Effect

LINK86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Correct the invocation line and relink.

```
ERROR 40:  SPECIFIED SEGMENT NOT FOUND IN THE GROUP
  SEGMENT: segment name
  GROUP: group name
```

Cause

Usually this is the result of a typographical error in the ORDER control.

Effect

LINK86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Correct the invocation line and relink.

```
ERROR 41:  SPECIFIED SEGMENT NOT FOUND IN INPUT MODULE
  SEGMENT: segment name
  CLASS: class name
```

Cause

Usually this is the result of a typographical in the SEGSIZE control.

Effect

LINK86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Find the module that contains the specified segment and add it to the input list.

```
WARNING 42:  DECREASING SIZE OF SEGMENT
  SEGMENT: segment name
```

Meaning

The size change specified in SEGSIZE has caused LINK86 to decrease the size of the specified segment.

Effect

Decreasing the size of a segment can cause sections of code to be unaccounted for during the memory allocation process. LINK86 continues processing with no side effects.

User Action

None if the size decrease was intended.

```
ERROR 43:  SEGMENT SIZE OVERFLOW; OLD SIZE + CHANGE > 64K
  SEGMENT: segment name
  CLASS: class name
```

Meaning

The size change specified in the SEGSIZE control caused the segment to become greater than 64K.

Effect

LINK86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Reinvoke LINK86 with the correct SEGSIZE control.

```
ERROR 44:  SEGMENT SIZE UNDERFLOW; OLD SIZE + CHANGE < 0
  SEGMENT: segment name
  CLASS: class name
```

Meaning

The size change specified in the SEGSIZE control caused the segment's size to be less than zero.

Effect

LINK86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Reinvoke LINK86 with the correct SEGSIZE control.

```
ERROR 45:  THE SEGMENT MAXIMUM SIZE IS LESS THAN THE
           SEGMENT MINIMUM SIZE
  SEGMENT: segment name
  CLASS: class name
```

Cause

Usually this is the result of a typographical error in the SEGSIZE control.

Effect

LINK86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Correct the invocation line and relink.

```
ERROR 46:  ILLEGAL USE OF SEGSIZE CONTROL
  SEGMENT: segment name
  CLASS: class name
```

Cause

A maximum size was specified for either a stack segment, an absolute segment, or a segment that is not the highest component of its group.

Effect

LINK86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Remove the specified segment from the SEGSIZE control and relink.

```
WARNING 47:  GROUP HAS NO CONSTITUENT SEGMENTS
  GROUP: group name
```

Meaning

The group has no segments and is not placed in the output object file.

Cause

Often this is the result of a typographical error in the invocation line.

Effect

LINK86 does not place the specified group in the object file and continues processing with no side effects.

User Action

Unless there is some particular need for the specified group, no user action is necessary.

```
WARNING 48:  SIZE OF GROUP EXCEEDS 64K
  GROUP: group name
```

Meaning

All of the segments that belong to the specified group do not fit within the physical segment defined for that group.

Cause

This error is usually caused by misuse of the SEGSIZE or ORDER controls.

Effect

LINK86 includes all segments in the object file and continues processing the input module. The output module will be executable, although addressing errors may occur.

User Action

Examine the invocation line and reinvoke LINK86 using the SEGSIZE or ORDER control more carefully.

```
WARNING 49:  MAXIMUM SIZE OF GROUP EXCEEDS 64K
  GROUP: group name
```

Meaning

The maximum segment size for the segments contained in the specified group exceeds 64K.

Cause

This error is usually caused by misuse of the SEGSIZE control.

Effect

LINK86 reduces the maximum size of the group and its constituent segments. LINK86 continues processing the input module. The output module will be executable.

User Action

No action is necessary. If you want to remove the error, examine the invocation line and reinvoke LINK86 using the SEGSIZE control more carefully.

```
WARNING 50:  MORE THAN ONE SEGMENT WITH THE MEMORY
             ATTRIBUTE
  SEGMENT: segment name
```

Meaning

After the first memory segment is found, LINK86 issues this warning each time it finds a segment with the memory attribute.

Effect

LINK86 ignores the memory attribute on the segment specified in the message. Processing continues with LINK86 treating the additional memory segment as just another segment.

User Action

Depending on your intentions, this message may be ignored or you may wish to change the segment definition and relink.

```
WARNING 51:  SEGMENT WITH MEMORY ATTRIBUTE NOT PLACED
             HIGHEST IN MEMORY
  SEGMENT: segment name
```

Meaning

The specified memory segment was not located at the highest offset in its group.

Cause

This can only occur when you explicitly request this organization through the ORDER control.

Effect

Since this can only occur by user request, LINK86 continues processing without side effects.

```
WARNING 52:  OFFSET FIXUP OVERFLOW
  FILE: pathname
  MODULE: module name
```

Meaning

While computing an offset from a base, LINK86 found that the offset was greater than 64K.

Cause

One of the segments of a group is outside the 64K frame of reference defined by its group base.

Effect

LINK86 continues processing. The print file will be valid, but the output file with regard to the out of place segment will not be usable.

User Action

Modify the group definitions in your source and retranslate.

```
WARNING 53:  OVERFLOW OF LOW BYTE FIXUP VALUE
  FILE: pathname
  MODULE: module name
```

Meaning

An 8-bit displacement value, when calculated, exceeded 255.

Cause

This type of error often occurs when a page resident segment crosses a page boundary.

Effect

LINK86 continues processing. The contents of both the print file and the object file will be valid. However, the fixup value will remain invalid.

User Action

Organize your segments so that the addressing error will not be encountered.

```
ERROR 54:  ILLEGAL USE OF ORDER CONTROL
  GROUP: group name
```

Meaning

The specified group's segments have already been ordered.

Cause

You are attempting to relink a file that has been linked with the BIND control.

Effect

LINK86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Relink using the unbound input modules.

```
ERROR 55:  ILLEGAL FIXUP
  FILE: pathname
  MODULE: module name
```

Meaning

While processing a fixup record, LINK86 found that the base for the reference and target are different.

Cause

This is usually a coding error.

Effect

LINK86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Examine your assembly language source and retranslate.

```
ERROR 56:  DATA ADDRESS OUTSIDE SEGMENT BOUNDARIES
  FILE: pathname
  MODULE: module name
  SEGMENT: segment name
```

Meaning

One of the data records associated with the specified segment contains an address outside of the segment's boundary.

Cause

This error can occur when you decrease the size of a segment.

Effect

LINK86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Change SEGSIZE control and relink.

```
ERROR 57:  MAXIMUM DYNAMIC STORAGE LESS THAN MINIMUM
           DYNAMIC STORAGE
```

Meaning

The size change specified in MEMPOOL has caused the maximum dynamic storage to be less than the minimum dynamic storage.

Effect

LINK86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Reinvoke LINK86 with correct arguments to MEMPOOL.

```
WARNING 58:  NO START ADDRESS SPECIFIED IN INPUT MODULES
```

Meaning

The BIND control was specified, and none of the input modules has a start address.

Cause

The input list contains no main module.

Effect

The CS and IP registers remain uninitialized, and their values are dependent on your system loader. The object module will be valid.

User Action

Reinvoke LINK86 with a main module or execute LOC86 with the START control.

```
ERROR 59:  I/O ERROR WITH ROOT-FILE IN ASSUMEROOT CONTROL
  FILE: pathname
  operating system message
```

Meaning

The ASSUMEROOT control was specified, but the root file identified by *pathname* in the invocation could not be accessed.

Effect

LINK86 immediately terminates processing.

User Action

Refer to your operating system documentation to correct the condition, then reinvoke LINK86.

```
ERROR 60:  OUTPUT FILE IS SAME AS INPUT FILE
  FILE: pathname
```

Meaning

LINK86 detected an output pathname identical to an input pathname.

Cause

The pathnames of the specified input file and the output file were identical.

Effect

LINK86 terminates processing immediately.

User Action

Reinvoke LINK86 after fixing the duplicate-name situation.

```
ERROR 61:  ROOT-FILE IN ASSUMEROOT CONTROL IS NOT PROPER
           OBJECT FILE
  FILE: pathname
```

Meaning

The ASSUMEROOT control was specified, but the root file is not found to have an overlay record in it.

Cause

The root file needs an overlay record.

Effect

LINK86 terminates processing immediately.

User Action

Relink the root file using the OVERLAY control.

```
WARNING 62:  ASSUMEROOT CONTROL MEANINGFUL ONLY WITH
             OVERLAYS
```

Meaning

The ASSUMEROOT control should be used only when the input modules do not contain overlay records.

Cause

ASSUMEROOT was specified, but not in conjunction with the OVERLAY control.

Effect

LINK86 ignores the ASSUMEROOT control. The object code is valid.

User Action

Reinvoke LINK86, using the OVERLAY and ASSUMEROOT controls.

```
WARNING 63:  BAD SEGMENT ALIGNMENT
  FILE: pathname
  MODULE: module name
  SEGMENT: segment name
```

Meaning

The segment is not paragraph/page-aligned.

Cause

The object code has references to the base of the specified segment, and the segment is not declared as paragraph/page-aligned.

Effect

Although the object module will be valid, the loader may not load the program correctly.

User Action

Declare the specified segment to be paragraph/page-aligned.

```
WARNING 64:  PUBLIC SYMBOLS NOT SORTED DUE TO INSUFFICIENT
             MEMORY
```

Meaning

The amount of memory required to sort the public symbols for the LINK86 print file listing is insufficient.

Cause

The number of public symbols in the input-list modules is too large for LINK86 to sort with the available memory resources.

Effect

The LINK86 print file listing provides public symbols in the order in which they were encountered in the input files. This condition has no effect on the correctness or validity of the output module.

User Action

Increase the amount of available RAM or decrease the number of public symbols.

```
WARNING 65:  ILLEGAL FIXUP:  INCORRECT DECLARATION OF
             EXTERNAL SYMBOL
  FILE: pathname containing external declaration
  MODULE: name of module containing external declaration
  SYMBOL: name of external symbol
  FRAME: identification of reference location
  TARGET: identification of target location
```

Meaning

The declaration of the specified SYMBOL was found to be inconsistent with a corresponding public symbol definition, and LINK86 could not resolve the reference.

Cause

This condition may exist for several reasons. The modules containing the external and public symbols may have been compiled under different translator controls (e.g., SMALL, LARGE). In the case of assembly language programs, the SYMBOL may be defined in a group, segment, or frame different from that in which it is declared as external. Or an attempt has been made to access absolute entry points from pre-located code without using the PUBLICSONLY control explicitly.

Effect

LINK86 internally converts these illegal fixups to legal formats to identify all occurrences in a single execution. Thus the output object module may not be correct, although it will be a valid 8086 object module.

User Action

If the warning occurred because of an attempted access of absolute entry points from pre-located code, use the PUBLICSONLY control in conjunction with the file that contains public definitions for those entry points. Otherwise, use the FRAME and TARGET information given in the warning message to pinpoint the source of the error, then correct the code.

For example:

```
WARNING 65:  ILLEGAL FIXUP:  INCORRECT DECLARATION OF
             EXTERNAL SYMBOL
  FILE:  EXTFIL
  MODULE:  EXTMODULE
  SYMBOL:  EXTSYM
  FRAME:  GROUP - DGROUP
  TARGET:  SEGMENT - CODE
```

The symbol EXTSYM is declared to be in SEGMENT5. The external-public resolution specified that the calculations be made with respect to the base of GROUP1, but the segment SEGMENT5 is not in GROUP1.

WARNING 66: CS AND IP REGISTERS ARE NOT INITIALIZED

Meaning

The INITCODE control was specified, and the register initialization record does not contain information for initialization of 8086 registers. CS means code segment register, and IP means instruction pointer.

Cause

This condition is usually the result of an incomplete END directive in your assembly language module or a translation error.

Effect

The values of CS and IP at the beginning of program execution are entirely dependent on the loader of your system. The object code will be valid.

User Action

Retranslate your code, then reinvoke LINK86.

WARNING 67: SS AND SP REGISTERS ARE NOT INITIALIZED

Meaning

The INITCODE control was specified, and the register initialization record does not contain information for initialization of 8086 registers. SS means stack segment and SP means stack pointer.

Cause

This condition is usually the result of an incomplete END directive in your assembly language module or a translation error.

Effect

The values of SS and SP at the beginning of program execution are entirely dependent on the loader of your system. The object code will be valid.

User Action

Correct your code if necessary, then reinvoke LINK86.

WARNING 68: DS REGISTER NOT INITIALIZED

Meaning

The INITCODE control was specified, and the register initialization record does not contain information for initialization of the 8086 DS (data segment) register.

Cause

This condition is usually the result of an incomplete END directive in your assembly language module or a translation error.

Effect

The value of the DS register at program execution is entirely dependent on the loader of your system.

User Action

Correct your code if necessary, then reinvoke LINK86.

WARNING 69: OVERLAPPING DATA RECORDS

Meaning

The FASTLOAD control was specified, and two data records belonging to the same segment have offsets which make them overlapping.

Cause

This warning is usually the result of a translation error, unless you have intentionally overlapped data records.

Effect

LINK86 ignores the second record and does not include it in the output file. The code will be unusable.

User Action

If you want an overlap condition to exist, reinvoke, but do not use the FASTLOAD control. Otherwise, retranslate, then reinvoke LINK86.

WARNING 70: INITCODE CONTROL INEFFECTIVE WITH BIND CONTROL

Meaning

The INITCODE and BIND controls were combined in one invocation statement.

Effect

The INITCODE control will be ignored by LINK86.

User Action

Do not invoke LINK86 using both of these controls at the same time. To invoke them separately, use the INITCODE control first, then the BIND control during a second invocation.

```
WARNING 71:  TOO MANY MAIN MODULES IN INPUT
  FILE: pathname
  MODULE: module name
```

Meaning

LINK86 discovered two or more main modules (modules with start addresses) in the input list.

Cause

The input list contains too many main modules.

Effect

LINK86 uses the start address of the first main module it reads and ignores the others. The object code will be valid.

User Action

Ensure that the LINK86 interpretation is suitable to your objectives. If not, modify the input list, and reinvoke LINK86.

```
WARNING 72:  REGISTER INITIALIZATION CODE EXISTS, NEW
             INITIALIZATION IGNORED
  FILE: pathname
  MODULE: module name
```

Meaning

Two or more initialization codes for 8086 registers were encountered in the input list.

Cause

This condition resulted from a translation or linkage problem.

Effect

LINK86 uses the first initialization code and ignores the others. The object code will be valid.

User Action

If retranslating or relinking does not correct the error, contact Intel.

```
WARNING 73:  INITCODE CONTROL INEFFECTIVE WITH OVERLAYS
```

Meaning

Both INITCODE and OVERLAY controls were specified.

Effect

The INITCODE control is ignored. The object code will be valid.

User Action

Reinvoke LINK86, using the INITCODE control. In a second invocation, specify the OVERLAY control.

```
ERROR 74:  PRINT FILE SAME AS INPUT FILE
  FILE: pathname
```

Meaning

The pathnames of the print file and one of the input files are identical.

Effect

LINK86 terminates processing immediately.

User Action

Reinvoke LINK86 after fixing the duplicate-name situation.

```
ERROR 75:  PRINT FILE SAME AS OUTPUT FILE
```

Meaning

The names of the print and output files are identical.

Cause

The invocation line included duplicate names.

Effect

LINK86 terminates processing immediately.

User Action

Correct the invocation line and reinvoke LINK86.

```
WARNING 76:  BASE OF REFERENCED SEGMENT DIFFERS FROM BASE
             OF CONTAINING GROUP
  FILE: pathname
  GROUP: group name
  MODULE: module name
  SEGMENT: segment name
```

Meaning

An assembly language reference to the base of the specified segment in the specified group exists. However, the specified segment is not the first segment in the group. This warning occurs only when BIND is in effect.

Cause

Unless you have deliberately created this reference, this warning is most likely the result of an incorrect ASSUME directive or an incorrect OFFSET operator specification.

Effect

LINK86 will process the specified reference to the segment base rather than to the group base. The output module will be valid.

User Action

If the reference to the segment base was deliberate, continue debugging your assembly language code as planned. Otherwise, check the correctness of the code, particularly the ASSUME directives and OFFSET operator specifications; then reassemble and relink.

```
WARNING 77:  REFERENCED OFFSET IN SEGMENT DIFFERS FROM
             OFFSET FROM GROUP BASE
  FILE: pathname
  GROUP: group name
  MODULE: module name
  SEGMENT: segment name
```

Meaning

An assembly language reference to an offset from the base of the specified segment in the specified group exists. However, the specified segment is not the first segment in the group. This warning occurs only when BIND is in effect.

Cause

Unless you have deliberately created this reference, this warning is most likely the result of an incorrect ASSUME directive or an incorrect OFFSET operator specification.

Effect

LINK86 will process the specified reference as an offset from the segment base rather than the group base. The output module will be valid.

User Action

If the reference to the offset in the segment was deliberate, continue debugging your assembly language code as planned. Otherwise, check the correctness of the code, particularly the ASSUME directives and OFFSET operator specifications; then reassemble and relink.

CREF86 Controls and Error Messages

E

Table E-1 lists all of CREF86's control syntax, abbreviations, and default settings.

Table E-1. Summary of CREF86 Controls

Control	Abbrev.	Default
PAGELENGTH(*number*)	PL	PAGELENGTH(60)
PAGEWIDTH(*number*)	PW	PAGEWIDTH(120)
PRINT (*pathname*)	PR	PRINT (*first input file .CRF*)
TITLE(*character-string*)	TT	Not applicable

The following are descriptions of all CREF86 error and warning messages. The description of each message has up to four parts:

- Meaning—how to interpret the message
- Cause—the usual reason for the error or warning condition
- Effect—the state of CREF86 and the object file(s) after the message is issued
- User Action—what you can do to correct the condition

Not all these parts are given for each message. However, parts excluded are self-explanatory.

Error messages are always fatal, but warning messages are not.

Error and warning messages are displayed at the console device, but printed only if a listing would otherwise be printed.

```
ERROR 1:  I/O ERROR
  operating system message explaining the cause of this erorr
  FILE: pathname
```

Meaning

An I/O error was detected. See the appropriate operating system documentation for interpretation.

Effect

CREF86 immediately terminates processing; all open files are closed. The contents of the print file are undefined.

User Action

Correct the error and restart CREF86.

```
ERROR 2:  SYNTAX ERROR IN INPUT COMMAND
```

Meaning

An error in the syntax of the invocation line was detected.

Cause

This condition is usually the result of a typographical error or transposition.

Effect

The invocation command line, to the point it is parsed, is written to the console with a # following this string.

User Action

Correct the syntactic error and retransmit the invocation line(s).

```
ERROR 3:  OUT OF MEMORY
```

Meaning

CREF86 does not have enough memory to create its internal data structures, tables, etc. This condition may also occur because of inadequate disk space for temporary files.

Cause

The input list contains too many symbols and/or too many references among them.

Effect

CREF86 immediately terminates processing, closing all open files. The contents of the print file are undefined.

User Action

Ensure that adequate resources are available to run CREF86.

```
ERROR 4:  I/O ERROR
  operating system error message
  FILE: pathname
```

Meaning

An I/O error was detected. See the appropriate operating system documentation for interpretation.

Effect

CREF86 processing is immediately terminated.

User Action

Correct the error and restart CREF86.

```
ERROR 5:  IMPROPER OBJECT MODULE
  FILE: pathname
  MODULE: module name
```

Meaning

The specified module does not meet 8086 object module requirements.

Cause

This condition may be caused by the translator or by an error in data transmission.

Effect

CREF86 processing is immediately terminated.

User Action

Try retranslating the source file. If the problem persists, call Intel.

```
ERROR 6:  PREMATURE EOF
  FILE: pathname
```

Meaning

CREF86 expects more input data, but encounters an end-of-file (EOF) condition.

Cause

This condition usually results from a translator error.

Effect

CREF86 processing is immediately terminated.

User Action

Return to the previous step in program development, then retranslate or relink.

```
ERROR 7:  LIBRARY SEEK ERROR
  FILE: pathname
  MODULE: module name
```

Meaning

CREF86 did not encounter a proper library record when scanning a library file.

Cause

The library file or the disk may be corrupted.

Effect

CREF86 immediately terminates processing.

User Action

Reinvoke CREF86 after replacing the file or the disk.

```
ERROR 8:  LIBRARY IN OVERLAY MODE
  FILE: pathname
```

Meaning

An input list contains object file(s) with an overlay record count greater than zero and a library file.

Cause

Libraries cannot contain overlay records. CREF86 can process either all modules or no modules with overlay records.

Effect

CREF86 immediately terminates processing.

User Action

Reinvoke CREF86 using a valid input list.

```
ERROR 9:  IMPROPER MODULE SEQUENCE
  FILE: pathname
  MODULE: module name
```

Meaning

A combination of modules containing overlay records with those containing nonoverlay records was encountered in the input list.

Cause

CREF86 can process input lists consisting of either all modules with overlay records or no modules with overlay records.

Effect

CREF86 immediately terminates processing.

User Action

Reinvoke CREF86 with a valid input list.

```
ERROR 10:  MORE THAN 255 OVERLAYS NOT SUPPORTED
```

Meaning

The input list contains over 255 files with overlay records.

Cause

CREF86 does not support more than 255 overlay files. In the case of input lists without overlays, however, there is no limit (except available memory) on the number of files CREF86 can process.

Effect

CREF86 immediately terminates processing.

User Action

Reinvoke CREF86 using fewer than 255 overlay files.

```
ERROR 11:  TOO MANY OVERLAYS
  FILE: pathname
  MODULE: module name
```

Meaning

The input file contains more than one overlay.

Cause

CREF86 can support files with only one overlay record each.

Effect

CREF86 terminates processing immediately.

User Action

Reinvoke CREF86 with an input list containing files with no more than one overlay each.

ERROR 12: I/O ERROR
operating system error message
FILE: *pathname*

Meaning

An I/O error was detected. See the appropriate operating system documentation for interpretation.

Effect

CREF86 terminates processing immediately.

User Action

Correct the error and restart CREF86.

ERROR 13: IMPROPER PAGE WIDTH SPECIFICATION

Meaning

The PAGEWIDTH control specification includes a *number* outside the valid syntactic range.

Effect

CREF86 terminates processing immediately.

User Action

Correct the syntax error and reinvoke CREF86. CREF86 accepts a PAGEWIDTH *number* in decimal form from 80 to 132, inclusive, in the following format:

PAGEWIDTH(*number*)

```
ERROR 14:  IMPROPER PAGE LENGTH SPECIFICATION
```

Meaning

The PAGELENGTH control specification includes a *number* outside the valid syntactic range.

Effect

CREF86 terminates processing immediately.

User Action

Reinvoke CREF86 using the proper PAGELENGTH syntax. CREF86 accepts a PAGELENGTH *number* in decimal form from 10 through 255, in the following format:

```
PAGELENGTH(number)
```

```
ERROR 15:  ILLEGAL LIBRARY FILE
  FILE: pathname
```

Meaning

CREF86 did not encounter a proper library record in the proper location.

Cause

The library file or disk may be corrupted.

Effect

CREF86 terminates processing immediately.

User Action

Reinvoke CREF86 after replacing the file or the disk.

```
ERROR 16:  IMPROPER OBJECT FILE
  FILE: pathname
  MODULE: module name
```

Meaning

CREF86 did not encounter an 8086 object module record in the proper location.

Cause

The object file may be corrupt or the file may not be an 8086 object file.

Effect

CREF86 terminates processing immediately.

User Action

Determine whether the integrity of the object file is intact and whether the file is a proper input file for CREF86. Reinvoke CREF86 with a valid and usable object file.

```
ERROR 17:  OUTPUT FILE SAME AS INPUT FILE
  FILE: pathname
  MODULE: module name
```

Meaning

CREF86 detected an output pathname identical to an input pathname.

Cause

The invocation line specified two identical pathnames.

Effect

CREF86 terminates processing immediately.

User Action

Reinvoke CREF86 after fixing the duplicate-name situation.

```
ERROR 18:  CREF86 INTERNAL ERROR
  FILE: pathname
  MODULE: module name
```

User Action

Contact Intel immediately. Forward a copy of the object file, the CREF86 invocation line, and your version of CREF86.

```
WARNING 19:  TYPE MISMATCH
  FILE: pathname
  MODULE: module name
  SYMBOL: symbol name
```

Meaning

CREF86 detected a type mismatch between two symbols with the same name.

Cause

Two symbols are declared to have identical names but different types, and the symbols are not in different overlay modules.

CREF86 does not check the entire TYPE declaration for any given symbol. For example, dimension values for arrays, number of parameters in procedure calls, etc. are not compared. Only simple types (e.g., byte, word, structure) are checked.

Effect

CREF86 flags the condition in the cross-reference listing.

User Action

Ensure that the condition is not damaging to your programming objectives.

```
WARNING 20:  SPECIFIED MODULE NOT FOUND
  FILE: pathname
  MODULE: module name
```

Meaning

A module explicitly included in the input list of the invocation is not found by CREF86.

Cause

The specified module is not part of the file specified by the pathname.

Effect

CREF86 continues processing the modules it is able to find.

User Action

Determine why the module is missing, then reinvoke CREF86.

```
ERROR 21:  OPERATING SYSTEM INTERFACE ERROR
  operating system error message
```

Meaning

CREF86 cannot open its temporary file.

Effect

CREF86 terminates processing immediately.

User Action

Refer to the documentation on the operating system to help diagnose any possible operating system malfunction.

LIB86 Commands and Error Messages

F

The table below shows all of LIB86's commands.

Table F-1. Summary of LIB86 Commands

Command	Abbrev.	Description
ADD {*pathname*[(*module name*[,...])]} [,...] TO *pathname*	A	Adds modules to a library
CREATE *pathname*	C	Creates a library file
DELETE *pathname*(*module name* [,...])	D	Deletes modules from a library file
EXIT	E	Terminates session with LIB86
LIST {*pathname*[(*module name* [,...])]} [,...] [TO *pathname*] [PUBLICS]	L [P]	Lists modules contained in a library file, and optionally lists all publics

The following are descriptions of all LIB86 error and warning messages. The description of each message has up to four parts:

- Meaning—how to interpret the message
- Cause—the usual reason for the error or warning condition
- Effect—the state of LIB86 and the object file(s) after the message is issued
- User Action—what you can do to correct the condition

Not all these parts are given for each message. However, parts excluded are self-explanatory.

Error and warning messages are displayed at the console device.

```
MODULE NOT FOUND
MODULE: module name
FILE: pathname
```

Meaning

The specified module could not be found in the specified library.

Cause

There is a typographical error in the command line.

Effect

LIB86 ignores the module in the list and continues processing.

User Action

No user action is necessary.

RIGHT PARENTHESIS EXPECTED
partial command tail

LEFT PARENTHESIS EXPECTED
partial command tail

INVALID MODULE NAME
partial command tail

MODULE NAME TOO LONG
partial command tail

INVALID SYNTAX
partial command tail

'TO' EXPECTED
partial command tail

Meaning

All of the above errors are syntax errors. For each of the above errors LIB86 issues the associated error message and displays the partial command up to the point of the error.

Cause

There is a typographical error in the command line.

Effect

LIB86 immediately terminates processing the command, displays the error message, and issues the prompt character (*).

User Action

Examine the command line, make the necessary corrections and reissue the command.

UNRECOGNIZED COMMAND

Cause

You mistyped a command (ADD, CREATE, DELETE, EXIT, or LIST).

Effect

LIB86 immediately terminates processing the command, displays the error message, and issues the prompt character (*).

User Action

Examine the command line and enter the corrected command.

INSUFFICIENT MEMORY

Meaning

There is not enough memory available to execute the command.

Cause

Exceptionally long and complex commands can cause this error.

Effect

LIB86 immediately terminates processing the command, displays the error message, and issues the prompt character (*).

User Action

Simplify your command line and reexecute.

COMMAND LINE TOO LONG
partial command tail

Meaning

The length of the LIB86 command you tried to execute exceeded the size limit of the system's command buffer.

Effect

LIB86 immediately terminates processing the command, displays this error message plus the portion of the command it would accept, then issues the prompt character (*).

User Action

Simplify your command line and reexecute.

LIB86 ERROR

Meaning

LIB86 failed an internal consistency check.

Effect

LIB86 immediately terminates processing. The results of previous commands on the library being manipulated when this error occurred may have been lost.

User Action

Contact Intel. Forward a copy of the libraries and object files used during the session in which the error occurred.

FILE ALREADY EXISTS
FILE: *pathname*

Meaning

The file specified in the CREATE command already exists.

Effect

LIB86 immediately terminates processing the command, displays the error message, and issues the prompt character (*).

User Action

Specify a nonexistent file in the CREATE command.

DUPLICATE SYMBOL IN INPUT
SYMBOL: *symbol name*
MODULE: *module name*
FILE: *pathname*

Meaning

The specified public symbol conflicts with a public symbol defined in one of the files given earlier in the input list. This error occurs only during the ADD command.

Effect

LIB86 immediately terminates processing the command, displays the error message, and issues the prompt character (*). The library being manipulated returns to the state it was in prior to the ADD command that prompted this message.

User Action

Correct the ADD command and reinvoke LIB86.

NOT A LIBRARY
FILE: *pathname*

Cause

The file that the command requests LIB86 to DELETE or LIST is not a library file.

Effect

LIB86 immediately terminates processing the command, displays the error message, and issues the prompt character (*).

User Action

Reissue the command specifying a library file.

ILLEGAL RECORD FORMAT
MODULE: *module name*
FILE: *pathname*

Cause

This error is usually caused by a transcription error or translation error in some part of an object file examined by LIB86.

Effect

LIB86 immediately terminates processing the command, displays the error message, and issues the prompt character (*).

User Action

Return to the last step in program development, then retranslate, relink, or relocate.

PREMATURE EOF
MODULE: *module name*
FILE: *pathname*

Meaning

Due to some transcription error or other the specified file has no module end record.

Cause

This is usually the result of a transcription error or translator error.

Effect

LIB86 immediately terminates processing the command, displays the error message, and issues the prompt character (*).

User Action

Return to the last step in program development, then restranslate, relink, or relocate.

```
CHECKSUM ERROR
MODULE: module name
FILE: pathname
```

Meaning

The specified file has an error in one of its checksum fields.

Cause

This is the result of a transcription error.

Effect

LIB86 immediately terminates processing the command, displays the error message, and issues the prompt character (*).

User Action

Return to the last step in program development, then retranslate, relink, or relocate.

```
ATTEMPT TO ADD DUPLICATE MODULE
MODULE: module name
```

Meaning

A module with the specified module name already exists in the library.

Effect

LIB86 immediately terminates processing the command, displays the error message, and issues the prompt character (*).

User Action

Remove the duplicate module from the list and reissue the command.

```
ATTEMPT TO ADD MODULE CONTAINING OVERLAYS
MODULE: module name
FILE: pathname
```

Effect

LIB86 immediately terminates processing the command, displays the error message, and issues the prompt character (*). All modules in the input list up to the erroneous file are added to library.

User Action

Reissue the command with all elements in the input list except those that contain overlays.

PUBLIC SYMBOL ALREADY IN LIBRARY
SYMBOL: *symbol name*
MODULE: *input module name*
FILE: *input pathname*

Meaning

The library already contains the public symbol identified in the error message.

Cause

This error occurs when a module is added that has a symbol definition already in the library.

Effect

LIB86 immediately terminates processing the command, displays the error message, and issues the prompt character (*).

User Action

Reexecute the command without the file that contains the duplicate symbol.

Table G-1 lists all of LOC86's control syntax, abbreviations, and default settings.

Table G-1. Summary of LOC86 Controls

Control	Abbrev.	Default
ADDRESSES({SEGMENTS({*segment*|*class*|*overlay*||(*addr*)}|,...|) | CLASSES({*class*(*addr*)}|,...| | GROUPS({*group*(*addr*)}|,...|) } [,...|)	AD (SM | CS | GR)	Not applicable
BOOTSTRAP	BS	Not applicable
COMMENTS	CM	COMMENTS
NOCOMMENTS	NOCM	
INITCODE[(*address*)]	IC	INITCODE(200H)
NOINITCODE	NOIC	
LINES	LI	LINES
NOLINES	NOLI	
MAP	MA	MAP
NOMAP	NOMA	
NAME(*module name*)	NA	Not applicable
OBJECTCONTROLS({LINES | NOLINES | COMMENTS | NOCOMMENTS | SYMBOLS | NOSYMBOLS | PUBLICS | NOPUBLICS | PURGE | NOPURGE} [,...])	OC	Not applicable
ORDER({SEGMENTS({*segment*[*class*[*overlay*||}[,...]) | CLASSES({*class*[(*segment*[,...])|} [,...])} [,...])	OD (SM | CS)	Not applicable
PRINT[(*pathname*)]	PR	PRINT(*objectfile*.MP2)
NOPRINT	NOPR	
PRINTCONTROLS({LINES | NOLINES | COMMENTS | NOCOMMENTS | SYMBOLS | NOSYMBOLS | PUBLICS | NOPUBLICS | PURGE | NOPURGE} [,...])	PC	Not applicable
PUBLICS	PL	PUBLICS
NOPUBLICS	NOPL	
PURGE	PU	NOPURGE
NOPURGE	NOPU	

Table G-1. Summary of LOC86 Controls (Cont'd.)

Control	Abbrev.	Default
RESERVE({*addr* TO *addr*} [,...])	RS	Not applicable
SEGSIZE({*segment*[*class*[*overlay*]] (*size*)}[,...])	SS	Not applicable
START({*symbol* | *paragraph*,*offset*})	ST	Not applicable
SYMBOLS	SB	SYMBOLS
NOSYMBOLS	NOSB	
SYMBOLCOLUMNS({1|2|3|4})	SC	SYMBOLCOLUMNS (2)

The following are descriptions of all LOC86 error and warning messages. The description of each message has up to four parts:

- Meaning—how to interpret the message
- Cause—the usual reason for the error or warning condition
- Effect—the state of LOC86 and the object file(s) after the message is issued
- User Action—what you can do to correct the condition

Not all these parts are given for each message. However, parts excluded are self-explanatory.

Error messages are always fatal, but warning messages are not. In the event of a warning, read the EFFECT of the warning carefully to determine whether the code is valid.

Error and warning messages are displayed at the console device, but printed only if a listing would otherwise be printed.

ERROR 1: I/O ERROR:
operating system error message

Meaning

An I/O error was detected. Refer to the documentation for your operating system for interpretation.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Correct the error and restart LOC86.

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

Meaning

A syntax error was detected in the invocation line. LOC86 repeats the invocation line up to the point of the error.

Cause

This is usually the result of a typographical error in the invocation line.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Reenter the invocation line more carefully.

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

Meaning

LOC86 was unable to find the input file name in the invocation. LOC86 repeats the invocation line up to the point of the error.

Cause

This is usually the result of a typographical error in the invocation line.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Reinvoke LOC86 more carefully.

```
ERROR 4:  INSUFFICIENT MEMORY
```

Meaning

The memory available on your system has been used up by LOC86.

Cause

This can be caused by an input module that has a very large number of segments or an impossibly long invocation line.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

This may require changing the source file to reduce the number of segments and retranslating.

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

Meaning

There is a record in the specified input module that has an incorrect format.

Cause

This is usually a transcription error.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Retranslate and relink the input files before attempting to locate the input module again.

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

Meaning

One of the controls or subcontrols in the invocation line is incorrect. LOC86 repeats the invocation line up to the point of the error.

Cause

This is usually the result of a typographical error in the invocation line.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Reinvoke LOC86 more correctly.

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

Meaning

You have specified an address greater than 1,048,575 (0FFFFFH). LOC86 repeats the invocation line up to the point of the error.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Examine the invocation line and invoke LOC86 with the correct address.

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

Meaning

This is a type of syntax error. LOC86 repeats the invocation line up to the point of the error.

Cause

This is usually caused by a typographical error in the invocation line.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Enter the invocation more carefully.

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

Meaning

You have specified an offset greater than 65,536 (0FFFFH). LOC86 repeats the invocation line up to the point of the error.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Retype the invocation line more carefully.

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

Meaning

LOC86 was expecting a segment name when it found a token that does not correspond to a valid segment name. LOC86 repeats the invocation line up to the point of the error.

Cause

This is usually the result of a typographical error.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Reinvoke LOC86 more carefully.

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

Meaning

LOC86 was expecting a class name when it found a token that does not correspond to a valid class name. LOC86 repeats the invocation line up to the point of the error.

Cause

This is usually the result of a typographical error.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Reinvoke LOC86 more carefully.

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

Meaning

The input module is invalid. It could mean that object module records are out of order, or LOC86 has found an invalid field within a record, or a required record is missing.

Cause

This is usually caused by a translator error or an attempt to locate something other than an object file (e.g., a source file).

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Retranslate source and relink, then try to locate again. If this error continues contact Intel.

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

Meaning

After the first memory segment is found, LOC86 issues this warning each time it finds a segment with the memory attribute.

Effect

LOC86 ignores the memory attribute on the segment specified in the message. Processing continues with LOC86 treating the additional memory segment as just another segment.

User Action

Depending on your intentions, this message may be ignored or you may wish to change the attribute for the segments and relink them.

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

Meaning

The specified group is defined by an external reference. This is a type of unresolved external reference.

Effect

LOC86 continues processing without side effects.

User Action

Find the module that defines the specified symbol and relink the input module.

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

Meaning

The specified symbol is more than 64K from its base. This error occurs when the segment containing the public symbol is not completely contained within the 64K physical segment defined by the symbol's base.

Effect

LOC86 continues processing. The object file will be executable. However, any attempt to access the specified public symbol will not produce the desired results. Debug symbols with this attribute will not be added to the object file.

User Action

Change the ORDER or ADDRESSES control so that the segment containing the public symbol will be within range of the symbol's base.

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

Meaning

The specified symbol is more than 64K from its base. This error occurs when the segment containing the local symbol is not completely contained within the 64K physical segment defined by the symbol's base.

Effect

LOC86 continues processing. The object file will be executable. However, any attempt to access the specified symbol will not produce the desired results. Debug symbols with this attribute will not be added to the object file.

User Action

Change the ORDER or ADDRESSES control so that the segment containing the local symbol will be within range of the symbol's base.

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

Meaning

The specified line is more than 64K from its base. This error occurs when the segment containing the line number is not completely contained within the 64K physical segment defined by the line's base.

Effect

LOC86 continues processing. The object file will be executable. However, any attempt to access the specified line number will not produce the desired results. Debug symbols with this attribute will not be added to the object file.

User Action

Change the ORDER or ADDRESSES control so that the segment containing the line number will be within range of the line's base.

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

Meaning

Some of the segments of the specified group are not contained within the physical segment defined by the group's base.

Cause

This error is usually caused by misuse of the ORDER or ADDRESSES control.

Effect

LOC86 continues processing the input module. The output module will be executable, but addressing errors may result.

User Action

Examine the invocation line and reinvoke LOC86 using the ORDER or ADDRESSES control more carefully.

```
WARNING 19:  BOOTSTRAP SPECIFIED FOR MODULE WITHOUT START
             ADDRESS
```

Meaning

You have specified BOOTSTRAP when locating a module that has no start address.

Effect

LOC86 continues processing as if no BOOTSTRAP control was specified.

User Action

If you wish initialization code in the program, relocate the input module specifying both BOOTSTRAP and START.

```
ERROR 20:  INVALID NAME
  NAME: bad name
```

Cause

This is the result of a typographical error in the NAME control. A valid name is composed of up to forty of the following characters in any order:

- Alphabetic (A, B, C, ..., Z)
- Numeric (0, 1, 2, ..., 9)
- Special (@, ?, :, ., _)

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Reinvoke LOC86 more carefully.

```
ERROR 21:  SEGMENT REGISTER DEFINED BY SPECIFIED EXTERNAL
           NAME
  NAME: external name
```

Meaning

A segment register or register pair is defined using the specified external symbol name.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Relink and relocate your object modules.

```
ERROR 22:  SEGMENT SIZE OVERFLOW; OLD SIZE + CHANGE > 64K
  SEGMENT: segment name
  CLASS: class name
```

Meaning

The size change specified in the SEGSIZE control caused the segment to become greater than 64K.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Look at the segment's size in the link map and reinvoke LOC86 with the correct SEGSIZE control.

```
ERROR 23:  SEGMENT SIZE UNDERFLOW; OLD SIZE - CHANGE < 0
  SEGMENT: segment name
  CLASS: class name
```

Meaning

The size change specified in the SEGSIZE control caused the segment's size to be less than zero.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Look at the segment's size in the link map and reinvoke LOC86 with the correct SEGSIZE control.

```
ERROR 24:  INVALID ADDRESS RANGE
```

Meaning

The arguments to the RESERVE control are invalid.

Cause

The usual cause of this error is that the low address is larger than the high address.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Examine the invocation line and reinvoke LOC86 correctly.

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

Meaning

The symbol specified in the START control was not found.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Either specify the argument to START with paragraph and offset, or specify an existing public symbol.

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

Meaning

The size change specified in SEGSIZE has caused LOC86 to decrease the size of the specified segment.

Effect

Decreasing the size of a segment can cause sections of code to be unaccounted for during the locating process. This is only a warning message. LOC86 continues processing with no side effects.

User Action

If the size decrease was not intended, examine the SEGSIZE control in the invocation line and relocate.

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

Meaning

You attempted to assign an address to an absolute segment.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Reinvoke LOC86 without using absolute segments in the ADDRESSES control.

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

Cause

If you have changed the specified segment's size, it may be too large to fit within a 256 byte page, or if you have specifed an address for the segment, it may force the segment to cross a page boundary.

Effect

Since this error can only occur when you have intentionally specified the segment in a control, LOC86 ignores the page resident attribute and continues to process the module as if no error has occurred.

User Action

If you have invoked LOC86 correctly, then the message is only verifying your intentions — no action is necessary.

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

Meaning

While computing an offset from a base (FRAME OF REFERENCE), LOC86 found that the REFERENCED LOCATION was more than 64K bytes away from the base.

Cause

This error usually occurs as a result of misuse of the ORDER or ADDRESSES control. One of the segments of a group is outside the 64K byte physical segment defined by its group base.

Effect

LOC86 continues processing. The print file will be valid, but the output file with regard to the out-of-place segment will not be usable.

User Action

Find the symbol that corresponds to the referenced location, and change the ORDER or ADDRESSES control.

```
WARNING 30:  UNRESOLVED EXTERNAL REFERENCE TO NAME
             AT SPECIFIED ADDRESS
  NAME: symbol name
  SEGMENT: segment name
  ADDRESS: 20-bit address
```

Meaning

There is no public definition for the specified public symbol. There is an unresolved external reference to that symbol in the specified segment.

Cause

You are locating a module that is not completely linked.

Effect

LOC86 continues processing with no side effects. The print file will be valid, and except for the unresolved references the object file should be executable.

User Action

No action is necessary if the unresolved reference is known. Otherwise, you must relink and resolve the external reference.

```
WARNING 31:  UNRESOLVED EXTERNAL REFERENCE TO NAME NEAR
             SPECIFIED ADDRESS
  NAME: symbol name
  SEGMENT: segment name
  ADDRESS: 20-bit address
```

Meaning

There is no public definition for the specified public symbol. There is an unresolved external reference to that symbol in the specified segment.

Cause

You are locating a module that has not been completely linked.

Effect

LOC86 continues processing with no side effects. The print file will be valid, and except for the unresolved references the object file should be executable.

User Action

No action is necessary if the unresolved reference is known. Otherwise, you must relink to resolve the external reference.

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

Meaning

An 8-bit displacement value, when calculated, exceeded 255.

Cause

This type of error often occurs when a page resident segment crosses a page boundary.

Effect

LOC86 continues processing. The contents of both the print file and the object file will be valid. However, the fixup value will remain invalid.

User Action

Find the symbol that corresponds to the REFERENCED LOCATION and organize your segments so that the addressing error will not be encountered.

```
WARNING 33:  GROUP HAS NO CONSTITUENT SEGMENTS
  GROUP: group name
```

Meaning

The group has no segments and is not placed in the output object file.

Cause

Often this is the result of a typographical error in the invocation line. However, it may be a linking error that has not shown up until now.

Effect

LOC86 continues processing with no side effects.

User Action

Unless there is some particular need for the specified group, no user action is necessary.

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

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Find the module that contains the specified class and link it into the module to be located.

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

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Find the module that contains the specified segment and link it into the module to be located.

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

Meaning

The two segments overlap in the specified address range.

Cause

This can be caused by any number of things: mistake in the SEGSIZE control, misuse of ADDRESSES, or two absolute segments that overlap.

Effect

LOC86 continues processing the input module. The print file is valid, and the object file, with the exception of the overlap, should be usable.

User Action

If overlap was intended, no action is necessary. Otherwise, depending on the cause of the message, it may be necessary to relocate or even modify the source, and retranslate, relink, and relocate.

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

Meaning

While attempting to locate the specified segment, LOC86 ran out of available 8086 address space.

Cause

Although it is possible to write a program that uses a full megabyte of memory, this error usually results from an error in the arguments to the RESERVE control.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Examine the RESERVE control. If, in fact, your program requires more than 1,048,576 bytes of memory, try optimizing with ASM86 or use overlays.

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

Meaning

The specified memory segment was not located at the highest address in memory.

Cause

This can only occur when you explicitly request this organization through the ORDER or ADDRESSES control, or when you implicitly request it by assigning another segment to the top of memory.

Effect

Since this can only occur by user request, LOC86 continues processing without side effects.

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

Meaning

In the ORDER control you have requested that the first segment be located below the second segment. LOC86 found that there is not enough memory to maintain this order.

Cause

This error can only occur when one of the segments in an ORDER control is absolute. The absolute segment is not necessarily either of the segments specified in the command.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Modify the order control.

```
WARNING 40:  CANNOT MAINTAIN SPECIFIED ORDERING
  SEGMENT: segment name
```

Meaning

LOC86 cannot locate all of the segments in the ORDER control consecutively.

Cause

This is usually caused by specifying absolute segments in the ORDER control or by specifying the same segments in ORDER and ADDRESSES. The conflict might not be immediately obvious. For example, the specified segment may be specified in the ORDER control by its segment name and specified in the ADDRESSES control by its class name.

Effect

LOC86 continues processing. The print and object files are valid. However, the requested segment ordering is not maintained.

User Action

Carefully examine your invocation line to find the conflict and relocate the input module.

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

Meaning

The ORDER control and ADDRESSES control for the specified class disagree.

Cause

Either you have assigned an address to the specified class or one of its constituent segments, or the translator has made one of its constituent segments absolute. In either case, the ORDER control cannot be realized.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Modify the ADDRESSES control or modify the ORDER control.

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

Meaning

The ORDER control and ADDRESSES control for the specified segment disagree.

Cause

Either you have assigned an address to the specified segment or the translator has made the segment absolute. In either case, the ORDER control cannot be realized.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Modify the ADDRESSES control or modify the ORDER control.

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

Cause

This is often caused by a typographical error or some other mechanical error while entering the invocation line.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Examine the invocation line and reinvoke LOC86 correctly.

```
ERROR 44:  SEGMENT ADDRESS PREVIOUSLY SPECIFIED IN INPUT
           MODULE OR COMMAND LINE
  SEGMENT: segment name
  CLASS: class name
```

Cause

Either the specified segment is absolute or it has been listed twice in the same ADDRESSES control.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Examine the invocation line. If the translator has made it an absolute segment, either use the translator-assigned address or retranslate the segment.

```
ERROR 45:  SEGMENT SPECIFIED MORE THAN ONCE IN ORDER
  SEGMENT: segment name
  CLASS: class name
```

Cause

This error can be caused by either of two errors in the invocation line. You have simply specified the same segment twice in the ORDER control.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Examine the invocation line and ORDER control and reinvoke LOC86.

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

Cause

You have specified the same class more than once in the same ORDER control.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Examine the invocation line and reinvoke LOC86 correctly.

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

Cause

This error is usually caused by a typographical error in the arguments to an ORDER control.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Examine the invocation line and reinvoke LOC86 correctly.

```
ERROR 48:  INVALID COMMAND LINE
```

Meaning

LOC86 has encountered an end-of-file or an I/O error while reading the invocation line.

Cause

You probably terminated the invocation line in the middle of a control argument. Most likely you forgot to type the ampersand before you typed the carriage return.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Examine the invocation line and reinvoke LOC86 correctly.

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

Meaning

The alignment attribute does not agree with the address specified in the ADDRESSES control.

Effect

LOC86 ignores the address assignment and treats the segment as any other relocatable segment.

User Action

If the address that LOC86 assigns is satisfactory, then no action is necessary. Otherwise, examine the print file and assign an address that will agree with the alignment attribute.

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

Meaning

An invocation line "token" is impossibly long. A token is a series of characters that are not broken by a parenthesis, a comma or a blank (space, carriage-return, line-feed or tab). Tokens are syntactic units used in invocation line parsing. Depending on how it is used, a token can be a control word, a symbol name, a segment name, a filename, etc.

Cause

This is often the result of a typographical error in the invocation line.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Examine the invocation line and reinvoke LOC86 correctly.

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

Meaning

The address of a self-relative reference lies outside of the 64K frame of reference of the jump or call. This error occurs while locating the module containing the self-relative instruction.

Cause

This error occurs as a result of misuse of the ORDER or ADDRESSES control.

Effect

LOC86 continues processing. The print file is valid, but the object file with respect to the module containing the self-relative reference is not executable.

User Action

Examine the locate map and reinvoke LOC86 modifying your ORDER and ADDRESSES control to correct the error.

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

Meaning

The target of a self-relative reference lies outside of the 64K frame of reference of the jump or call. This error occurs while locating the module containing the target of a self-relative instruction.

Cause

This error occurs as a result of misuse of the ORDER or ADDRESSES control.

Effect

LOC86 continues processing. The print file is valid, but the object file with respect to the module containing the self-relative reference is not executable.

User Action

Examine the locate map and reinvoke LOC86 modifying your ORDER and ADDRESSES control to correct the error.

```
WARNING 53:  CANNOT ALLOCATE CLASS AT SPECIFIED ADDRESS
  ADDRESS: 20-bit address
  CLASS: class name
```

Meaning

The specified class cannot be located at the address requested. This is the result of a conflict with another address assignment, or an absolute segment, or an address less than 200H.

Effect

LOC86 assigns the class to the nearest address that will not cause conflict. LOC86 continues processing, and both the print and object file are valid.

User Action

If the alternate address suits your purpose, then no action is necessary. Otherwise, examine the locate map and modify your invocation line.

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

Meaning

One of the data records associated with the specified segment contains an address outside of the segment's boundary.

Cause

This error can occur when you assign an address or an order to an absolute segment, or a size to a segment. Under some circumstances this can be the result of a linkage or translation error.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Change the ADDRESSES, ORDER, or SEGSIZE control and relocate.

```
WARNING 55:  UNDEFINABLE SYMBOL ADDRESS
  SEGMENT: segment name
  MODULE: module name
```

Meaning

A local symbol, line number, or public symbol has been found in the specified segment that is addressed relative to the specified group's base address. However, the segment containing the symbol is not within the 64K frame of reference that is defined for that group.

Cause

This is usually the result of an address assignment error in the invocation line.

Effect

LOC86 continues processing with no other side effects. The print file and object files are valid. However, you cannot use the symbols contained in the specified segment.

User Action

Examine the invocation line and reinvoke LOC86.

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

Cause

Either an absolute segment uses the area reserved in the invocation line or you assigned an address to a segment or class that forces the specified segment to be located in the reserved area.

Effect

The specified segment is located in the reserved area, and LOC86 continues processing with no other side effects. Both the print file and object file are usable.

User Action

If the assigned address is acceptable for the segment, no action is necessary.

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

Meaning

LOC86 was expecting a group name when it found a token that did not correspond to a valid group name. LOC86 repeats the invocation line up to the point of the error.

Cause

This is often caused by a typographical error in the invocation line.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Examine the invocation line and reinvoke LOC86 correctly.

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

Cause

This is often caused by a typographical error in the invocation line.

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Examine the invocation line to LOC86 and the link map for the input module. Reinvoke LOC86 correctly.

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

Cause

Either you gave a single group an address twice in the same ADDRESSES control or the group already had an address (due to a previous locate).

Effect

LOC86 immediately terminates processing; all open files are closed. The contents of the print and object files are undefined.

User Action

Examine the invocation line and either use the previously assigned address or assign the group one address per ADDRESSES control.

```
WARNING 60:  REFERENCED LOCATION IS NOT WITHIN 32K OF
             SPECIFIED ADDRESS
  MODULE: module name
  REFERENCED LOCATION: 20-bit address
  FRAME OF REFERENCE: 20-bit address
```

Meaning

An 8089 self-relative reference is not within 32K bytes of its target address.

Cause

Either with the ORDER or ADDRESSES control you have separated the reference from its target or the 8089 segment is too large.

Effect

LOC86 leaves the invalid reference and continues processing with no other side effects. Both the print file and the object file will be valid.

User Action

Examine the invocation line and reinvoke LOC86 correctly.

```
ERROR 61:  OVERLAY ERROR
```

Meaning

An internal LOC86 error has occurred.

User Action

Contact Intel immediately.

WARNING 62: CS AND IP REGISTERS NOT INITIALIZED

Meaning

This warning occurs when INITCODE is specified and the input register initialization record does not specify intialization of the 8086 code segment (CS) register and the 8086 instruction pointer (IP) register.

Cause

This condition is usually the result of an incomplete END directive in your assembly language module or a translation error.

Effect

The values of CS and IP at the beginning of program execution are completely dependent on the loader of your system.

User Action

Invoke LOC86 with the START control if desired.

WARNING 63: SS AND SP REGISTERS NOT INITIALIZED

Meaning

The INITCODE control was specified, but the register initialization record does not contain information for initialization of stack segment (SS) and stack pointer (SP) records.

Cause

This condition is usually the result of an incomplete END directive in your assembly language module or a translation error.

Effect

The values of SS and SP at the beginning of program execution are entirely dependent on the loader of your system.

User Action

If you will need to use the stack, retranslate your code, then relink and relocate.

```
WARNING 64:  DS REGISTER NOT INITIALIZED
```

Meaning

INITCODE was specified, but the data segment (DS) register initialization record is incomplete.

Cause

This condition is usually the result of an incomplete END directive in your assembly language module or a translation error.

Effect

The value of the CS register at program execution is entirely dependent on the system loader.

User Action

Correct your code if necessary, then reinvoke LINK86 and LOC86.

```
WARNING 65:  SEGMENT ORDER IN ORDER-CONTROL CANNOT BE
             MAINTAINED
  SEGMENT: segment name
```

Meaning

The ADDRESSES and ORDER control specifications for a segment are in conflict and/or the segment cannot be allocated space in accordance with the ORDER control.

Effect

The conflicting segment is allocated space after all other segments in the target 8086 memory.

User Action

If desired, reinvoke LOC86, using the appropriate ADDRESSES and ORDER controls.

WARNING 66: START ADDRESS NOT SPECIFIED IN OUTPUT MODULE

Meaning

The CS (code segment) and IP (instruction pointer) registers are not initialized.

Cause

The input module does not have an explicit start address, and the START control was not specified.

Effect

The values of these registers upon initial program execution are entirely dependent on the loader.

User Action

Either reinvoke LOC86 using the START control or relink to include a main module.

OH86 Error Messages

The following are descriptions of all OH86 error and warning messages. The description of each message has up to four parts:

- Meaning—how to interpret the message
- Cause—the usual reason for the error or warning condition
- Effect—the state of OH86 and the object file(s) after the message is issued
- User Action—what you can do to correct the condition

Not all these parts are given for each message. However, parts excluded are self-explanatory.

Error messages are always fatal, but warning messages are not. In the event of a warning, read the EFFECT of the warning carefully to determine whether the resulting code is valid.

Error and warning messages are displayed at the console device.

pathname, PREMATURE END-OF-FILE ENCOUNTERED

Meaning

OH86 has scanned the entire input file without finding the record that signals the end of the module.

Cause

There is a transcription error in the specified file.

Effect

OH86 immediately terminates processing and closes all open files. The contents of the output file are undefined.

User Action

Return to the last step in the program development process that did not generate this error and relocate, relink, or even retranslate.

pathname, EXPECTED MODULE HEADER NOT FOUND

Meaning

The first record in the input file was not a module header record.

Cause

This is usually caused by specifying an input file that does not contain an 8086 object module.

Effect

OH86 immediately terminates processing and closes all open files. The contents of the output file are undefined.

User Action

Check the invocation line; if you specified the input file incorrectly, then reinvoke OH86 more carefully. Otherwise, return to the last step in the program development process and reexecute.

pathname, ILLEGAL RELOCATION RECORD ENCOUNTERED

Cause

This error occurs whenever you specify a non-absolute 8086 object module as the input file.

Effect

OH86 immediately terminates processing and closes all open files. The contents of the output file are undefined.

User Action

Locate the object module with LOC86 before reinvoking OH86.

pathname, INSUFFICIENT MEMORY TO PROCESS DATA RECORD

Meaning

There is insufficient memory in your system for OH86 to process your input file.

Cause

You are trying to convert a file that is too complex for the available memory in your system.

Effect

OH86 immediately terminates processing and closes all open files. The contents of the output file are undefined.

User Action

Expand the memory on your system.

pathname, ILLEGAL REGISTER INITIALIZATION RECORD ENCOUNTERED

Cause

Your input module contains a register initialization record.

Effect

OH86 immediately terminates processing and closes all open files. The contents of the output file are undefined.

User Action

Relocate with INITCODE in effect.

pathname, ILLEGAL OVERLAY INFORMATION ENCOUNTERED

Cause

You attempted to convert a file containing overlay information.

Effect

OH86 immediately terminates processing and closes all open files. The contents of the output file are undefined.

User Action

If overlays are necessary, create root and overlay in separate files.

Index

intel®

REQUEST FOR READER'S COMMENTS

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

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

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

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

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

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

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

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

(COUNTRY)

Please check here if you require a written reply. ☐

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 HILLSBORO, OR

POSTAGE WILL BE PAID BY ADDRESSEE

DEVELOPMENT TOOLS OPERATION HF2-38
INTEL CORPORATION
5200 NE ELAM YOUNG PARKWAY
HILLSBORO OR 97124-9978

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

Send your problem report and any additional material to the address printed above.

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

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

Thanks for your comments.

INTEL CORPORATION
3065 Bowers Avenue
Santa Clara, California 95051

BELGIUM
Intel Corporation SA
Rue des Cottages 65
B-1180 Brussels

DENMARK
Intel Denmark A/S
Glentevej 61-3rd Floor
dk-2400 Copenhagen

ENGLAND
Intel Corporation (U.K.) LTD.
Piper's Way
Swindon, Wiltshire SN3 1RJ

FINLAND
Intel Finland OY
Ruosilante 2
00390 Helsinki

FRANCE
Intel Paris
1 Rue Edison-BP 303
78054 St.-Quentin-en-Yvelines Cedex

ISRAEL
Intel Semiconductors LTD.
Atidim Industrial Park
Neve Sharet
P.O. Box 43202
Tel-Aviv 61430

ITALY
Intel Corporation S.P.A.
Milandfiori, Palazzo E/4
20090 Assago (Milano)

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

NETHERLANDS
Intel Semiconductor (Nederland B.V.)
Alexanderpoort Building
Marten Meesweg 93
3068 Rotterdam

NORWAY
Intel Norway A/S
P.O. Box 92
Hvamveien 4
N-2013, Skjetten

SPAIN
Intel Iberia
Calle Zurbaran 28-IZQDA
28010 Madrid

SWEDEN
Intel Sweden A.B.
Dalvaegen 24
S-171 36 Solna

SWITZERLAND
Intel Semiconductor A.G.
Talackerstrasse 17
8125 Glattbrugg
CH-8065 Zurich

WEST GERMANY
Intel Semiconductor GmbH
Seidlestrasse 27
D-8000 Muenchen 2

Printed in U.S.A.

DEVELOPMENT SYSTEMS