MCS-80/85 UTILITIES USER'S GUIDE

for 8080/8085-Based Development Systems

Manual Order Number: 121617-001 Rev. A

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

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

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

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

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

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

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

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

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

A434/981/5K DD

PREFACE

This manual describes the ISIS-II LINK, LOCATE, and LIB utility programs, as well as the OBJHEX and HEXOBJ code conversion programs. These programs manipulate 8080 and 8085 object modules. The following references will probably aid you in your use of this manual:

Related Literature

Intellec Series III Microcomputer Development System Product Overview	121575-001
Intellec Series III Microcomputer Development System Console Operating Instructions	121609-001
Intellec Series III Microcomputer Development System Programmer's Reference Manual	121618-001
Intellec Series III Microcomputer Development System Pocket Reference	121610-001
8080/8085 Assembly Language Programming Manual	9800301D
ISIS-II 8080/8085 Macro Assembler Operator's Manual	9800292D
8080/8085 Assembly Language Reference Card	9800438D

You may also wish to keep the language manual and compiler operating instructions handy if you are programming in either PL/M-80 or Fortran-80.

Notational Conventions

UPPERCASE	Characters shown in uppercase must be entered in the order shown. You may enter the characters in uppercase or lowercase.
italics	Italics indicate variable information, such as *filename* or *address*.
[ ]	Brackets indicate optional arguments or parameters.
{ }	One and only one of the enclosed entries must be selected unless the field is also surrounded by braces, in which case it is optional.
{ }...	At least one of the enclosed items must be selected unless the field is also surrounded by braces, in which case it is optional. The items may be used in any order unless otherwise noted.
...	Ellipses indicate that the preceding argument or parameter may be repeated.

punctuation

Punctuation other than ellipses, braces and brackets must be entered as shown. For example, the punctuation shown in the following command must be entered:

```
SUBMIT PLM86(PROGA,SRC,'9 SEPT 81')
```

input lines

In interactive examples, input lines and user responses are printed in white on black to differentiate input lines from system output.

CONTENTS

CONTENTS (Cont'd.)

TABLES

ILLUSTRATIONS

CHAPTER 1
INTRODUCTION TO MODULAR PROGRAMMING

The "Tools" of Modular Programming

Modular programming is fairly straightforward once you have determined module inputs and outputs, such as parameters passed between modules, and links to the "outside world" of other programs and I/O. Modular programming produces clear, efficient programs. However useful modular programming may be, it would not help much without the software tools needed to manipulate modules. Three module handling programs can fill this need:

- A linking program which combines separate modules into a single module.
- A locating program which turns relative memory addresses into absolute addresses, so that the program may be loaded for execution or debugging.
- A library manager which permits you to create and update libraries of program modules which you will need in the future.

ISIS-II fulfills these requirements with the LINK, LOCATE, and LIB programs, along with the standard object code format produced by the PLM80 and FORT80 compilers and the ASM80 assembler. These programs provide full modular programming capacity.

NOTE

These 8080/8085 linkage and location features are not applicable to iAPX 86 and iAPX 88 object code. For information on iAPX 86,88 family object module management, see the *iAPX86,88 Family Utilities User's Guide for 8086-Based Development Systems*.

Other absolute object code formats are not compatable with the ISIS-II system so two code conversion programs have been provided:

- The HEXOBJ command translates hexadecimal code to the absolute modules format for execution on the ISIS-II system. Hexadecimal format code is produced by translators on large systems and by earlier ISIS assemblers.

- Translation from ISIS-II absolute module format to hexadecimal is performed by the OBJHEX program. This command is used to convert files from the ISIS-II format to Hexadecimal for PROM loading or execution on another system.

The LINK program combines ISIS-II 8080/8085 object modules in a single file. The relative addresses of instructions within the modules are adjusted to correspond to their position in the linked module, and references between modules and between programs are resolved.

The relative addresses assigned by LINK to instructions in program modules are converted to absolute addresses by the LOCATE program. LOCATE produces absolute object modules which can be executed under ISIS-II.

Modules which will be used again are usually kept in libraries. LIB allows the programmer to create libraries, to add or delete modules, and to list available modules. Standard mathematical functions, I/O routines, and frequently used program segments make up most libraries.

The Advantages of Modular Programming

Most programs are too long or complex to write as a single block. Programming becomes much simpler when the code is divided into small functional units. Modular programs are usually easier to code, debug, and change than straightline programs, and generally run more efficiently, too. The main program and the subroutines can be coded separately once the module interfaces have been determined. The modular approach to programming is similar to the design of hardware which contains numerous circuits. The device or program is logically divided into "black boxes" with specific inputs and outputs. The internal structure of each unit need not be known to design others. Each routine or circuit is a discrete unit whose inputs and outputs are fully defined.

Efficient Program Development

Programs can be developed more quickly with the modular approach since the small sub-programs are easier to understand, design, and test than large programs. With the module inputs and outputs defined, the programmer can supply the needed input and verify the correctness of the module by examining the output. The debugged modules are separately written and translated to machine-understandable code and stored in a library file with the LIB command. When all the needed modules are completed and tested, the LINK command will combine them into one program module. This module can be assigned absolute memory addresses by LOCATE, and the entire program tested.

Use of Different Source Languages

One of the greatest advantages of modular programming on the ISIS-II system is that individual modules may be programmed in different source languages. "Number crunching" routines are best written in FORTRAN because of its mathematical capabilities. I/O routines can be coded in PL/M because of the ease with which that language handles input and output. Assembly Language is best suited for routines which handle any bit manipulation which may be needed.

The ISIS-II FORT80 and PLM80 compilers and the ASM80 assembler produce the same object code format, so a finished program can be built from these segments.

Multiple Use of Subprograms

Code written for one program is often useful in others. Modular programming allows these sections to be saved for future use. Because the code is relocatable, saved modules can be linked to any program which fulfills their input and output requirements. With straightline programming, such sections of code are buried inside the program and are unavailable.

Ease of Debugging and Modifying

It is much easier to track down bugs in modular programs. Once the faulty module is identified, fixing the problem is considerably simpler. When a program must be modified, modular programming again simplifies the job. You can link new or debugged modules to the existing program with confidence that the rest of the program will not be changed.

Microprocessor Memory Allocation

Microprocessor memory in field applications is usually tailored for a specific task, with RAM (read/write memory) for variable data and ROM (read-only memory) or PROM (programmable ROM) for your program and constant data. Memory can be installed in such a way that some memory addresses refer to no actual memory.

You may not know the addresses of RAM and ROM in your final application during early development, while you are still writing and testing your program on the Intellec system. The only memory constraints imposed by the development system are that your programs do not overwrite ISIS-II resident routines, and that they fit within available memory. Intellec system memory is a series of RAM locations from 0 to 32, 48, or 64k bytes.

Intellec Memory Organization

ISIS-II occupies all memory below 12k (3000H), except for locations 24-63. Those memory locations contain the eight interrupts. ISIS-II and the monitor use interrupts 0, 1, and 2; and interrupts 3 through 7 are available for use by your program. Memory locations 24-63 are the only ones below 12k that can be loaded with your code. Loading other positions below 12k is not possible, but care must be taken to assure that your program does not write to memory locations in this area—it is protected from program loads, but not from program output operations. Writing to memory positions below 12k will damage ISIS-II and will cause errors when system functions are requested. (Resetting the system will restore ISIS-II if your program does accidentally write to this area.)

Starting at 12k is the buffer area, used for input and output buffers of 128 bytes each. One buffer is reserved by ISIS-II for console input/output. The minimum area allows for three permanent buffers, including the console buffer. If your program requires more than these permanent buffers, more will be added and removed as necessary.

Your programs run in the area above the buffers and below the top of memory. ISIS-II nonresident routines such as the editor and the command interpreter also run in this area. Some nonresident programs of ISIS-II use almost all of the RAM between the buffers and 32k. For this reason, the first 32k of memory in an ISIS-II system must be RAM. Above 32k, the memory may be any combination of ROM and RAM. User programs which remain permanently in memory must be placed above 32k so that the ISIS-II programs do not overwrite them.

The monitor MEMCK routine returns the address of the top of contiguous RAM minus the workspace reserved for the monitor. On a 64k system, the monitor itself uses 2k of RAM, so the value returned by MEMCK will be equal to the top of memory less 2k less 320 bytes. On a 32k system, the monitor is not located in contiguous RAM, so MEMCK will return the location of the top of memory minus 320 bytes. These reserved areas are treated in the same way as the ISIS-II reserved areas—protected from program loading, but not from program output.

As your program develops, and as your application hardware becomes available, address requirements will become more specific. In the final application, code may be in PROM beginning at location 0 and variable data may be in the first block of RAM. A program which had a base address of 4000H for compatability with ISIS-II and had variable data immediately following the code will need new addresses to fit these application requirements. A development system which produced only absolute code would require program modification to produce new addresses. With ISIS-II's relocatable object code, you need only produce a new absolute module with the LOCATE program. LOCATE can assign specific addresses to place code in ROM or PROM and variable data in RAM.

Program Segments and Memory Requirements

The locate program allows the user to specify the location of segment to areas of memory because of the way in which the language translators divide modules into segments. The segments are:

- Code—This segment contains the machine instructions and constant data which is not modified during program execution. It is generally placed in ROM or PROM.
- Data—The data segment, which must be in RAM, contains variable data and storage for input and output buffers.
- Stack—The program stack, which must also be in RAM, contains the data and return address needed when a program enters and returns from a subroutine or function.
- Memory—This segment is the remaining system RAM memory which is not required for code, data, or the stack. Its size is determined by LOCATE with the monitor MEMCK routine when the program is located for execution on ISIS-II.
- Common segments—FORT80 requires RAM memory for COMMON and BLANK COMMON areas. These areas are needed by FORT80 for storage of variables which are used in more than one block of code.
- Absolute information—Relocatable modules can contain absolute information in addition to these relocatable segments. The ASEG statement in Assembly Language causes the instructions following it until the next CSEG or DSEG statement to have absolute addresses.

  LINK can accept absolute modules produced by LOCATE. The linked module produced will be relocatable except for that section. PL/M-80 variables declared with the AT attribute have absolute references when compiled.

LINK gives segments a relative start address of zero. Subsequent instructions and data are given increasing relative addresses. The addresses are "relative" to the beginning of the segment. With LOCATE, a base address (the absolute memory location which the start of a segment will occupy) can be specified for any segment when a relative object module is located. LOCATE adds this base address to each relative address, forming absolute addresses. References between relative memory addresses are adjusted appropriately. When all relative addresses have been replaced with absolute addresses, the module is an absolute object module.

Relocation and Linkage

The ISIS-II resident compilers, FORT80 and PLM80, and the assembler, ASM80, provide LINK and LOCATE with various types of information in the relocatable object modules they create. This information consists of:

- Relative addresses of instructions and data in the module.
- A list of addresses, contained in the address fields of instructions or in data, which refer to locations within the same segment. These addresses are called intra-segment references.
- A list of addresses which refer to locations in the same module, but in a different segment. These are called inter-segment references.
- A list of addresses which refer to locations in other modules. These are termed external references.
- A list of the public and/or external symbols in the module. These symbols allow LINK to satisfy external references.

You must understand public and external symbols and external references to successfully use LINK and LOCATE. Other topics will be discussed in this section to give a complete picture of the mechanics of relocation and linkage.

Relative Addressing

When a source program is compiled by PLM80 or FORT80 or assembled by ASM80, relative addresses are assigned to instructions and information in the code and data segments. The addresses are assigned by a location counter which is set to zero at the start of each segment, and is incremented by the byte length of each instruction or data element in the segment.

LINK combines several modules to form one object module by combining all segments of a kind into one segment. All code segments are joined in one code segment; all data segments are combined into one data segment, and so on. The relative addresses in the first of each type of segment are unchanged. The addresses of the segments which are appended to that segment are equal to the length of the first segment plus their original relative address.

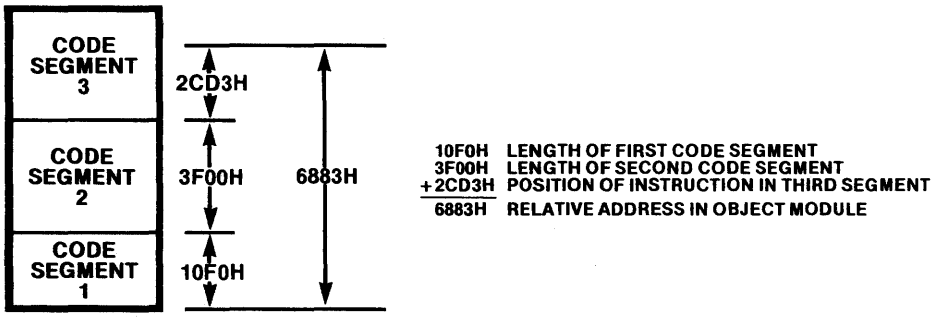

Figure 1-1. Computing Relative Addresses 121617-1

LOCATE produces an absolute module from a relocatable one by adding an absolute base address to each address in every segment. You may select the base address for each segment, but they can be left for LOCATE to assign. The order of segments in the absolute module can be assigned by the user or may be left to be set by LOCATE.

Intra- and Inter-Segment References

Relative addresses in data or in the address fields of instructions which make reference to other locations must always match the addresses of the referents. If the address refers to a location within the same segment, it is an intra-segment reference. Instructions which jump back to the beginning of a loop are typical intra-segment references. References to a location outside the current segment but within the same module are called inter-segment references. A statement in the code segment which refers to a location in the data segment is an example of an inter-segment reference.

LINK updates intra-segment references with the new addresses of the referents once they have been assigned. LOCATE converts these references to absolute addresses by adding the base address of the segment in which they appear to them.

Inter-segment references are processed similarly. The new relative addresses of the referents are substituted for the original ones, just as above, but LOCATE adds the base address of the segment which contains the referent to the address field of the reference, rather than the base of the segment containing the reference.

External References and Public Symbols

The remaining type of references are those which refer to locations not within the same module. These references differ from those above in that the translator knows nothing about the referent. In order for the translator to permit a reference to an external location, you must declare the referent as an external symbol. The translator will then know that the referent is defined in some other module. The referent is an external symbol, and the statement which uses it is an external reference.

Modules containing external references are said to be "unsatisfied" modules. LINK searches through all of the modules it is to connect for matching public symbols and external references. A public symbol is a variable or a label which is declared to be public in the source program. The public declaration is placed in the object code to be used by LINK.

When LINK matches an external reference to a public symbol, the address of the public symbol is placed in the address field of the external reference. The address may be absolute or relative. Since the modules are linked into one module, and the external reference is satisfied, the external declaration is no longer necessary. LINK removes external symbols, but not public symbols. The public symbols remain to satisfy any modules added in the future which need them. In the linked object module, the reference is now an intra- or inter-segment reference. LINK issues a warning for each unsatisfied external reference. This is not necessarily an error, so LINK continues processing until every possible external reference has been satisfied. The unsatisfied module must again be linked to satisfy the remaining external references. If you declare an external symbol but never make an external reference to that symbol, LINK will give a warning even though no unsatisfied reference exists.

When all external references have been matched to public symbols, a module is "satisfied." If LOCATE finds an external reference in an object module which it is processing, it issues a warning, but continues to produce the absolute module. This allows execution of the program for debugging up to but not including the unsatisfied reference. If the reference is executed, the result is unpredictable.

Keeping Track of Finished Files

You should have some way of keeping track of the status of object modules. One way is saving the LINK and LOCATE memory maps. Another is using extensions on the filenames which reflect the file's status. Use caution with the extension. .TMP, as many ISIS-II routines use temporary files with this extension. They will delete your file if it has the same name and extension as the temporary files they create.

Libraries

Libraries of program modules make the job of building programs even simpler. The LIB library manager program permits you to create and alter libraries of object modules. LIB creates a directory of modules in each library to keep track of the modules it contains. It will give you a list of all the modules in a library, and will list the public symbols in each module if you wish. LINK uses the libraries in the following manner. Once all the external references have been satisfied by public symbols within the modules to be joined, LINK searches the libraries which you have specified in the invocation line for public symbols to match any remaining unsatisfied external symbols. It then copies the object code of the library module containing the public symbol into the linked object module. Only the modules which are required to satisfy external references are copied. The process is repeated until all possible matches have been made.

CHAPTER 2
USING THE LINK PROGRAM

Introduction

The ISIS-II LINK program combines a number of object modules specified in an input list into a single object module in an output file. The input files may be relocatable modules produced by any of the language translators available for use with ISIS-II. They may be relocatable files which have already been linked one or more times. LINK can even accept absolute files as input. Files which have already been linked and located can be run on ISIS-II. Any file in an ISIS-II object module format can be processed by LINK.

Libraries as LINK Input Files

When libraries of object modules are among the input files, LINK searches them for symbols which resolve external references in other input modules. The only library modules which are included in the output module are those which satisfy external references in other modules. If any unsatisfied references remain after the library search is complete, a warning is placed on the LINK map. (The LINK map describes the structure of the output module and includes other diagnostic information. The map is described in detail on pages 2-4 and 2-5.)

As stated in Chapter 1, this warning is not necessarily an indication of an error, as during the intermediate steps of program development and debugging, such situations are not uncommon. If you place breakpoints in your program before the external references during these development stages, you can execute the program up to the breakpoints. If any external reference is left unsatisfied, and the program executes through it, the results cannot be predicted.

Program Segments in the LINK Process

As LINK combines the modules, it adjusts the relative addresses of the instructions and data elements in them. To do this, the modules are separated into the segments which make them up. Segments of each type (CODE, DATA, STACK, and MEMORY) are appended to one another; all data segments are joined into one data segment, all code segments are combined, and so on. The relative address of each instruction or data element is adjusted to be relative to the beginning of the new longer segment. Finally, the new segments are rejoined to form the object module.

The order of the modules in the output file is determined by the order of the input list. Thus, the file which is specified first in the list will have the lowest relative addresses in each segment. The second module listed will begin at the first available address after the first module. Again, if a library file is a member of the input list, then it must be specified after the module which refers to it.

For example, the following LINK command is given:

```
-LINK A, OBJECT.LIB,B TO C<cr>
```

Module A contains code, data, and stack segments, and a reference satisfied by a module in the library OBJECT.LIB. Module B contains code and data only. The resulting object module C has the following structure:

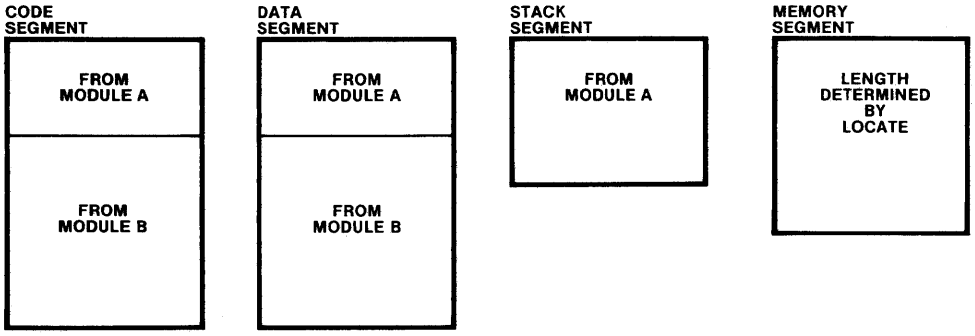

Figure 2-1. Segment Type Combination with LINK 121617-2

The exception to this rule is absolute information. LINK does not change absolute addresses. When LINK-assigned addresses conflict with absolute addresses, a message is placed on the LINK map to indicate this. This message is not necessarily an error indication. You may wish to overlap relocatable and absolute addresses.

If module B had contained a reference to OBJECT.LIB, then the invocation would have read:

```
-LINK A,B,OBJECT.LIB TO C<cr>
```

Relocation Types

All program segments have one of the following relocation types:

- Absolute, or non-relocatable (A) segments have fixed base addresses and must be placed at that address.
- Byte relocatable (B) segments are placed at the next available byte.
- Page relocatable (P) segments are located at the first available byte which lies on a page boundary (a location with an address evenly divisible by 100H, or 256). Such addresses end with -00H.
- Inpage relocatable (I) segments are located at the first available byte such that the segment does not cross a page boundary. Inpage relocatable segments must be less than or equal to 100H bytes in length.

When LINK combines segments with different relocation types, it does so using the following rules:

- Byte relocatable segments follow the preceding segment at the next byte. The output segment is byte relocatable only if all input segments are byte relocatable. Otherwise, it is page relocatable.
- Page relocatable segments follow the preceding segment at the first free page boundary. The memory locations from the end of the last segment to the page boundary are unused. This space is called a gap and will appear on the memory map. The output segment is page relocatable.

- Inpage relocatable segments are located either immediately after the preceding segment (if there is enough room between the end of the last segment and the next page boundary) or at the top of the next page. This is because inpage relocatable segments cannot cross page boundaries. LINK will change an inpage relocatable segment to page relocatability if it is longer than 100H or 256 bytes.

  The output segment will be inpage relocatable only if both of the following conditions are met. The segment must consist of all inpage relocatable segments, and it must be 256 bytes in length or less. If either of these conditions are not met, the output segment is page relocatable.

The memory that is left unused by LINK as it preserves the relocation type of the segments is marked on the LINK map. Gaps will be denoted by the entry "*GAP*" in the segment name column.

LINK Invocation

The LINK program is invoked by

```
-LINK inputlist TO outputfile [controls] <cr>
```

where *inputlist* is either

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

or

```
PUBLICS(filename[(modname,...)],...)
```

In the first input list, *filename* is an ISIS-II file containing object modules. If *filename* is a library of object modules, you have the option to specify which of its modules are to be linked into the object file. If *modnames* are specified, then only those modules will be linked in. If the modules are not specified, then LINK will search the library for modules which satisfy external references in other modules. You may specify as many *filenames* and *modnames* as you wish.

Remember—libraries must be listed after the modules for which they satisfy external references.

In the second input list, only the absolute public declarations from the specified files are included. The external references in other modules are satisfied by these declarations, but the object file produced does not contain the code of the specified files. This permits linking without combining for program overlays. (See Chapter 4.) The rules for inclusion of library modules are the same.

The *outputfile* will contain the object module produced by LINK. The *outputfile* must not be the same as any entry in the *inputlist*.

The *controls* are any of the following:

```
MAP
NAME(modulename)
PRINT(filename)
```

If the invocation is longer then one line on your terminal (or more than 122 characters), you can continue it on the next line by placing an ampersand (&) as the last non-blank character before the <cr> at the end of the line. You may continue in this way for as many lines as are necessary to complete the invocation. LINK will echo a double asterisk (**) to indicate the continuation.

For example:

```
-LINK :F1:FIRST.OBJ,:F1:SECOND.OBJ,PUBLICS&<cr>
**(THIRD.LIB(MODA,MODB)) TO &<cr>
**OUTPUT.LNK PRINT(:F1:OUTPUT.MAP)<cr>
```

The ampersand must not appear within a filename or keyword, but can be placed between a keyword and the parenthesis which surrounds its associated parameter list as above, between PUBLICS and its list.

LINK uses a temporary file named LINK.TMP on the disk which is to contain the output module. If you have a file with that same name on the disk, LINK will destroy it.

Appendix D describes error messages produced by LINK.

MAP

Syntax: MAP

Default: No LINK map is produced.

Definition: MAP requests that a LINK map be produced. The map is sent to the :CO: if no file has been specified with the PRINT(filename) control.

The LINK map includes the following information:

- LINK sign-on message
- Invocation line, exactly as entered (unless map is output to :CO:)
- The length of the relocatable segments in the output module
- The addresses of the absolute information in the output module
- The names of all of the input modules included in the object module
- Any non-fatal error messages or warnings

The example which follows includes all of this information.

```
ISIS-II OBJECT LINKER Vx.y INVOKED BY:

-LINK ZANA.DOO,KUBLA.KON,:F1:ORSON.WLZ(TINSTAAFL),&
**PUBLICS(DEVERE.AUX) TO ROSE.BUD NAME(CITIZEN@KANE)&
**MAP PRINT(TREZUR.MAP)

TINSTAAFL - MODULE NOT FOUND IN LIBRARY

LINK MAP OF MODULE CITIZEN@KANE
WRITTEN TO FILE :F0:ROSE.BUD
```

Figure 2.2 LINK Map

```
SEGMENT INFORMATION:

  START   STOP   LENGTH REL NAME

                 2345H   B  CODE
  10BCH   10EEH    33H      CODE          *GAP*
  22FEH   22FFH     2H      CODE          *GAP*
                  107H   B  DATA
                    0H   B  MEMORY
    0H      2H      3H   A  ABSOLUTE
   40H    711H   6D2H   A  ABSOLUTE
  700H    7FFH   100H   A  ABSOLUTE      *OVERLAP*

INPUT MODULES INCLUDED:
:F0:ZANA.DOO
:F0:KUBLA.KON(ALPH)
:F0:DEVERE.AUX(LIBMOD)(PUBLICS)

UNRESOLVED EXTERNAL NAMES:
   TWILIGHT-REFERENCED IN :F0:KUBLA.KON(ALPH)

<Other errors, if any, would appear here.>
```

Figure 2-2. LINK Map (Cont'd.)

The entries in the REL column indicate the relocation type of each segment. They are encoded as follows:

A Absolute or non-relocatable. Such segments must appear at the exact address which is specified in the START column.

B Byte-relocatable. Segments with this relocation type can be placed anywhere in memory which does not cause a conflict. There must be enough room to contain the entire segment without any overwriting of other segments.

P Page-relocatable. These segments must be placed at a page boundary in memory. A page boundary has an address which is divisible by 256. 8080 addresses are 16 bits long, and the addresses are contained in two bytes, called the high- and low-order bytes. The high-order byte for all addresses within a page is the same. Thus, the beginning of page-relocatable segments may be accessed by specifying only the high-order byte, and setting the low-order byte to zero. Once within the segment, other addresses in the segment are accessed by the usual combination of high- and low-order bytes.

I Segments of this type must be 256 bytes or less in length. This is because such segments contain code which only changes the low-order byte of the address. Such a segment cannot cross a page boundary, because the code within cannot make reference to an address with a different high-order address byte.

NAME

Syntax: NAME(*modulename*)

Default: *outfile* name without extension

Definition: The NAME control allows you to select a name for the output module which is more descriptive than the six-letter file name. The name can be from 1 to 31 characters long, with the following format:

letter [*letter* | *digit* | @ | ?] ...

Where letter is A through Z, digit is 0 through 9. The commercial at sign and the question mark may appear anywhere after the first character, and are generally used to separate words when there are more than one.

If NAME(*modulename*) is not specified, then the six-character name (without the three-letter extension) of the output file will be used.

Example: (See previous example.)

PRINT

Syntax: PRINT(*filename*)

Default: The LINK map, if the MAP control (Page 2-4) is specified, is output to :CO:.

Definition: PRINT(*filename*) directs the placement of the LINK map. The *filename* is any ISIS filename—but not one which already exists unless you don't care if that file's contents are destroyed and the LINK map put in their place.

Example: In the MAP control example, the LINK map is directed to the file :F0:TREZUR.MAP.

CHAPTER 3
LOCATE COMMAND

Introduction

The LOCATE program converts relocatable input files into absolute output files ready for execution on the ISIS-II system. With LOCATE, you have complete control over the absolute object module:

- You may allow LOCATE to determine the order of segments in the output file.
- You may specify the order of all of the segments.
- You may specify the order of some of the segments, and allow LOCATE to select the rest.
- You may let LOCATE determine the location of each segment in memory.
- You may specify the location of any or all of the segments.

Remember—LOCATE will determine the order and location of all of the segments which you do not specify.

If you specify memory locations incorrectly—by forcing in-page relocatable segments across page boundaries, or by causing segments to overlap—then LOCATE will issue an error message, but will continue processing the input file. The LOCATE program does this because some seeming errors—especially overlapping segments—are intentional.

Invoking the LOCATE Program

The LOCATE program is called by

LOCATE *inputfile* [TO *outputfile*] [*controls*]

where

inputfile is an ISIS-II file containing relocatable object code. *outputfile* is the name of the file to contain the absolute object code.

If TO *outputfile* is omitted, then LOCATE sends the output to a file with the same filename as the *inputfile*, but with no extension.

If you have a file with the same name as the appropriate output file, then it will be destroyed. If *outputfile* is not specified, then the *inputfile* must have an extension to prevent its destruction.

The *controls* are any of the following:

COLUMNS(*number*)
LINES
MAP
NAME(*output modname*)
PRINT(*file*)
PUBLICS
PURGE
SYMBOLS
START(*address*)
CODE(*address*)
DATA(*address*)
MEMORY(*address*)
STACK(*address*)
STACKSIZE(*length*)
/*common*/(*address*)
//(*address*)
ORDER(*segment sequence*)
RESTART0

The controls can be grouped into two categories: the format control group and the absolute control group.

The first group sets the format of the output files. The list file is formatted by all but the NAME control, which affects the object module. The second group controls the order of segments, the absolute memory locations of segments, and other parameters of the absolute object module.

How LOCATE Locates Segments

Module segments are normally located sequentially in memory in the following order:

```
CODE
STACK
/commons/
DATA
MEMORY
```

You may allow the segments to be located by this default order; you may use the ORDER command to override some or all of the defaults; or you may use the segment base address controls alone or in conjunction with either or both of these methods.

Locating with the Default Order

When you allow LOCATE to choose the order and location of the segment, the code segment is located 680H bytes above the top of ISIS-II. The rest of the segments are placed immediately above the code segment. Gaps between segments are generated only when required by segment relocation type.

- Byte relocatable (B) segments are located at the next available byte.
- Page relocatable (P) segments are located at the first available byte which lies on a page boundary (a location with an address evenly divisible by 100H, or 256). Such addresses end with 00H.
- In-page relocatable (I) segments are located at the first available byte such that the segment does not cross a page boundary. In-page relocatable segments must be less than or equal to 100H bytes in length.

Memory Pages and the H and L Registers

These relocation types are provided for programs that reference memory by manipulating the H and L registers independently. (See the 8080/8085 Assembly Language Programming Manual, order number 9800301, for a description of the HIGH and LOW operators.) You can store data on a page boundary and address elements in it by changing only the L register. Doing so saves considerable time. If the data does not cross a page boundary, you never have to change the H register at all. Be careful in using the HIGH operator on relocatable addresses. You may get an incorrect address because LOCATE assumes the unused portion of the address is zero. If the unused portion of the address is not zero, and the addition of the low order portion of the segment base address causes a carry into the high order portion, that carry will not be detected when the HIGH operator is used. In the following example, the HIGH operation is performed on the relocatable address 1234H, and LOCATE adds a segment base address of 10F0H:

HIGH(1234H) = 12H (which LOCATE assumes is 1200H)

WITH HIGH OPERATOR	WITHOUT HIGH OPERATOR
1200H	1234H
+10F0H	+10F0H
22F0H	2324H
The high part of which is 22H	The high part of which is 23H

Because LOCATE has no knowledge of the low order portion of the address, there is a chance that the high-order part of the located address will be off by one. The located address will be correct only if there is no carry from the low order portion.

You can avoid this situation by using the HIGH operator only on addresses in page relocatable segments.

This problem does not occur with the LOW operator. Addresses on which the LOW operator is used will always be correct.

Savings in memory space and execution time can result from the use of the HIGH and LOW operators, but access to some areas of memory may be lost because of the way LINK and LOCATE act to preserve relocation types.

If you let LOCATE assign the base addresses of segments, it does so in a way that preserves the relocation type of the segment. If you specify an address that violates the relocation type of a segment, LOCATE will move the segment to the next page boundary. If LOCATE encounters an in-page relocatable segment which cannot be fit into one page of memory, it changes the relocation type to page relocatable. LOCATE issues a message to the list file to report any changes.

Locating with the ORDER Control

You can completely determine the order of the segments with the ORDER control. When you list all of the segments in its parameter list, the the first segment listed will be located 680H bytes above ISIS-II. All the others will follow in order, just as with the default, but in the order specified in the parameter list. Gaps are left only when required by relocation types.

If you do not specify all of the segments in the ORDER control:

- First, the segments specified in the ORDER control are located in the order specified.
- Next, the non-specified segments are located immediately after the last specified segment. They are located according to the default order minus the already-located segments.

For example, the ORDER control

```
ORDER(DATA,STACK)
```

will cause the segments to be located as follows:

DATA
STACK
CODE
/commons/
MEMORY

Locating with the Specific Address Controls

You can also change the order of the segments by specifying the base addresses of the individual segments. As with the ORDER control, not all segments need to be specified. When the address controls are used, the following method is used to determine the location of the segments:

- Segments are located according to the ORDER control (if it is in effect) and the default.
- The starting address of a segment is either the first available byte or page boundary (according to relocation type), or the address specified in the segment address controls. Gaps will be generated when the relocation type requires it and when the next segment's specified address causes it.

Remember that LOCATE will change the specified base address of a segment if the given address violates the segment's relocation type, and in-page relocatable segments will be changed to page relocatable if the segment is longer than 256 bytes. In either case, an error message is generated.

This set of controls

```
ORDER(DATA,STACK) CODE(6000H)
```

will produce the following sequence of segments:

DATA (located 680H bytes above ISIS-II)
STACK (located according to its relocation type after DATA)
CODE (beginning at 6000H)
/commons/ (if any, immediately after CODE, according to relocation type)
MEMORY (immediately after /commons/, if any, otherwise, immediately after CODE)

There will be a gap between the end of the STACK segment and the beginning of the CODE segment.

Special LOCATE Considerations

When Using the Segment Location Controls

If you intend to specify only some of the segments, and want LOCATE to take care of the rest, it is best to use the ORDER control to modify the default sequence so that there is:

1. Enough space in memory for all of the segments
2. No chance of conflicts for the same memory space.

If you do not design the memory allocation carefully, you run the risk of conflicts. Just to be safe, always use the MAP control when specifying segment locations. Use it to verify the correctness of your memory allocation. Conflicts do not stop LOCATE, for reasons explained earlier.

When you specify FORTRAN common segment addresses using the /commons/ and // controls, you should also specify the memory segment address. The memory segment must be above the top of the highest of these segments. Since LOCATE handles common segments in an arbitrary order, you will not know which common segment will be located last. If the last common segment handled by locate is in low memory, and LOCATE locates the memory segment by default, it will be placed immediately after this common segment and it will overwrite the segments which follow.

Allocating I/O Buffer Space

Care must be taken to assure that you leave enough space for the buffers needed by ISIS-II for I/O. The maximum space required for buffers must be determined before deciding the base address of your program. The number of buffers varies dynamically, so the largest buffer area required is one large enough to handle the greatest number of buffers open at any instant. If you attempt to set the base address of your program (the lowest address in memory which your program occupies) below 3180H, an error message will be generated. The maximum number of buffers is 19.

Your program base address can be calculated with either of the following formulas:

12288 + (128 * N) (N in decimal)

or

3000H + (80H * N) (N in hexadecimal)

where N equals the greatest number of buffers required at any time during your program execution.

Use the following rules to calculate N:

- Each open disk file requires two buffers as long as it is open.
- An open line-edited file including :CI: requires one buffer until the file is closed. For a disk file, this buffer is in addition to the two already required by rule 1.
- Any system call that accesses a disk directory (LOAD, DELETE, RENAME, ATTRIB, or CONSOL when it specifies a disk file) requires two buffers during the duration of the call. The buffers are freed upon return to the calling program.
- When the CONSOL system call assigns the console input or output device to a disk file, three buffers are required for the console input file and two are required for the output file. The buffers are freed upon return to the calling program.
- When the CONSOL system call assigns the console input or output device to a disk file, three buffers are required for the console input file and two are required for the output file. The buffers are allocated until an end-of-file is encountered.

EXAMPLE: Suppose a program does not contain system calls, does not assign the console to a disk file, and is not called by a command in another disk file. Such a program needs a minimum of three buffers. If the program opens a disk file, then it will need five buffers.

To calculate this program's base address, use either formula:

12,288 + (128 * 5) = 12,928

or

3000H + (80H * 5) = 3480H

Now suppose this same program has been called from a SUBMIT file, where the console output is also a disk file. This adds two buffers for the disk output file and two for the program call from the SUBMIT file:

12,288 + (128 * 9) = 14,340

or

3000H + (80H * 9) = 3408H

If you wish your program to be independent of the type of device which is used for data transfers and independent of how it is called (from the console or from a submit file), you should allow for the maximum number of buffers it might need. This means that for any open file you should allow two buffers whether or not it is a disk file. You should also allocate five buffers for the console input and output files, whether or not it is a disk file. You should also allocate five buffers for the console input and output files, whether or not they are disk files.

Multiple-Line Invocation

LOCATE allows you to continue the invocation on more than one line if required. If the invocation is longer than one line on your terminal (up to a maximum of 122 characters), then you can continue on the next line by placing an ampersand (&) as the last non-blank character before the <cr>. LOCATE responds with a double asterisk, indicating that it is ready for the continuation. The rules for the placement of the ampersand are the same as for LINK—between words, and between controls and associated parameters, but never inside any word. For example:

```
-LOCATE SAM,PLE TO EXAM,PLE MAP PRINT(TOPO.MAP) COLUMNS&<cr>
**(3) PURGE<cr>
```

LOCATE uses a temporary file named LOCATE.TMP on the same disk as the output file. If you have a file by that name, LOCATE will overwrite it.

Appendix E, LOCATE Error Messages, lists error messages and warnings.

The control definitions follow.

COLUMNS

SYNTAX: COLUMNS(*number*)

where *number* is 1,2,or 3.

DEFAULT: COLUMNS(1)

DEFINITION: COLUMNS determines whether the symbol table in the list file is to be printed in 1,2, or 3 columns.

EXAMPLE: The control `COLUMNS(1)` will produce the following symbol table:

```
SYMBOL TABLE OF MODULE BATTLE
READ FROM FILE BATTLE.SHP
WRITTEN TO FILE BOTTLE

VALUE      TYPE      SYMBOL

           MOD       SINK
3011H      PUB       FIRE
3015H      SYM       SIO
3027H      SYM       SI1
3040H      SYM       TIGER
3062H      PUB       TANK
3102H      PUB       BIRD
3230H      SYM       DRONE
3340H      LIN       272
```

Figure 3-1. LOCATE Symbol Table

NOTE: The COLUMNS control is ignored unless SYMBOLS, PUBLICS, or LINES is specified.

LINES

SYNTAX: LINES

DEFAULT: The line numbers are not listed in the symbol table.

DEFINITION: The LINES control requests that all line numbers and module names be entered in the list file as the symbol table. The VALUE column in the table is blank for module names. The TYPE column displays LIN for line numbers, and MOD for module names.

EXAMPLE: On the symbol table above, the symbols SINK and 272 are a module name and a line number placed in the table by the LINES control.

NOTE: The LINES control is not the only control on the contents of the symbol table. The SYMBOLS control places input module names and local symbols in the table.

Although the SYMBOLS and LINES controls both place module names into the table, LOCATE only includes them once. The PUBLICS control includes public symbols on the table.

MAP

SYNTAX: MAP

DEFAULT: A memory map is not produced.

DEFINITION: The MAP control enables the listing of a memory map on the list device or file. The map lists the actual start address of the module as well as the start and stop addresses of each individual segment, the length of each segment, and the relocation type of the segment. These relocation types are from the input file, not the output file, as the segments are no longer relocatable. The relocation types are the same as for LINK. See page 2-2 for an explanation of relocation types.

When two segments overlap, a warning (*OVERLAP*) is issued. LOCATE does not stop processing when an overlap is discovered, as they are often intentional. The three-byte overlap in the following example is an intentional overlap. The three absolute bytes are intended to fit into the relocated data segment.

```
MEMORY MAP OF MODULE VOICE
READ FROM FILE :F1:VOICE.SYN
WRITTEN TO FILE :F1:VOICE.
MODULE START ADDRESS AT 3000H

START     STOP      LENGTH      REL     NAME

0008H     000AH        3H       A       ABSOLUTE
3000H     343FH      440H       B       CODE
3440H     472EH     12EFH       B       DATA
3630H     3632H        3H       A       ABSOLUTE  *OVERLAP*
472FH     475FH       31H       B       STACK
4760H     F6BFH     AF60H       B       MEMORY
```

Figure 3-2. LOCATE Memory Map

NOTE

The length given on the map for the MEMORY segment is equal to the length of the available memory on the host Intellec system. It has no relation to the actual amount of memory available in the device for which the program is destined. You must be certain that there is sufficient memory available in the target device. LOCATE has no way of knowing how much memory it has.

NAME

SYNTAX: NAME(*output module name*)

where *output module name* will contain the absolute object module.

DEFAULT: The module is given the name of the input module.

DEFINITION: This control allows you to assign a descriptive name to the object module produced by LOCATE. This name may be from 1 to 31 characters in length, and must follow the format:

letter [*letter* | *digit* | @ | ?] ...

where *letter* is A through Z, *digit* is 0 through 9, and @ and ? may appear anywhere after the first letter. These two characters are generally used to separate words in the name.

EXAMPLE:

```
NAME(VOICE@SYNTHESIS@SYSTEM)
```

PRINT

SYNTAX: PRINT(*filename*)

where *filename* is the ISIS-II file which is to receive all the matter directed to the list device.

DEFAULT: If PRINT(*filename*) is omitted, then the listing file is :CO:.

DEFINITION: The PRINT control specifies the list file for the ancillary output which is not part of the object module. The memory map, symbol table and error messages are produced for your use in debugging the program.

EXAMPLE:

```
PRINT(TREZUR.MAP)
```

PUBLICS

SYNTAX: PUBLICS

DEFAULT: The printing of the symbol table is supressed, or the public symbols are not included if it is printed by another control.

DEFINITION: The PUBLICS control causes the public symbols in the input module to be included in the symbol table. If no other symbols have been requested, then the PUBLICS control causes the generation of the table.

NOTE: This control, the SYMBOLS control and the LINES control all determine the printing or non-printing of the symbol table and its contents.

PURGE

SYNTAX: PURGE

DEFAULT: Symbols are left in the absolute module.

DEFINITION: When the PURGE control is used, all of the public symbols, local symbols, module names, and line numbers are removed from the object module. This is done to make the module smaller and to help it load more rapidly. The symbols should be left in until the module is completely debugged as the line numbers, local symbols, and module names aid this process. If the module is to be linked with another module, then the public symbols will be necessary, too. PURGE should only be used on completely debugged programs.

SYMBOLS

SYNTAX: SYMBOLS

DEFAULT: The Symbol table is not printed, or the local symbols and module names are not included.

DEFINITION: This is the third of the symbol table controls. Like the LINES control, this one causes the module names to be included in the table, but includes local symbols instead of line numbers.

START

SYNTAX: START(*address*)

where *address* is the address of the first instruction in the code segment of your program. As with all of the following controls, the value of *address* may be given in any of these bases:

Decimal: START(1000D) or START(1000) Note—a value with no base indicated is assumed to be decimal.

Hexadecimal: START(2FFFH) Note—the address must start with a digit. If the value is, for example C40F, then the control must read START(0C40FH).

Octal: START(2777O) or START(2777Q) Note—the use of the letter "O" can lead to confusion, as it looks like the numeral "0". For this reason, the letter "Q" may be used to indicate an octal value.

Binary: START(11010100101B)

DEFAULT: The start address is taken from the input module.

DEFINITION: START permits you to specify the exact starting address of your program. You can align the code segment with the ROM in your final application by specifying the address of the first ROM location. This lets you load the program into the correct memory area without having to change the object code.

The reason that device memory addresses usually do not match those from the Intellec system on which you developed the program is that Intellec addresses for user-written programs begin at location 3000H to leave room for the ISIS-II resident routines. Application systems generally do not have such restrictions.

START can also be used to create room for user-written resident routines on the Intellec system. Suppose you have such routines and that they occupy 1000H bytes of Intellec memory. These routines can reside in the 1000H bytes above the ISIS-II routines. If so, then other programs will have to begin at address 4000H or higher so as not to over-write your resident programs. Specifying the START(4000H) control will locate other programs accordingly.

EXAMPLE:

```
START(00F6H)
```

The program now begins at memory location 00F6H. Note that this address is incompatible with ISIS-II.

RESTART0

SYNTAX: RESTART0

DEFAULT: Locations 0, 1, and 2 are left unchanged in the absolute module.

DEFINITION: RESTART0 places a jump instruction in the first three locations of the absolute module. The address field (the target of the jump) is the start address of your program. The start address is taken from the START control, if that control is in effect, or it is determined by the start address in the input module. When the CPU receives a RESTART signal, the program counter is set to 0, and the jump command is executed. The jump restarts your program. Use RESTART0 when you are ready to test the program on your system, whether standalone or with the in-circuit emulator.

NOTE: RESTART0 is ignored if the module is not a main module. RESTART0 is not compatable with ISIS-II, which does not allow user code to be loaded into these locations.

STACKSIZE

SYNTAX: STACKSIZE(*value*)

where *value* is the desired length (in bytes) of the stack segment.

DEFAULT: The stacksize is obtained from the input module.

DEFINITION: STACKSIZE allows you to allocate a specific space for the stack segment of your program. The ISIS-II stacksize calculation allows for extra information which your final application will not need. By specifying the exact stacksize to fit your needs, you make your program smaller and more efficient.

EXAMPLE:
```
STACKSIZE(33)
```

NOTE: When debugging your program on an Intellec system, 12 additional bytes are required. If the STACKSIZE control is not specified, then LOCATE adds these bytes to the stack length obtained from the input file.

ORDER

SYNTAX: ORDER(*segment list*)

where *segment list* is a list of the segments which you wish to place first, second, third, etc, above the top of ISIS-II.

DEFAULT: See page 3-2, "How LOCATE Locates Segments."

DEFINITION: This control permits you to select the order of the segments in the absolute module. Those segments not mentioned in the *segment list* are located by the default as described on page 3-3.

EXAMPLE:
```
ORDER(STACK,DATA,/AHAB/)
```

CODE

SYNTAX: CODE(*address*)

where *address* is the desired base address of the code segment.

DEFAULT: See page 3-2, "How LOCATE Locates Segments."

DEFINITION: The CODE(*address*) control locates the beginning of the code segment at the specified address.

EXAMPLE:
```
CODE(3FFFH)
```

DATA

SYNTAX: DATA(*address*)

where *address* is the desired base address of the data segment.

DEFAULT: See page 3-2, "How LOCATE Locates Segments."

DEFINITION: The DATA(*address*) control locates the beginning of the data segment at the specified address.

EXAMPLE:

```
DATA(32768D)
```

MEMORY

SYNTAX: MEMORY(*address*)

where *address* is the desired base address of the memory segment.

DEFAULT: See page 3-2, "How LOCATE Locates Segments."

DEFINITION: MEMORY(*address*) control locates the beginning of the memory segment at the specified address.

EXAMPLE:

```
MEMORY(1011000110101001B)
```

STACK

SYNTAX: STACK(*address*)

where *address* is the desired base address of the stack segment.

DEFAULT: See page 3-2, "How LOCATE Locates Segments."

DEFINITION: The STACK *address* control locates the beginning of the STACK segment at the specified address.

EXAMPLE:

```
STACK(6770Q)
```

/common/

SYNTAX: */common segment name/(address)*

where *common segment name* is the name of a FORTRAN common segment, and *address* is the desired base address for that segment.

DEFAULT: See page 3-2, "How LOCATE Locates Segments."

DEFINITION: The */common/(address)* control locates the beginning of the specified FORTRAN common segment at the specified address.

EXAMPLE:

```
/NUNZIO/(3077H)
```

//

SYNTAX: *//(address)*

where *address* is the desired base address of an unnamed FORTRAN common segment.

DEFAULT: See page 3-2, "How LOCATE Locates Segments."

DEFINITION: The *//(address)* control locates the beginning of an unnamed FORTRAN common segment at the specified address.

EXAMPLE:

```
//(16384)
```

CHAPTER 4
ISIS-II LIBRARIAN

Introduction

The ISIS-II LIB command allows you to create specially formatted files to contain libraries of object modules, to alter the contents of these files by adding new modules and deleting old ones. LIB will also provide a listing of the modules in a given library, if desired. Libraries can be used as input to the LINK program, which searches libraries for modules required by the other programs being linked. The library modules which LINK selects are those which satisfy external references in the other modules.

The library manager program is called by the LIB command:

```
-LIB<cr>
```

LIB identifies itself with its sign-on message, followed by an asterisk prompt:

```
ISIS-II LIBRARIAN Vx.y
*
```

You will receive this prompt while in LIB after each command is completed. At the asterisk, you may enter any of the following LIB sub-commands:

CREATE
ADD
DELETE
LIST
EXIT

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

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

Appendix F lists the LIB error messages.

The LIB sub-command definitions follow.

CREATE

SYNTAX: CREATE *filename*

where *filename* is the name of the new library file. If another file exists with that name, an error message is given, and you are prompted for a new *filename*.

DEFINITION: The CREATE control creates an empty library file. You must then ADD modules to the file.

EXAMPLE:
```
CREATE CARNEG.LIB<cr>
```

ADD

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

where *filename* is the name of a file containing at least one object module, and *modname* is the name of a library module if that file is a library, too. If *filename* is a library file, but *modnames* are not given, all modules in *filename* are copied into *library*. You may enter as many *filenames* or *modnames* as you wish. *library* is the name of an existing library file.

DEFINITION: This command inserts modules into the *library*. The modules may be elements of another library, or they may be in object files. If a module is in an object file, then it is placed in the *library*, and the directory in the *library* is updated. If the module is contained in a library, then you may specify the modules you wish to copy, or you may omit this list and let LIB copy all of the modules in the source library. If you specify the modules, then only those modules are copied into the *library*. If you omit the *modnames*, then LIB copies the entire input library into the *library*.

EXAMPLE:
```
ADD BOOK, MONDO.LIB(BIG, GOOD) TO CARNEG.LIB<cr>
```

DELETE

SYNTAX: DELETE *library*(*modname*,...)

where *library* is the library from which you would like to remove some modules, and *modname* is the name of one of the modules you're removing.

DEFINTITION: The DELETE control permits you to remove modules from libraries for which you have no need. DELETE removes the module and alters the directory of the library to reflect this change.

EXAMPLE:
```
DELETE CARNEG.LIB(ANDREW, DALE)<cr>
```

LIST

SYNTAX: LIST*library*[(*mod*,...),...] [TO*listfile*] [PUBLICS]

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

DEFINITION: With the LIST command, you can examine the contents of your libraries. You can send this list to a file to print later, or you may print the list directly, depending upon *listfile*. If TO*listfile* is omitted, the listing will be sent to the console output. With the *library* and *modname* controls, you may specify which of the modules in *library* LIB should list. The PUBLICS control lists the public symbols in each module after the module name.

EXAMPLE: This LIST command

```
*LIST CARNEG.LIB(PROJECTOR,FILM,SLIDE)&<cr>
**TO :LP: PUBLICS<cr>
```

will cause the following to be listed on the line printer:

```
PROJECTOR
  RACK
  NUMBER
  STATUS
FILM
  HOUR
  MINUTE
  SECOND
  FRAME
SLIDE
  TRAY
  POSITION
```

Figure 4-1. LIST Command Output

EXIT

SYNTAX: EXIT

DEFINITION: The EXIT command returns control to ISIS-II.

EXAMPLE:

```
EXIT
```

CHAPTER 5
PROGRAM OVERLAYS AND LINKED LOADING

Overview

When a program is larger than the available memory, it is necessary to link the modules which make it up without combining them in one file. During execution, when one part of the program is finished, and another needed, the second can be loaded into the area of memory which had held the first. The same area of memory can hold different sections of the program at different times. This multiple use of the same memory area is called a program overlay.

Under ISIS-II, modules to be loaded separately must be in different files. The first module is loaded by giving the name of the file which contains it as a command. The subsequent loads of the overlays must be performed by your program using the LOAD system call.

In the typical use of LINK and LOCATE, modules with external references are combined with modules with matching public symbols to produce a module with no unresolved external references. In linking without combining, the external references must still be satisfied, but the program containing the public symbols must not be included. The LINK "PUBLICS" control links the public symbols from the input modules listed in the PUBLICS control to matching external references. The word PUBLICS tells LINK that the modules themselves are not to be combined into the output module, just the public symbols.

For example:

```
-LINK A, PUBLICS(B,C) TO A.LNK
```

results in a module A.LNK, whose external references to the absolute modules B and C are satisfied. B and C must be absolute modules for LINK to know the addresses of the public symbols they contain (—obviously, since relocatable modules by definition do not contain absolute addresses). The typical usage of LINK before LOCATE is reversed. You must LOCATE B and C to create absolute modules (which may contain unresolved external references--it really doesn't matter) before you LINK them to A.

Consider the following example. A "root" segment (the segment loaded first) calls segment "A." Segment "A," in turn, calls another segment, "AA." Next, segment "AAA" overlays segment "AA." Finally, the root segment calls segment "B," which overlays "A," but does not disturb "AAA," which it uses. The illustration below depicts this overlay scheme. The "cards" are program segments. The segments which overlap are overlays. Segments beside one another occupy different areas of memory.

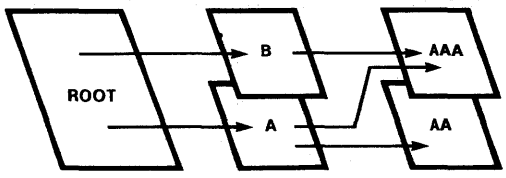

Figure 5-1. Overlays 121617-3

All of these segments must be in different files because they are loaded separately. The root loads "A" and "B," and "A" loads "AA" and "AAA."

If you locate the root first, you can use the memory map produced by LOCATE to determine the base address of "A" and "B." The maps produced by locating "A" and "B" can be used to determine the base address of "AA" and "AAA." You must locate "AA" above the top of "A," and "AAA" above the top of "A" or "B," depending upon which is higher.

Suppose the modules make the following references to one another:

The ROOT has external references to A and B.
A has external references to AA, AAA, and the ROOT module.
B contains external references to AAA and the ROOT.
AA has external references to A.
AAA contains external references to A and B.

The absolute modules produced by LOCATE are as yet unsatisfied, and have been given the .TMP extension. The following LINK commands, performed upon these absolute modules, will produce a working program.

```
-LINK ROOT.TMP, PUBLICS(A.TMP, B.TMP) TO ROOT <cr>
-LINK A.TMP, PUBLICS(ROOT.TMP, AA.TMP, AAA.TMP) TO A <cr>
-LINK B.TMP, PUBLICS(ROOT.TMP, A.TMP) <cr>
-LINK AA.TMP, PUBLICS(A.TMP) <cr>
-LINK AAA.TMP, PUBLICS(A.TMP, B.TMP) <cr>
```

The modules are now fully connected but not combined, and are ready to run. You may wish to LOCATE all of the modules again just to make certain that all of the external references have been satisfied.

NOTE

When you link without combining to produce overlays, you must provide overlay management in your program. Before a segment makes use of another, both must be in memory. In the above example, ROOT must load A and B, and A must load AA and AAA. B does not load anything, A has already loaded AAA, and the root is loaded from the start. You must also make sure that you allocate memory properly—if segment B is longer than A, and AAA begins immediately above A, then AAA will be destroyed by loading B. If AA contains data which will be needed after it is overwritten, you must provide the means to save the data when AAA is loaded. Linking without combining provides the hooks for overlaying, but the runtime management must be designed into your software.

CHAPTER 6
CODE CONVERSION PROGRAMS

Introduction

The two ISIS-II code converters exist to provide compatibility with systems employing a hexadecimal object code format by converting programs between the ISIS-II format and the hexadecimal format, and vice-versa.

The code conversion programs change the character coding, but not the content of the files they process. Instructions and addresses will be the same, but expressed in a different format.

HEXOBJ

SYNTAX: HEXOBJ *hexfile* TO *absfile* [START(*address*)]

where ***hexfile*** contains hexadecomal MCS-80/85 code, ***absfile*** is the output file to contain the ISIS-II compatible absolute object module, and ***address*** is the desired start address of the output module.

DEFINITION: HEXOBJ converts hexadecimal-encoded MCS-80/85 code into ISIS-II compatible form. The output module receives the name portion of ***absfile***. HEXOBJ produces a symbol table only if symbols were defined in the ***hexfile***.

With the optional START(***address***) control, you may specify the start address of the object module. This address may be given in any of the bases described on page —.— of chapter —. If START(***address***) is omitted, then HEXOBJ will search the end-of-file record of ***hexfile*** for that information. The address stored there is determined by an assembly-language statement, a numeric lable on the first statement of a PL/M program, or a compiler control. If none of these have been used, then there will be no start address in ***hexfile***, and HEXOBJ will set the start address of ***absfile*** to 0. You cannot load and run such a program under ISIS-II, since all memory below 3000H is reserved.

EXAMPLE:

```
HEXOBJ :F1:PRIMO.HEX TO ABS.OBJ START(3300H)
```

OBJHEX

SYNTAX: OBJHEX *absfile* TO *hexfile*

where *absfile* contained an ISIS-II absolute object module, and *hexfile* will contain hexadecimal-format object code.

DEFINITION: OBJHEX is the converse of HEXOBJ—it produces hexadecimal object code from and ISIS-II formatter file. The starting address is obtained from *absfile*, and the hexadecimal code does not contain a symbol table. You may wish to convert files to hexadecimal format for loadng into PROM, or so that the program may run on a system which uses hexadecimal coding.

EXAMPLE:

```
OBJHEX SOURCE.ABS TO :F2:SINK.HEX
```

APPENDIX A
HEXADECIMAL PAPER TAPE FORMAT

Object code is stored on paper tape in an ASCII representation of the program in memory. The code is blocked into records, each of which contains the record type, length, type, memory load address, and checksum in addition to the data. Figure A-1 shows the frames of a tape record.

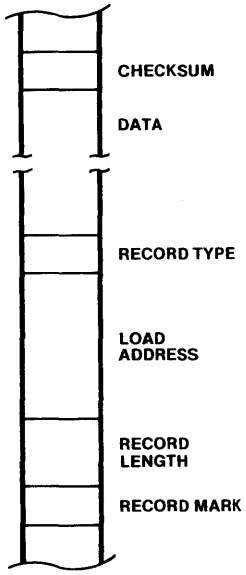

Figure A-1. Paper Tape Record Format 121617-4

The Record Mark is a colon (3AH) and is used to signal the start of a record.

The Record Length is the count of the data bytes in the record. A record length of zero indicates end-of-file.

The Load Address specifies the address at which the first data byte will be loaded. The successive data bytes will be stored in successive memory locations.

The Record Type specifies the type of this record. All data records are type 0. End-of-file records can be type 0 or 1.

The Data consists of two frames per memory word. The data is represented by hexadecimal values 00H through FFH.

The Checksum is the negative of the sum of all 8-bit bytes in the record, beginning with the Record Length and ending with the last Data byte, evaluated modulo 256. The sum of all bytes in the record (including the checksum) should be zero.

APPENDIX B
HEXADECIMAL-DECIMAL CONVERSION

The following table is for hexadecimal to decimal and decimal to hexadecimal conversion. To find the decimal equivalent of a hexadecimal number, locate the hexadecimal number in the correct position and note the decimal equivalent. Add the decimal numbers.

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

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

APPENDIX C
ASCII CODES

Table C-1. ASCII Code List

Decimal	Octal	Hexadecimal	Character
0	000	00	NUL
1	001	01	SOH
2	002	02	STX
3	003	03	ETX
4	004	04	EOT
5	005	05	ENQ
6	006	06	ACK
7	007	07	BEL
8	010	08	BS
9	011	09	HT
10	012	0A	LF
11	013	0B	VT
12	014	0C	FF
13	015	0D	CR
14	016	0E	SO
15	017	0F	SI
16	020	10	DLE
17	021	11	DC1
18	022	12	DC2
19	023	13	DC3
20	024	14	DC4
21	025	15	NAK
22	026	16	SYN
23	027	17	ETB
24	030	18	CAN
25	031	19	EM
26	032	1A	SUB
27	033	1B	ESC
28	034	1C	FS
29	035	1D	GS
30	036	1E	RS
31	037	1F	US
32	040	20	SP
33	041	21	!
34	042	22	"
35	043	23	#
36	044	24	$
37	045	25	%
38	046	26	&
39	047	27	'
40	050	28	(
41	051	29	)
42	052	2A	*
43	053	2B	+
44	054	2C	,
45	055	2D	-
46	056	2E	.
47	057	2F	/
48	060	30	0
49	061	31	1
50	062	32	2
51	063	33	3
52	064	34	4
53	065	35	5
54	066	36	6
55	067	37	7
56	070	38	8
57	071	39	9
58	072	3A	:
59	073	3B	;
60	074	3C	<
61	075	3D	=
62	076	3E	>
63	077	3F	?

Decimal	Octal	Hexadecimal	Character
64	100	40	@
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]
94	136	5E	^
95	137	5F	_
96	140	60	`
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~
127	177	7F	DEL

APPENDIX D
ISIS-II ERROR MESSAGES

Introduction

This appendix lists the error messages issued by ISIS-II. These errors can occur at the invocation of any of the ISIS-II routines, or can be generated by error conditions encountered in the operation of those routines.

The error numbers are divided into three groups as follows:

- Errors 1-99 are those which occur within ISIS-II resident routines.
- Errors 100-199 are reserved for user-written program errors.
- Errors 200-255 are generated by ISIS-II non-resident routines.

ISIS-II errors can be either fatal or non-fatal. Fatal errors halt program execution immediately, and do not permit recovery. Fatal errors result in the return of control to ISIS-II, which overwrites some of the user program area. ISIS-II displays this error message:

ERROR *nnn* USER PC *mmmm*

Where *nnn* is the error number, and mmmm is the address in the program counter when the error occurred.

Non-fatal errors halt the execution of the current program, but do not return control to ISIS-II. User-written error handling routines can restore the program to the pre-error condition. If the error occurs in a program invocation, the errant input is echoed, and an appropriate error message is given, as below:

```
-COPY :G1:CREDIT TO :F0:CRDT<cr>
:G1:CREDIT, UNRECOGNIZED DEVICE NAME
```

The non-fatal errors encountered by some ISIS-II non-resident routines have appropriate error messages—these are included in the error descriptions.

In the list below, fatal errors are identified by an asterisk before the error description.

ISIS-II Error Descriptions

0 No error detected. This is the normal state.

*1 Too few buffers have been allocated than are required to meet the current program needs. Trying to allocate more than the maximum of 19 buffers will cause this error as well.

2 ILLEGAL AFTN ARGUMENT

The value specified as an AFTN (Active File Table Number) is either not in the table of open files or is otherwise incorrect. Such an error will occur when your program tries to read a file which has been closed. To recover from this error, replace the AFTN with a correct value.

*3 Attempt to open more than six files simultaneously. The AFT can contain, at most, six files. To recover from this error, close one file, thereby freeing an AFT position.

4 INCORRECTLY SPECIFIED FILENAME

You have entered a filename which is illegal for one of these reasons:

- It contains more than six letters, or has an extension of more than three letters.
- It contains illegal characters. Filenames may contain only the characters A - Z and the numerals 0 - 9.
- The first character of the file name is a numeral. Filenames must begin with a letter.

5 UNRECOGNIZED DEVICE NAME

The device name specified is not a legal device. Check the ISIS-II User's Guide for a full list of legal devices.

6 ATTEMPT TO WRITE TO AN INPUT DEVICE

It is impossible to write to an input device such as a file open for input or any of the terminal input files. :CI:, :VI:, and :TI: are all input devices.

*7 Insufficient disk space. The disk is full. Before writing to a disk, make sure that it has sufficient room.

8 ATTEMPT TO READ FROM AN OUTPUT DEVICE

Some devices, like the line printer (:LP:), are output-only devices. You cannot read from them. :CO:, :VO:, and :TO: are output-only, and cannot be read.

9 DISK DIRECTORY FULL

There is no room in the disk directory for any new files. There is a limit of 200 files for floppy disks, and 992 for hard disks.

10 NOT ON SAME DISK

The second file specified is not on the same disk as the first, but it is expected to be. An attempt to rename a file to another disk will produce this error.

11 FILE ALREADY EXISTS

The specified filename matches one already on the disks. Many ISIS-II routines which produce this error allow you to decide whether to replace the old file with the specified one, or to change the name of the new file.

12 FILE IS ALREADY OPEN

Only the console input and output (:CI: and :CO:) may be opened more than once without being closed. Other files can only be opened once. Check your program for these possible causes:

- The program makes an unintended jump to the OPEN statement.
- The filename is misspelled in the second OPEN statement.
- You have coded more than one OPEN statement for the file.

13 NO SUCH FILE

The file specified is not in the directory of the disk specified. Probable causes are an incorrect filename or device designator.

14 WRITE PROTECTED

The intended WRITE, RENAME, or DELETE operation could not be performed because the file is write protected (attribute W). This error also occurs when one of these operations is attempted on a disk whose write-protect slot is open.

*15 Attempted load into ISIS-II reserved area. The system will not allow you to load programs below 3000H because that area is reserved for ISIS-II resident programs. Such a load operation would not permit the use of system calls, and would make return of control to ISIS-II impossible.

*16 Illegal load format. The file to be loaded is not an ISIS-II absolute-format file. To be loaded and executed by the ISIS-II system, code must be in the proper absolute format.

17 NOT A DISK FILE

An attempted reference to a disk file has been made using a non-disk device identifier. An example would be :HP:THIS in place of :F1:THIS. This error differs from number 5 in that the device name given here is a legal device name, but not the correct one.

*18 This error reports that an ISIS system call was made from a program with an illegal command number.

19 ATTEMPTED SEEK ON NON-DISK FILE

Seeks on devices other than disk drives are invalid, with the exception of :BB:. Check for possible errors in AFTNs or misspellings.

20 ATTEMPTED BACK SEEK TOO FAR

The seek went beyond the beginning of a file. MARKER is set to zero. See the ISIS- II User's Guide.

21 CAN'T RESCAN

Files which are not open for line-editing cannot be rescanned.

22 ILLEGAL ACCESS MODE TO OPEN

There are only three legal access mode parameters for OPEN:

1	Input (Read-only)
2	Output (Write-only)
3	Update (Read-Write)

Error 22 can also indicate that the access mode selected is not legal for this file.

23 MISSING FILENAME

A filename is expected as an argument to a command, and is not present. DELETE <cr> (no file designated for deletion) is a case for which this error would occur.

*24 Disk I/O error. An additional message follows the usual error number line

```
STATUS=00nn
D=x T=yyy S=zzz
```

where x is the drive number
yyy is the track address
zzz is the sector address
nn indicates the following:

For Floppy disks:

01 Deleted record
02 Data field CRC error
03 Invalid address mark
04 Seek error
08 Address error
0A ID field CRC error
0E No address mark
0F Incorrect address mark
10 Data overrun or underrun
20 Attempt to write on write protected disk
40 Drive has indicated a write error
80 Drive not ready

For Hard disks:

01 ID field miscompare
02 Data field CRC error
04 Seek error
08 Bad sector address
0A ID field CRC error
0B Protocol violation
0C Bad track address
0E No ID address mark or sector not found
10 Format error
20 Attempt to write on write protected drive
40 Drive indicates write error
80 Drive not ready

Error 24 may indicate permanent damage to the disk. If so, then you may wish to copy the salvagable files to a new disk with the COPY command.

25 Echo files, like all other I/O files, must have an AFTN which is between 0 and 255. Echo files must be open for output. Check that both of these requirements are met.

26 ILLEGAL ATTRIBUTE IDENTIFIER

The second parameter to the ATTRIB system call (not to be confused with the ATTRIB program) must be

0 Invisible file
1 System file
2 Write-protected file
3 Format attribute file

27 ILLEGAL SEEK COMMAND

The MODE for the SEEK system call must be one of these values:

0	Return marker location
1	Move marker backward
2	Move marker to specific location
3	Move marker forward
4	Move marker to end of file

Error 27 can also indicate that the mode selected is not possible for the specified file.

28 MISSING EXTENSION

A filename extension is expected but not entered. Check for mistyped filename.

*29 Premature EOF. An unexpected end-of-file has been encountered from the console or an input file. Check for possible misspellings or other errors which might cause the incorrect input file to be read.

*30 Drive not ready. The disk drive specified is not ready. Check the drive to make certain that the disk is inserted correctly, and the door is completely closed and latched.

31 CAN'T SEEK ON WRITE ONLY FILE

Seeks can only be performed on files open for update or read. This non-fatal error can be processed by selecting the correct file or by closing and re-opening the specified file in one of these modes.

32 CAN'T DELETE OPEN FILE

You must close a file before attempting to delete. Check to be certain that the filename is correct.

*33 Illegal system call parameter. Check all parameters of the call at the location specified in the PC portion of the error message.

*34 Illegal return switch in LOAD system call. The only legal values are

0	Control is returned to the calling program. The debug toggle is unchanged.
1	Control is returned to the loaded program, and the debug toggle is reset.
2	Control is returned to the Monitor. To restart program, use the Monitor "G" command.

35 SEEK PAST EOF

The file is opened for input, and a SEEK has been attempted beyond the end-of-file. You may SEEK beyond the end of a file open for update.

Error Messages for Nonresident System Routines

201 UNRECOGNIZED SWITCH

A character not known as a switch for this routine was entered. The Escape character, for example, is not allowed in invocation lines, so COPY A TO B<esc> is illegal.

202 UNRECOGNIZED DELIMITER

A character which is not allowed in a name and is not recognized as a valid delimiter has been entered.

203 INVALID SYNTAX

A parameter in a system call was inappropriate in the context used. Such a case is COPY A FOR B, where FOR is inappropriate in the COPY command.

204 Premature end-of-file. See error 29.

206 ILLEGAL DISK LABEL

The label entered in the IDISK or FORMAT command is either too long or violates the legal character rules for disk labels. These rules are the same as for files.

207 No END statement found in input. The input file being read lacks the expected END record. Check for misspelled input file name. Other causes can be a problem in translating or linking the file.

208 CHECKSUM ERROR

The bits of the records read do not add up properly. Either an I/O error has occurred or the input source is damaged. Check for damage to the disk, paper tape, or other input medium.

209 RELO FILE SEQUENCE ERROR

Either an I/O error has occurred, or an incorrect input file has been specified.

210 INSUFFICIENT MEMORY

The required or requested amount of RAM is not available. Either too much RAM was requested, or some hardware problem has arisen.

211 RECORD TOO LONG

A record longer than expected has been encountered. Make sure the input file is correct.

212 ILLEGAL RELO TYPE

Relocation types must be one of these:

A Absolute, non-relocatable
B Byte-relocatable
P Page-relocatable
I In-page relocatable

See Chapter 2 for an explanation of relocation types.

213 FIXUP BOUNDS ERROR

The address required violates numeric bounds on addresses. See Chapter 1 for an explanation of Intellec memory configuration.

214 ILLEGAL SUBMIT PARAMETER

One of the actual parameters to be substituted for a formal parameter within a submit file is in error. See the *ISIS-II User's Guide*.

215 ARGUMENT TOO LONG

The number of characters in an actual argument may not exceed 31.

216 TOO MANY PARAMETERS

More parameters supplied than defined, or the limit of 10 parameters has been exceeded.

217 OBJECT RECORD TOO SHORT

One of the records in an object module file has fewer bytes than were expected. This error can be caused by an I/O error or by an incorrect file specification.

218 ILLEGAL RECORD FORMAT

The record format must conform to Intel standard. Check for a misspelled filename or damage to the disk or input medium.

219 PHASE ERROR

The expected phase input (for the next step of a translation process) is incorrectly specified.

220 No end-of-file record in object module file. An end-of-file record must be contained in every object file according to the standard object file format.

221 Segment overflow during LINK. Because of the size of the Intellec memory area, no segment may exceed 64K bytes in length. This error indicates that some segment has exceeded this limit.

222 Unrecognized record in object module file. Here, again, the object module file does not match the standard object module format.

223 Fixup record pointer is incorrect. The fixup record allows LINK to adjust the addresses of inter- or intra-segment references, and external references. If the pointer is incorrect, then LINK cannot correctly adjust the addresses.

224 Illegal record sequence in LINK object module file. The records within one of the LINK object module files are out of order. This could indicate an I/O error, damaged disk, or an incorrect file specification.

225 Illegal module name specified. Module names must conform to the standard format explained on page —.—.

226 Module name exceeds 31 characters. There is a 31 character limit on the length of filenames.

227 Command syntax requires left parenthesis. Self explanatory.

228 Command syntax requires right parenthesis. Self explanatory.

229 Unrecognized control specified. One of the controls to a command is probably misspelled, or is not a legal control for this command. Example: LINK A.OBJ, B.LIB TO AB.OBJ START(3000H). The START control is a LOCATE control, not a LINK control.

230 Duplicate symbol found. If LINK encounters a symbol which is used in more than one module, this message is issued. If it indicates an error, then you must change the source code of one of the modules so that the symbols differ.

231 File already exists. See error 11.

232 Unrecognized command. The command specified does not exist. Check for misspellings. Check the ISIS-II User's Guide for a list of legal commands.

233 Command syntax requires a "TO" clause. Some commands, such as COPY, require a TO <filename> clause.

234 CANNOT FORMAT FROM TARGET DRIVE

The source disk for formatting another disk must be in a different drive from the target drive, except when the single drive mode is selected with the IDISK command.

235 NON-DISK DEVICE

You have specified a non-disk device, such as :LP:, when the system expects a disk device. You cannot use IDISK on the line printer, for example.

236 More than 249 common segments in input files. No more than this number can be processed. This limit should not impose restrictions under normal circumstances.

237 Specified common segment not in object file. You have specified a non-existent common segment in the "/common name/" control to LOCATE.

238 Illegal stack content record in object file. The stack content record in the object file does not conform to the expected format. This usually indicates an I/O error.

239 No module header record in input object file. Again, an error in the format to the input object module file, possibly due to an I/O error. Check also for a damaged disk.

240 Program exceeds 64K bytes. Obviously, this will not fit into the Intellec memory. You may wish to subdivide the program and use overlays so that the memory needed at any one time is less than 64K bytes. Chapter 4 explains the use of program overlays.

APPENDIX E
LINK ERROR MESSAGES

Introduction

Errors encountered by LINK cause an unnumbered message to be sent to the current console device and, in the case of non-fatal errors, to the LINK map. Fatal error messages are sent only to the console device because processing is halted. Non-fatal errors do not halt LINK processing.

Fatal Errors

Command Errors

Errors caused by improper command entry are followed by a partial copy of the errant command followed by a number sign (#) in the vicinity of the error.

```
ERROR MESSAGE
partial copy of command#
```

The following messages identify improper command errors.

```
INVALID SYNTAX
```

This message occurs when some part of the command is not recognized. The problem can be a mistyped keyword, a missing comma, or an non-blank character following a line-continuation ampersand.

```
DUPLICATE FILE NAME
```

The same file is specified as the input and output file.

```
'TO' EXPECTED
```

The command syntax requires a 'TO' clause for the output file. This message indicates that the clause has been omitted.

```
LEFT PARENTHESIS EXPECTED
```

A PUBLICS, NAME, or PRINT keyword is not followed by a "(".

```
RIGHT PARENTHESIS EXPECTED
```

The list following one of the three keywords listed above is not terminated with a ")".

```
INVALID NAME
```

The module name contains an illegal character, or begins with a numeral. See Chapter 2 for the rules concerning module names.

```
NAME TOO LONG
```

Module names may not be longer than 31 characters.

```
UNRECOGNIZED CONTROL
```

A character string other than NAME, MAP, or PRINT has been encountered.

Input File Errors

If there is an error in the internal format of a specified input file, one of the following messages will be generated. These errors can be the result of something as simple as a misspelling of a filename, or the error may be the result of problems generated by a language translator or a previous LINK. If the filenames all appear to be correct, then compile the program and try to LINK it again. If the problem persists, and the fault seems to be either LINK or the translator, report it to Intel with a Software Problem Report.

filename, PREMATURE EOF (see ISIS-II error 29)
filename, CHECKSUM ERROR (see ISIS-II error 204)
filename, RECORD TOO LONG (see ISIS-II error 211)
filename, ILLEGAL RELO RECORD (see ISIS-II error 212)
filename, FIXUP BOUNDS ERROR (see ISIS-II error 213)
filename, ILLEGAL RECORD FORMAT (see ISIS-II error 218)
filename, NO EOF
This error indicates that the file being read in has no end-of-file record. LINK cannot process such a file.
filename, BAD RECORD SEQUENCE
The records in the object module file specified are out of order. LINK cannot process a file unless the records are in the proper order.
filename, ILLEGAL STACK CONTENT RECORD (see ISIS-II error 238)
filename, NO MODULE HEADER RECORD
The file named lacks the module header record which contains information needed by LINK to process the file.
filename, NOT A LIBRARY
You have specified a list of library modules following a file which is not a library.
filename, SEGMENT TOO LARGE (see ISIS-II error 221)
filename, INSUFFICIENT MEMORY
LINK cannot create the output file specified because there is not enough room in memory for the LINK work space, which consists mainly of the symbol table.

Non-Fatal Error Messages

Non-fatal errors issued by LINK are written to the map file and to the current console device (if different).

MORE THAN 1 MAIN MODULE, CONFLICT IN modname

This indicates that LINK found more than one main module in the input list. The module named in the message is the second main module found. All of the main modules are included in the output module, but the starting address of the output module is taken from the first main module detected.

name-MULTIPLY DEFINED, DUPLICATE IN modname

The public name given here was defined in more than one module. The second definition was detected in the module specified.

MODULE NOT IN LIBRARY, LOOKING FOR filename(modname)

The module named has not been found in the library given in the error message.

/name/-UNEQUAL COMMON LENGTH, CONFLICT IN modname

Two named common segments with the same name but different lengths were found. The module containing the second segment found is given in the message.

APPENDIX F
LOCATE ERROR MESSAGES

Introduction

Errors encountered by LOCATE cause an unnumbered message to be sent to the current console device and, in the case of non-fatal errors, to the memory map. Fatal error messages are sent only to the console device because processing is halted. Non- fatal errors do not halt LOCATE processing.

Fatal Errors

Command Errors

Errors caused by improper command entry are followed by a partial copy of the errant command followed by a number sign (#) in the vicinity of the error.

ERROR MESSAGE
partial copy of command#

The following messages identify improper command errors.

INVALID SYNTAX

This message occurs when some part of the command is not recognized. The problem can be a mistyped keyword, a missing comma, or an non-blank character following a line-continuation ampersand.

'TO' EXPECTED

The command syntax requires a 'TO' clause for the output file. This message indicates that the clause has been omitted.

LEFT PARENTHESIS EXPECTED

A COLUMNS, ORDER, START, STACKSIZE, CODE, DATA, STACK, MEMORY, /common name/, //, NAME. or PRINT keyword is not followed by a "(".

RIGHT PARENTHESIS EXPECTED

The list following one of the twelve keywords listed above is not terminated with a ")".

INVALID NAME

The module name or /common name/ contains an illegal character. See Chapter 2 for the rules concerning module names.

NAME TOO LONG

Module names and /common name/s have a length limit of 31 characters.

common name, COMMON NOT FOUND

The input module does not contain the common segment specified in the command.

UNRECOGNIZED CONTROL

A character string other than NAME, MAP, PRINT, COLUMNS, SYMBOLS, LINES, PUBLICS, PURGE, ORDER, CODE, DATA, STACK, STACKSIZE, MEMORY, /common name/, //, RESTART0, START, or STACKSIZE has been encountered.

Input File Errors

If there is an error in the internal format of a specified input file, one of the following messages will be generated. These errors can be the result of something as simple as a misspelling of a filename, or the error may be the result of problems generated by a language translator or during LINK. If so, then translate the source code again, and re-LOCATE the object module. If the problem persists, and seems to be the fault of LINK or the translator, report it to Intel with a Software Problem Report.

filename, PREMATURE EOF (see ISIS-II error 29)
filename, CHECKSUM ERROR (see ISIS-II error 204)
filename, RECORD TOO LONG (see ISIS-II error 211)
filename, ILLEGAL RELO RECORD (see ISIS-II error 212)
filename, FIXUP BOUNDS ERROR (see ISIS-II error 213)
filename, ILLEGAL RECORD FORMAT (see ISIS-II error 218)
filename, NO EOF
This error indicates that the file being read in has no end-of-file record. LOCATE cannot process such a file.
filename, BAD RECORD SEQUENCE
The records in the object module file specified are out of order. LOCATE cannot process a file unless the records are in the proper order.
filename, ILLEGAL STACK CONTENT RECORD (see ISIS-II error 238)
filename, NO MODULE HEADER RECORD
The file named lacks the module header record which contains information needed by LOCATE to processs file.
filename, PROGRAM EXCEEDS 64K (see ISIS-II error 221)
filename, INSUFFICIENT MEMORY
LOCATE cannot process the input file specified because there is not enough room in memory for work space.

Non-Fatal Error Messages

Non-fatal errors issued by LOCATE are written to the map file and to the current console device (if different).

IN-PAGE SEGMENT > 256 BYTES COERCED TO PAGE BOUNDARY

An in-page relocatable segment has been discovered which is longer than the limit of 256 bytes for such segments. See Chapter 2 for a description of relocation types.

UNSATISFIED EXTERNAL(n) external name

This error occurs when an unsatisfied external name is encountered in the input file. The number (n) is the count of the number of unsatisfied names uncovered so far. It is used to identify the unsatisfied name in the following message.

REFERENCE TO UNSATISFIED EXTERNAL(n) AT xxxxH

This message reports the address of the reference to the unsatisfied external name identified by (n).

(MEMORY OVERLAP FROM xxxxH THROUGH yyyyH

This message is issued if the same memory location is defined in more than one program segment.

APPENDIX G
LIB ERROR MESSAGES

Introduction

All LIB error messages are nonfatal because LIB is an interactive program. The command which caused the error will be aborted, but LIB will not be interrupted.

Command Errors

Errors caused by improper command entry are followed by a partial copy of the incorrect command with a number sign (#) in the vicinity of the error.

ERROR MESSAGE
partial command#

The following are the LIB command error messages:

INSUFFICIENT MEMORY

LIB cannot execute the command given because it requires more memory than in available in the intellec system.

INVALID MODULE NAME

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

INVALID SYNTAX

Check the command for one of the following:

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

LEFT PARENTHESIS EXPECTED

There is a missing "(" in the command.

RIGHT PARENTHESIS EXPECTED

There is a missing ")" in the command.

MODULE NAME TOO LONG

The specified module name exceeds 31 characters.

'TO' EXPECTED

The TO *filename* is omitted in the ADD command.

UNRECOGNIZED COMMAND

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

File or Module Errors

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

FILE ALREADY EXISTS

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

filename, CHECKSUM ERROR (see ISIS-II error 208)

Filename, DUPLICATE SYMBOL IN INPUT

You have attempted to ADD a file which contains a PUBLIC symbol already within the library.

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

filename, NOT LIBRARY

The specified file is not a library.

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

filename(modname): NOT FOUND

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

modname-ATTEMPT TO ADD DUPLICATE MODULE

The specified module name already appears within the library.

symbol-ALREADY IN LIBRARY

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

INDEX

REQUEST FOR READER'S COMMENTS

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

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

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

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

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

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

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

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS ...

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

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

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

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

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

Printed in U.S.A.