ITHACA INTERSYSTEMS
LINK Z
A LINKING LOADER
REVISION 1.0

Copyright by
© Ithaca Intersystems, Inc.

LINK/Z

A Linking Loader

© Copyright 1980 by
Ithaca InterSystems, Inc.

Manual Revision 1

TABLE OF CONTENTS

This program is dedicated to and named after a good friend, Link Hogthrob.

INTRODUCTION

This manual is concerned with the actual operation of the linker. The RELOCATION section in the assembler manual discusses the use of the assembler and linker together to produce a program.

You may be reading this manual because you want to know how to link and run a program written in Pascal. If this is the case, you need read only the first few sections of this manual up to and including the WORKED EXAMPLE.

On the other hand, you may want to add your own assembler routines to a Pascal program, or you may want to write a stand-alone assembler program. In that case, you should first read the entire assembler manual and then read this manual including the sections immediately following.

The examples in this manual assume that you are using the linker with the CP/M™ operating system.

WHY LINK?

Linking together and loading a program from relocatable modules gives you the following advantages:

1) You may load the program almost anywhere in memory.

2) You may use a library of pre-assembled subroutines in your program.

3) The search option loads only the needed modules from the library which keeps the program size down.

4) Editing and assembling a small main module and linking it with a library is quicker than editing and assembling a single large equivalent program.

Let us look at the steps you go through from the creation of a Pascal program to its execution.

1) Run the text editor, create a new PAS file on a disk, and type in your Pascal program.

2) Run the Pascal compiler which translates your Pascal statements into assembler mnemonic statements.

3) Run the assembler which translates the assembler mnemonic statements into a relocatable machine language module.

4) Run the linker and load the relocatable module you generated in the previous step along with the Pascal subroutine library.

5) Save your new program (if you want to) and run it.

The compiler translates your Pascal statements into assembler statements which, after assembling and linking, are translated into machine executable code. This code is capable of loading and storing variables and of doing a few other general tasks. The compiler handles the more specialized tasks (multiplying floating point numbers, reading and writing data files, etc.) by calling subroutines contained in the library. This is why the module that you create must be linked with the library to make a complete executable program.

WORKED EXAMPLE

In this example, assume that you have already used the text editor to create a Pascal file called TESTPROG.PAS. Below is an example of what your console might look like after the creation of an executable program.

```
A>PASCAL TESTPROG
InterSystems Pascal v-3.0   Copyright 1980 by Jeff Moskow
---

A>ASMBL MAIN,TESTPROG/REL
PASCAL RUN-TIME SUPPORT LIBRARY                ASMBLE v-5b

A>LINK TESTPROG /N:TESTPROG /E
LINK version 1e
Load mode
Generate a COM file

A>
```

The first command, PASCAL TESTPROG, runs the Pascal compiler and compiles TESTPROG.PAS into TESTPROG.SRC.

The second command, ASMBL MAIN,TESTPROG/REL, runs the assembler and assembles MAIN.SRC (the start-up code) with your program, TESTPROG.SRC, into TESTPROG.REL.

The third command, LINK TESTPROG /N:TESTPROG /E, runs the linker and loads your module, TESTPROG.REL, into memory. It then automatically loads the necessary parts of the library, LIB.REL, in search mode, writes a COM file, TESTPROG.COM, and returns to the monitor. You may now run this COM file by typing:

```
TESTPROG
```

When you are debugging a program you do not have to generate a COM file in order to test the program. Instead, you may give the following linker command:

```
LINK TESTPROG /G
```

This loads your module along with the necessary parts of the library and starts it. When you have completely debugged your program you may link it again and save it in a COM file.

LINKER OPERATION

The linker may operate in one of two modes: the librarian mode (to build libraries) or the load mode (to load programs).

In the librarian mode, the linker reads in several relocatable modules and writes them all out together to a single library file.

In the load mode (the default mode), it reads one or more relocatable modules, loads them into memory, and links them together (matches external symbols with entry points).

COMMAND LINE

A command line tells the linker what to do. You can put a complete set of commands on one line or you can put the commands on many lines. For example, you could write:

```
NAVIGAT,TRIG/G
```

or you could write:

```
NAVIGAT
TRIG
/G
```

Both have the same meaning to the linker.

The command lines are made of one or more items (file names, options) which are separated from each other by spaces or commas or carriage returns. These items are grouped together to form entries which consist of a file name and/or options. An entry consists of everything in the line up to but not including the next file name. Each option starts with a slash (/). An option may be followed by a colon (:) and an output file name. A command line is read and executed from left to right, entry by entry. For example, consider the following command line:

```
/A FILE1 FILE2/B:OUTFILE/C FILE3/D
```

The first entry is just an option, /A. It is executed first. The next entry is FILE1 with no options. It names an input file which is read in. The next entry is FILE2/B:OUTFILE/C. The input file name is FILE2 which is read in. Two options, /B:OUTFILE and /C, apply to this entry. The /B option is followed by a file name, OUTFIL, which is opened for output. The last entry in the line is FILE3/D. FILE3 is read in. An option, /D, applies to this entry.

If you forget to specify an output file name or if the file cannot be successfully opened the linker will not let you proceed with an operation which requires an output file. It prints an error message and ignores the remainder of the command line. At that time you should specify the option and file name again. For example:

```
/B:OUTFILE
```

When the linker has exhausted a command line it prompts you for another one by printing an asterisk (*).

A library is a collection of subroutines which are related in some way. For example, the Pascal system uses a library which contains subroutines to multiply and divide, do floating point operations, handle files, etc. This library is called LIB.REL. You may want to generate your own libraries. For example, suppose you write a lot of text processing programs and you notice that certain subroutines appear in many of the programs (search a buffer for a string, print a string, etc.). You may put these subroutines into a library and add them to your programs at link time. This saves a lot of writing, assembling, and debugging.

To make a library you first assemble each subroutine (one at a time) into a relocatable (REL) module. Then you run the linker and type /L:filename as the very first command (filename is replaced by the actual name of the library file you wish to create). For example, you might type:

```
LINK /L:TRIG
```

It is important to type /L as the first command to tell the linker to enter the librarian mode. If you make a mistake and type something else the linker enters the load mode. You can restart it by typing /R (on a line by itself) and then /L. If a file with the same name exists on that device it is deleted before the new output file is opened. For example:

```
LINK /L:B:TRIG
```

Now you specify the modules you want to be included in the output file in the order in which they should appear. You might type:

```
COT
TAN
SIN
COS
DIV
MULT
```

or you could specify all the files names on a single line (80 characters max).

```
COT,TAN,SIN,COS,DIV,MULT
```

As each module is read its internal name (specified by the NAME command in the assembler) is printed on the console.

If you make a mistake and include a module that you really didn't want you can start over by typing /R. This restarts the linker. You now have to type /L:filename and all of the input file names again.

QUERY

The librarian has a query option, /Q. If you select this option (for a given file) the linker asks you if you want to include a particular module from an existing library in the output file. It does this by printing OK? after the module name. If you want to include the module type Y (yes), if you do not want to include any more modules from this file type Q (quit), if you want to skip just this one module type any other key. This is a one keystroke response. The linker begins including or skipping the module as soon as you hit a key, so be careful. You can't delete a wrong keypress. The linker is purposely made this way so that you can place your fingers over two keys (Y and space, for example) and quickly zip through a library.

Suppose that you rewrite the COT module and want to put it in the library to replace the existing module. The best place to put the new module is at the beginning of the library. You might type:

```
LINK /L:TRIGNEW COT TRIG/Q/E
```

LINK loads and starts the linker. /L puts it into the librarian mode. :TRIGNEW generates a new output file called TRIGNEW.REL. Notice that its name is different from the old library file, TRIG.REL. If it was given the same name the old library file would be deleted before it is read in. After the linking session is complete you can delete the old library and rename TRIGNEW.REL to TRIG.REL.

The new COT module is included in the new library. The old library, TRIG.REL, is read in query mode. You select which modules you want to include.

When you have included everything you want in the library type /E. This closes the output file and returns control to the operating system.

The console looks like this at the end of the previous example:

```
A>LINK /L:TRIGNEW COT TRIG/Q/E
LINK version 1
Librarian mode
Module name is COT
Module name is COT     OK?
Module name is TAN     OK? Y
Module name is SIN     OK? Y
Module name is COS     OK? Y
Module name is DIV     OK? Y
Module name is MULT    OK? Y
A>
```

AUXILIARY INPUT FILE

It is often necessary to replace one or two modules in the middle of a library file. You can easily do this by opening an auxiliary file in the middle of a normal library file and adding the modules from the auxiliary file. You then continue processing the remainder of the normal file.

If you are reading a library file in query mode you may respond to the OK? with A ("auxiliary"). The linker saves the current normal module, prompts you with an asterisk (*), and waits for you to type an input file name. When you give it a file name it reads that file and prints the name of the first module in the file followed by OK?. You may respond by typing Y, Q, or any other key. When the linker has finished processing the auxiliary file it returns to the normal file, prints the name of the module again (the name to which you responded with A), and continues. Note: the remainder of the command line is ignored when an auxiliary file name is given.

Suppose you have rewritten the SIN module. You might replace it in the library as follows. First you type:

```
LINK /L:TRIGNEW TRIG/Q
```

This is the same command line that you used before. You tell the linker to include the COT and TAN modules in the library. When it comes to the SIN module you tell it to open an auxiliary file and include the new SIN module. Then you tell it to skip the old SIN module and include the remainder of the old library. The console looks like this at the end of the operation:

```
A>LINK /L:TRIGNEW TRIG/Q
LINK version 1
Librarian mode
Module name is COT    OK? Y
Module name is TAN    OK? Y
Module name is SIN    OK? A
*SIN
Aux Module name is SIN    OK? Y
Module name is SIN    OK?
Module name is COS    OK? Y
Module name is DIV    OK? Y
Module name is MULT   OK? Y
*/E
A>
```

LOAD

If the first command you give the linker is anything but /L it enters the load mode. If you make a mistake (load a module you really didn't want to, for example) you may restart the linker by typing /R. The linker forgets everything it has loaded and starts from scratch.

LOADING SPACE

The linker takes up some space in memory (five or six kilobytes, depending upon the version). It generates two symbol tables; one for entry points and one for unresolved external symbols. Each symbol takes up eight bytes. Suppose that your system has twenty kilobytes of user space and that you want to load a program with 125 symbols. The linker (6K) and the symbol table (1K) take up about seven kilobytes of memory. You have thirteen kilobytes left for your program. You may load anywhere in that space from location zero on up.

LOADING BUFFER

The linker does not load code directly into memory but instead puts it in a loading buffer. After all the code for a program has been loaded the linker moves the code down to its actual execution location (where you want it). This means that you may load code anywhere you want, even at location zero. It will not interfere with the operating system because the code that you load does not actually appear in low memory until after the linker has completed its task. The code to execute the move consists of a block move instruction (LDIR, two bytes) and a jump to the beginning of the program or a return to the operating system (three bytes). This code is placed at location 80H (the command line buffer). In a CP/M environment, COM files (the usual type of file containing an executable program) must start at 100H; the linker automatically puts eight bytes of code there to initialize the stack pointer and jump to the beginning of the program. (The stack pointer is initialized to the top of the TPA - the transient program area.) You are free, then, to store code from 0 to 7FH, from 85H to FFH, and from 108H to the top of the TPA - assuming, of course, that the quantity of code you load leaves room for the operation of the linker.

LOADING ORDER

If no forced loading is specified all relocatable code is loaded starting at location 108H as follows:

1) All common sections from the first module are loaded first.
2) The program section from the first module is loaded next.
3) The data section from the first module is loaded.

4) Common sections from the second module that have not yet been defined (whose names are different from those in the previous modules) are loaded.
5) The program section from the second module is loaded.
6) The data section from the second module is loaded.

7) Etc. for the remaining modules.

You may force all of the sections (/A), the common sections (/C), the data sections (/D), or the program sections (/P) to be loaded starting at a particular location. These options must always be followed by a colon and a hex address. For example:

```
/A:1234
```

forces the modules to be loaded, in the order given above, starting at location 1234H.

The /C, /D, and /P options break up the loading order. For example:

```
/C:4000
```

forces all common sections to be loaded starting at location 4000H. Since the data and program section locations have not been forced they are loaded in their normal order starting at the default location (108H); PROG1, DATA1, PROG2, DATA2, etc.

A program which is to be burned into a PROM may be loaded as follows: Suppose the PROM is to be at location 1000H and the data space is to be in RAM at location zero.

```
/P:1000 /D:0
```

Any or all of these options may be used together (if you are careful). When the linker loads code it checks to make sure that the code fits in the available space (that is, it doesn't overwrite the linker, tables, or operating system). However, it

does not keep track of what has been written in the available space (where your program is). You must carefully examine the load map to make sure that one section has not overwritten another section. If you discover that something has been overwritten you can start over (with /R) and allow more room to prevent the overwrite.

OFFSET LOAD

Sometimes it is necessary to load code outside of the normal linker loading space (for instance, when patching or building an operating system). The offset load mode lets you do this. You enter the offset load mode by typing /O:XXXX (letter oh) as the first command to the linker (or as the first command following a /R). XXXX is the lowest address into which you want to load code. For example, if you want to load a routine at E000 you would type:

```
/O:E000
```

Several things happen differently in the offset load mode:

1) The start-up code at location 100H (load stack pointer, jump to beginning of program) is not generated.

2) The slide-down code (LDIR, jump to start) is not generated and the code in the buffer is not moved down (or up) to its execution location.

3) You cannot generate a true COM file, only a HEX file. (If you use the /N option, the linker will display the message "Generate a non-standard object code file", and will generate an output file designed specifically for use with the Pascal/Z overlaying compiler. The format of this file is as follows: the first two bytes give the address at which the code is supposed to be executed. This is followed by 126 nulls. Then the actual code begins. For the code to be executed, the address and the 126 nulls must be stripped off, and the code must be moved to the specified address. The extension COM will not be added to the output file name -- it is left with no extension.)

4) You cannot start the program with a /G command. You may only exit with /E.

You use the offset load mode to load code into the buffer and then store it in a HEX file. You can later load the HEX file and execute the program.

You may use the /A, /C, /D, and /P options to force the loading address of any section of code as long as the forced address is at or above the address given in the /O command. The amount of loading space is the same as in the normal mode except that now the available locations start at the address given in the /O command instead of starting at location zero.

INPUT FILES

An input file may be preceded by a device specification and may be followed by one or more options. The extension is always forced to REL. The file is loaded as soon as its name is encountered in the command line. Any options immediately following a file name apply to that file.

The linker does several operations when it loads a module:

1) If /V ("verbose") has been specified it prints the module name.

2) Entry points (if any) are added to the entry point symbol table.

3) The sizes of the common, program, and data sections are defined. If /V has been specified the sizes and section addresses are printed.

4) The code is loaded.

5) External symbols (if any) are added to the external symbol table.

6) External symbols are resolved (if possible) and removed from the external symbol table.

External symbols are resolved as follows: The linker takes a symbol from the external symbol table and tries to find a match for it in the entry symbol table. If it finds a match it sets all of the corresponding external references to the address value found in the entry symbol table and removes the external symbol from the table. If it cannot find a match it skips the symbol and goes on to the next one.

Note that some space is taken up by unresolved symbols but when all the modules in a program are loaded and all external symbols are resolved the external symbol table is empty and the space it occupied is reclaimed.

LOAD MAP

The /M option prints a load map, that is, a list of entry points (to date) with their absolute addresses followed by a list of unresolved external symbols (marked with asterisks). The map tells you which entry points have been loaded (and where) and which have not. The map is normally printed on the console. However, if you follow the /M with a colon and a file name the map is stored in a file with that file name. The extension is set to MAP. For example, the following command line generates a file called TRIG.MAP on drive B:

```
/M:B:TRIG
```

Note that if generating a load map when linking a program with a collection of library subroutines, the library must be specified in the command line; the default library will not be linked in automatically.

UNRESOLVED EXT SYMBOLS

The /U option prints a list of the unresolved external symbols. Each symbol is preceded by an asterisk (*) to mark it as unresolved. Like the /M option, the /U option normally prints the list on the console but it may also take a file name and generate a file with the extension MAP.

SYMBOL TABLE

The /Y option sets up a request to generate a symbol table. The table is generated when a /G or /E command is given, that is, when everything has been loaded. The symbol table is in the same format as the load map. Like the /M option, the /Y option normally prints the symbol table on the console but it may also take a file name and generate a file with the extension SYM.

SEARCH

Library files usually contain more modules than are needed by a program being linked. The search option (/S) tells the linker to load only the modules needed to resolve pending external symbols. For example, you might type:

```
NAVIGAT TRIG/S
```

The main program, NAVIGAT.REL, is loaded first. It has probably generated some entries in the external symbol table. The library, TRIG.REL, is loaded next in search mode. This means that as each library module is read in its entry symbols are compared with external symbols from the table. If a match is found the module is loaded. If not it is skipped. This continues until all the modules in a library have been processed.

Note that a module in the library which is loaded may add more external symbols to the table. These symbols will (hopefully) be resolved by subsequent modules in the library. This is very useful, especially with Pascal, since you end up with a minimum run-time package.

OUTPUT FILES.

The linker's primary task in the load mode is to load a program into memory. After the program is loaded you may save it in either COM file or in a HEX file. You may request a COM file with the /N option ("name") or a hex file with the /H option ("hex") followed by a colon (:) and the file name. The extension is forced to COM or HEX. Notice that this is only a request for an output file. The file is not actually generated until the linker executes a /E or /G option (see below). Therefore, you may rename the output file as often as you like and you may even change its type. For example, you might type:

```
/N:NAVIGAT      Request NAVIGAT.COM.
/N:B:FLY        Change it to FLY.COM on drive B.
/H              Change it to FLY.HEX on drive B.
```

The linker keeps track of the lowest and highest addresses into which it has actually loaded code. When it saves a program it uses the lowest and highest addresses as bounds and saves everything in between. (A COM file always starts at location 100H and therefore always uses 100H as the lowest address). This means that if your program reserves some space at the very end with the DS instruction those locations are not saved as part of the output file (this saves some space).

If the linker is operating in the offset load mode it cannot generate a COM file, only a HEX file.

After you have finished loading a program you leave the linker with either the /E ("exit") or /G (exit and "go") options. The linker does several operations when it executes either of the options:

1) It checks the external symbol table. If it finds any unresolved external symbols it loads the library file, usually LIB.REL, (on the logged-in drive) in search mode (/S).

2) It checks the external symbol table again. If it still finds any unresolved external symbols it prints a list of the external symbols, ignores the remainder of the command line, and prompts you for a new command line.

3) It generates a COM or HEX file if one has been requested.

4) It generates a symbol table.

5) It sets up the block move routine (at location 80H).

6) It sets either the program starting address (for /G) or zero (for /E) into the jump address in the block move routine.

7) It prints the lowest, highest, and starting addresses of the program.

8) It executes the block move routine (which moves the program down to the location at which it will execute) and jumps to either the beginning of the program (/G) or to the operating system via location zero (/E).

At this time the linker is no longer in memory. It has been replaced by your program.

If the linker is operating in the offset load mode it stops after step 4 and returns to the monitor without moving any code.

Commands may be given to the linker on the same line in which it is called:

```
LINK FILE1 FILE2 FILE3
```

and/or they may be given while the linker is operating. The command line is read and executed from left to right, entry by entry. An entry is everything in the line up to but not including the next file name. Any options following a file name are part of that entry and are executed with it. For example:

```
/A FILEB/B/C FILED
```

The first entry is just the option /A. The second entry is the name, FILEB, and its options, /B and /C. The third entry is just the file name, FILED, with no options. Note that in this example A, B, and C are arbitrary options.

The linker has two operating modes: the librarian mode and the load mode. You may enter the librarian mode by typing /L:filename as the very first command given to the linker. If you start with anything else you enter the load mode. You may enter the offset load mode by typing /O:XXXX as the very first command given to the linker. XXXX is the lowest address that you want to load code into.

At any time you may restart the linker by typing /R ("restart"). This clears all internal tables and tells the linker to forget that it has read, loaded, or written any files. It also ignores the remainder of the command line.

Input file names are not preceded by anything and may be followed by one or more options. The input file extension is always forced to REL.

Output file names follow colons (:) in options. The output file extension is forced to COM, HEX, MAP, SYM, or REL.

A file name may contain a drive specification. For example:

```
INPUT1/M:A:LIST B:INPUT2/H:OUTPUT
```

The following options are valid in the librarian mode:

/L:NAME Open a library output file named NAME.REL. Once this option is given and the file is opened the name is fixed. It cannot be renamed with this option.

/Q Ask, module by module, if it is OK to keep each of the modules in this library file. You may respond with Y (yes: accept this module), Q (quit: ignore the remainder of this file), A (open an auxiliary input file), or any other key (skip this module). This option takes effect while its input file is read.

/E Close the library file and return to the monitor. This option takes effect after its input file (if any) is read.

The following options are valid in the load mode:

/O:XXXX Load in the offset mode. XXXX is the lowest address available for loading.

/S Load a file (probably a file already created by the linker in the librarian mode) in search mode. Load only the modules needed to resolve symbols in the external symbol table. Skip the rest. This option takes effect while its input file is read.

/V Print the name of each module loaded and the locations and sizes of each of its sections. This option takes effect before its input file (if any) is read and stays in effect forever.

The address setting options takes effect before their input files (if any) are read.

/A:XXXX Load all sections (unless otherwise specified) starting at location XXXX.
/C:XXXX Load all common sections starting at location XXXX.
/D:XXXX Load all data sections starting at location XXXX.
/P:XXXX Load all program sections starting at location XXXX.

/H:NAME Request a hex file called NAME.HEX.
/N:NAME Request a COM file called NAME.COM.

The map and list options take effect after their input files (if any) are read.

/M:NAME Generate a map of all symbols in both symbol tables and send it to a file called NAME.MAP. If :NAME is missing print the map on the console.

/U:NAME Generate a list of unresolved external symbols and send it to a file called NAME.MAP. If :NAME is missing print the list on the console.

/Y:NAME Generate a symbol table just before a /E or /G command is executed and sent it to a file called NAME.SYM. If :NAME is missing print the table on the console.

/E Load the library file (usually LIB.REL), if necessary, generate an output file if requested, move the program down to its actual execution location (if not in the offset load mode), and return to the monitor. This option takes effect after its input file (if any) is read.

/G Do the same as /E except start the program instead of returning to the monitor. The /G option is not available in the offset load mode.

Bad address	/A, /C, /D, /O, or /P is not followed by a colon. The remainder of the command line is ignored.
Bad command character	There is an extraneous character in the command line, probably a punctuation character. The remainder of the command line is ignored.
<name> is a bad EXT chain	An address link in an external symbol chain points to a location outside of the available program area. The relocatable input file may be bad or some code may have overwritten part of the chain. This is a fatal error.
Bad file name	There is something wrong in the file name specification. The remainder of the command line is ignored.
Bad input file	The input file does not make sense as a REL file. A read error may have occured or the file may not actually contain relocatable code. The remainder of the command line is ignored.
Bad option	An option was specified which is not valid in the current mode. The remainder of the command line is ignored.
Can't find <name>	The specified input file does not exist on the specified device. The remainder of the command line is ignored.
Can't open output file	The specified output file cannot be open on the specified device. The device may be full or not operating. The remainder of the command line is ignored.
Code below lowest address	You have attempted to load code below the lowest available address (usually in the offset load mode).

	This is a fatal error.
Code overwrites tables	There is not enough memory to hold the linker, its symbol tables, and your program. You lose. This is a fatal error.
Entry point symbol redefined	The linker tried to load a module containing an entry point symbol identical to one already in the entry point table. It may be trying to load the same module twice.
Error writing file	A write error occurred while writing an output file. It may not be valid. You may have to write it again. The remainder of the command line is ignored.
Name too long	There are too many characters in the file name. The remainder of the command line is ignored.
No COM file in offset mode	You cannot generate a COM file in the offset load mode because the load image in the buffer is not the same as a COM file memory image. You can generate a HEX file instead.
No GO in offset mode	You cannot execute a /G command in the offset load mode because the load image in the buffer is not moved down (or up) after everything has been loaded. You can only generate a HEX file and load it later.
No output file	You attempted to read an input file in the librarian mode without opening an output file. The remainder of the command line is ignored. Open it now with the /L:filename option. You attempted to exit the linker in the load mode with an output file request pending but no output file name. The remainder of the command line is ignored. Specify the output file now with the

	/H:filename or /N:filename option.
Second common larger	You loaded a module which attempted to define an existing common to a larger size. This is not allowed. If a module contains a common section which has already been defined (by a previous module) it may use less than or all of that common space but not more. This is a fatal error.
Starting address redefined	You loaded two modules which both have starting addresses. You may have loaded the same one twice. Only one starting address may be specified in a program.
Too many commons	More than 15 commons have been defined. This is a fatal error.
Undefined common	A module tried to reference a common section which has not been defined. The input file is defective. It may not actually contain relocatable code.
Undefined REL entry	A module contains a relocation instruction which the linker does not understand. The relocation instruction is ignored. The input file may be defective or it may contain a feature which this version of the linker cannot handle.

LINKING UNDER THE K3 OPERATING SYSTEM

There are several differences between using a K3 operating system and using a CP/M operating system.

1) You must tell the operating system to run the linker with the command R LINK. You may type a string of commands on the same line if you wish.

2) A SAV file is created instead of a COM file.

3) A SAV file may be created in the offset load mode.

4) The K3 operating system is able to start a program no matter where it is located in memory. It also has a sufficient stack space for most programs. Therefore, no code is automatically generated at location 100H to set the stack and jump to the start of the program. You may load code into location 100H if you wish.

5) The default loading address is 70H.

6) The code that moves the program from the offset buffer to its final execution address (LDIR, JMP start) is stored at location 3.

A command to link a test program from DK0:, print a load map on the line printer, and save the program in a SAV file on DK2: is as follows:

```
.R LINK TSTPRG/M:LP:/N:DK2:TEST/E
```