TDL Z80 LINKER

User's Manual

(Manual Revision 0)
March 24, 1978

Written by David S. Hirschman

Table of Contents

1 Introduction

LINKER is a TDL utility program that can bind together individually compiled modules of a program into a single file that may be loaded and executed by the CP/M * operating system.

There are many advantages to the practice of linking together separately compiled modules instead of working with a single, large program. A large program may be decomposed into small modules which may be edited and compiled more quickly. For example, to correct a bug, the programmer need only re-compile the affected modules and re-link the program, instead of re-compiling the entire program. Generally, the linking process is faster than compilation.

It often happens that a routine is used in several programs, a special I/O routine or COSINE function, for example. Instead of copying the source code for this routine into each program, it may be compiled once and then linked in wherever it may be required. Furthermore, using LINKER, routines written in different languages may be combined into a single program.

The Z80 Macro Assembler and FORTRAN IV can produce "libraries", or files containing more than one separately compiled module. LINKER offers methods for including all or only some of the modules in a library into the program.

The remainder of this guide describes how to use LINKER. An overview of LINKER concepts and operation is offered in section 2. The input format to LINKER is defined in section 3.

* CP/M as it appears in this manual is a Registered Trademark of Digital Research.

2 Overview of LINKER Operation

LINKER takes as input, FILEs which contain one or more separately compiled MODULEs. Files containing many modules concatenated together are referred to as LIBRARYs.

Each module has a name. In Z80 Assembler, the .IDENT pseudo operation is used to declare the module name. Its use is highly recommended, as the default module name is ".MAIN.", and duplicate module names in a program are not allowed. The other translators assign a module name automatically.

Each module is made up of SEGMENTs (also called "relocation bases"). Segments are the basic units of code and/or data involved in the linking process. After LINKER is aware of what modules are to be included in the program, it assigns an absolute memory address to each segment in each module. Any code in each segment is relocated so that it will execute at the address to which it is assigned.

Several kinds of segments may be contained in a module. The main code segment, usually containing all of the executable code in the module, has the same name as the module itself. The main data segment of each module also has the same name as the module, preceded by a quote ('). For example, a module named ARCTAN would contain a code segment named "ARCTAN" and a data segment named "'ARCTAN".

All of the other segments in each module are common areas, usually containing only data, which may be shared by other modules. One of these segments is named ".BLNK.", and is referred to as the "unlabeled common". This is the common block that will be created by FORTRAN when the programmer doesn't supply a specific name for a common block. All of the other common blocks have names specified by the programmer.

One of the major features of the LINKING process is that each separately compiled module may access code and data defined in other modules. An INTERNAL symbol is one whose address is available to modules other than the one in which it is defined. Symbols which are not INTERNAL are invisible to other modules. An EXTERNAL symbol is one which is used in a module, but is actually an INTERNAL symbol in another module. All EXTERNAL references must be satisfied by INTERNAL declarations in another module, with two exceptions: symbols may be explicitly defined using the /DEFINE option (section 3.7), and some symbols are pre-defined by LINKER (see Appendix B)

An ENTRY point is an INTERNAL symbol which comes into play in library search mode. In this mode of operation only those library modules having ENTRY points which are referenced as EXTERNAL symbols by one or more already linked modules are included in the program (see section 3.6).

3 LINKER Input Format

MODULE, SEGMENT, and SYMBOL identifiers

An identifier is a string of characters from the following set:

A-Z, 0-9, #$%*&'?+-\^~{|}!.:<>[]_

Normally, an identifier consists of no more than six characters. However, an identifier for a .DATA. segment of a module (as discussed in the previous section) is preceded by a quote (').

Identifiers may not contain blanks. Lower case letters, when used, are automatically translated into upper case. The first character of an identifier may not be a number 0 - 9. The following are examples of valid identifiers:

```
PROG5A
SORT-3
'SORT-3     (a .DATA. segment name)
FOO$$$
```

The following are not valid identifiers:

```
34ABC    - an identifier may not begin with a number
CHECKERS - too many characters
NIM A    - contains a space
```

FILE NAME

A file name has the following format (with brackets [] indicating optional portions):

[device:]name[.extension]

The "device:" indicates on what disk drive the file resides. If present, it must be one of "A" through "P". If omitted, the logged-in disk is assumed. If LINKER can't locate an input file on the specified disk, it will try drive A.

The file "name" is required, and must consist of no more than eight characters from the character set given above for identifiers, except that the characters <>.:[]_ may not be used.

The ".extension" indicates what the type of the file is. It may consist of no more than 3 characters, from the same set of characters allowable in the file "name". The defaults for ".extension" are defined on page 7.

16 BIT VALUE

A 16 bit value may be expressed as a literal or as a number. A literal is one or two characters enclosed in quotes, for example: "V1".

A number may be expressed in several different bases, as shown in the table below. A radix character immediately following the number indicates which number system is being used:

Base	Radix	Valid Digits	Valid Range
hex	H	0-9 , A-F	0 - 0FFFF
decimal	.	0-9	0 - 65535
octal	O	0-7	0 - 177777
binary	B	0 and 1	16 digits

If the trailing radix character is omitted, "H" (hex) is assumed. All numbers must begin with a numeric digit (0-9). A preceding minus sign indicates a negative number. In this case, a two's complement representation is used.

The following are examples of 16 bit values:

```
14170    - an octal number
0C1B5    - a hex number
-55.     - a negative decimal number
"A"      - a one character literal
```

The following are not valid 16 bit values:

```
100000.  - decimal number too large
960      - invalid octal digit
"AB      - missing closing quote
C1C2     - does not begin with a digit
```

INITIATING LINKER

LINKER may be used interactively, or input may be given as it is executed:

LINKER <commands> <cr>

This means that LINKER may used in a SUBMIT file.

To use LINKER in the interactive mode, simply enter

LINKER <cr>

on the console. LINKER will read commands from the console, prompting with an asterisk "*". All input is stored uninspected until a carriage return is typed. The standard line editing features of CP/M * (rubout, CTL-U, CTL-C, CTL-E, etc.) are available. If a CTL-T is found on any line, the entire command being entered is aborted.

A disk file containing all or only part of a command may be inserted into the input at any point by preceding the disk file name with an "@". The default file extension is ".LNK". These disk files may not contain further "@" specifications. The most common use of this feature is to prepare a file containing a complete command; then,

LINKER @<file name> <cr>

links the program. Usually, these ".LNK" files may be prepared once for a given program and used over and over again, greatly simplifying the whole process.

All LINKER commands have the same format, regardless of whether the interactive mode is used. Commands are separated by a semi-colon ";". LINKER terminates when it receives the "Q" command (quit). For example,

<command> ; <command> ; <command> ; Q

LINKER also terminates when input provided with its execution is exhausted.

If an error is found, the current input line is echoed with two question marks inserted after the point at which the error was detected. This is followed by an error message (see Appendix A). The command must then be re-entered.

All input is free format. Blank lines are ignored, and a command may extend to any number of lines. All lower case letters are automatically translated to upper case. Comments may be included with input from any source by using an asterisk "*". When an asterisk is encountered, all remaining characters on the same line are ignored.

If a CTL-C is typed while LINKER is running, it will quit and return to the monitor. If CTL-E is typed, the current command is aborted, and LINKER will prompt for more input if it is being used interactively.

3.1 Command Syntax

Each command to LINKER links one program, and is of the format:

```
[<output file> =]
    <input file 1>, <input file 2>, ... , <input file m>
    /<option 1> /<option 2> ... /<option n>
```

LINKER links together appropriate modules from the input files to create the output file, under control of any options present. If the program is linked successfully, its name is printed on the console, along with the address of the highest byte used in the program and the program size rounded up to the nearest K (1K = 1024 bytes).

3.2 Output File

The output file is the file which will contain the linked program. The file extension indicates what kind of file is to be produced. If given, it must be one of the following:

COM - Absolute binary core-image file, ready to be loaded and executed by the operating system.

HEX - INTEL "hex" format file (see Appendix I of the Z80 Relocating Macro Assembler User's Manual).

If the .<extension> is not given, ".COM" is assumed. The output file replaces any existing file of the same name.

Examples:

B:PROG1	- A .COM file for PROG1 is placed on disk B.
PROG2.HEX	- An INTEL "hex" file for PROG2 is placed on the currently logged-in disk.

The output file and equal sign following it may be omitted; then, the name of the first input file is used, and an extension of .COM is assumed.

3.3 Input Files

Each <input file> may contain either a single compiled module, or may be a library containing many compiled modules. Normally, all modules contained in each <input file> will be included in the output file, but this default action may be overridden as explained below. The <input file>s must contain all modules that are to be included in the output file, unless the /SEARCH option is used (see section 3.6).

If the file extension is not given, ".REL" is assumed. Of course, all files must contain only compiled, relocatable object modules, in either ascii or binary format.

A module selection clause may optionally be added immediately after each input file name, to indicate that only some of the modules within the file are to be linked. It has two possible formats:

(INCLUDE <module 1>, <module 2>, ... <module n>)

which causes only the named <module>s to be included in the output file, and

(EXCLUDE <module 1>, <module 2>, ... <module n>)

which causes all modules in the library EXCEPT the listed ones to be included in the output file.

3.4 /MAIN Option

This option specifies the main module of the program. Its format is:

/MAIN <module name>

The main module must have a defined starting address. This is done in Z80 Assembler by supplying a label with the ".END" pseudo op. The other translators automatically supply a starting address. Execution of the program will begin at this address.

If the /MAIN option is omitted, LINKER looks for a global symbol named .MAIN. and uses this for the starting address if found. If not, the first module encountered in the input files which has a defined starting address is assumed to be the main module of the program.

3.5 /MAP Option

The /MAP Option may be used to obtain a printout of the memory map on the list device. Reports can be selected that show the memory addresses assigned by LINKER to the segments and symbols in the linked program, or that describe the modules that were linked.

The format of the /MAP option is:

/MAP <flag 1> <flag 2> ... <flag n>

The <flag>s control what items will be included in the memory map report, as follows:

G - Global symbols (i.e. all internal symbols of all loaded modules). The symbols are listed in alphabetical order, with their assigned addresses. The address shown is the address that will be used for all references to this symbol. This may not be the same as the address where the symbol is loaded, if the /ACTUAL option is used.

S - Segments. All of the program segments are listed in alphabetical order, and the assigned address and size is given for each. If the segment is to be relocated so that it will execute at an address different from its assigned one, via the /ACTUAL option, this address is given also.

A - All. This option combines the information given by the S and G flags. All segments are listed, in order of ascending memory address. Each segment is followed by all of the global symbols contained within that segment, again listed by ascending memory address. Absolute symbols are listed under a dummy segment named .GLOB.

M - Modules. Each module is listed, along with its ID number, version and revision number, and date and time assembled (older versions of the Z80 Assembler do not output the information needed to generate this report. The .PROGID pseudo op is used to create this information for each module).

If no <flag>s are given, /MAP A is assumed.

3.6 /SEARCH Option

This option causes library files to be searched in order to satisfy external references which remain unresolved after all modules contained in the input files have been linked. The format of the option is:

/SEARCH <library 1>, <library 2>, ... <library n>

Each <library> has the same syntax as the input files of section 3.3. INCLUDE and EXCLUDE clauses may be used.

A module in a library is loaded when one or more of its ENTRY points (see section 2) are referred to by other modules, but have not yet been defined anywhere. As long as undefined symbols exist, all specified libraries are searched iteratively in the order given, until a complete pass over the libraries yields no new modules to be loaded. That is, if loading a library module creates new unresolved symbols, all of the libraries may be searched again in an attempt to find it.

When FORTRAN IV modules are included in a program, the FORTRAN library "LIBRARYS.REL" is automatically added to the end of the list of libraries to be searched. It must be present on the logged-in disk or on drive A. This library is designed to be searched in a single pass, and error #31 (see Appendix A) may result if an additional pass must be made over it. Therefore, it may necessary to design any other libraries that are to be searched so that only a single pass is required to pick up all needed modules.

3.7 /DEFINE Option

This option may be used to give values to symbols which are not defined by any module in the program. These defined symbols are then used to resolve EXTERNAL references made by the program modules.

The syntax of this option is:

```
/DEFINE <symbol 1> = <value 1>,
        <symbol 2> = <value 2>,
                .
                .
        <symbol n> = <value n>
```

Each symbol is given a 16 bit value. This value could represent a constant, or an absolute address.

The following is an example of /DEFINE usage:

```
/DEFINE CONST1=1238., FLAGS = 10110011B, COUNT = 0C1D4,
        VRSION = "A1"
```

There are some symbols which are pre-defined by LINKER. A list of them is given in Appendix B.

3.8 /LOCATE Option

Normally, LINKER assigns memory addresses sequentially to segments as they are encountered in the input files. This option may be used to specify the absolute memory address where a segment is to be located instead of allowing LINKER to choose it. It is useful for accomplishing actions such as locating a common block segment in a video refresh memory area.

The format is:

```
/LOCATE <segment-1> = <address-1>,
        <segment-2> = <address-2>,
                    .
                    .
                    .
        <segment-n> = <address-n>
```

LINKER will assign each segment at the given 16 bit address, and will avoid assigning other segments to the same memory area. However, no check is made to see if two segments are LOCATEd so that they overlap. Also, if a segmented is LOCATEd too low in memory, an error #46 may occur during pass 2 (see Appendix A). If a segment is located too high in memory, error #79 may occur.

A "/LOCATE .DATA. = <address>" will concatenate all of the data segments from each module and treat them as a single segment to be assigned to the given address.

A "/LOCATE .PROG. = <address>" will concatenate all of the main code segments of each module, and assign them to the given address if possible. If a locate of .DATA. is done as well, the program is divided into code and data areas (as long as the programmer creates pure .PROG. segments). If not, all of the program segments, code and data, are loaded beginning at the given address.

3.9 /ACTUAL option

As discussed in section 2, each program segment is normally designed to execute at the memory address at which it is to be loaded. Using this option, however, a segment may be relocated so that it will execute at a different address (presumably, the segment will be moved at run time to the correct location).

The format is:

```
/ACTUAL <segment-1> = <address-1>,
        <segment-2> = <address-2>,
                      .
                      .
        <segment-n> = <address-n>
```

Each segment, which will be loaded wherever it would normally be loaded, will be relocated to execute at the given address. All references from other segments into them will also be relocated.

APPENDIX A - LINKER Error Messages

A few LINKER error conditions are indicated by a short message which should be self-explanatory. For the rest, an error number is given which may be looked up in the table below. In the case of a syntax error, the input line containing the error is echoed, with two question marks "??" following the point where the error was detected. Other errors may be flagged as occurring in PASS 1 or PASS 2.

Many of the error messages involve a problem with a disk file. In this case, the name of the disk file is given, as well as a byte offset (in hex) indicating the position in the file where the error was detected.

Any error codes not appearing in this table are diagnostic errors indicating a bug in LINKER. Try running LINKER again. If the error persists, please collect the relevant information (error message, LINKER version date, input files, etc.) and notify the Technical Assistance Manager at Technical Design Labs.

Error Codes

1 - Expecting equal sign.

2 - Expecting "/" or ";". The command parser has reached the end of the input files, and is trying to read the options.

3 - Bad option name. See sections 3.4 and following.

4 - Option not implemented. The version of LINKER you are using does not contain this option yet.

5 - Expecting identifier. See Section 3 for an explanation of correct identifier format.

7 - Wrong digits in number. Which digits are valid depends, of course, on the radix you are using. See Section 3.

8 - Number or literal too large. All numbers and literals must be able to fit into 16 bits. See section 3.

9 - Token too large. The string of characters you entered at this point is too long to possibly be any kind of valid input.

10 - Expecting "device:" or "file" name. A proper file name should appear in the input at this point (see section 3, file name format).

11 - Invalid "device:" specifier. Valid device specifiers are "A:" through "P:".

12 - Invalid file name. A file name must consist of no more than eight characters from the proper character set (see section 3, file name format).

13 - Invalid file extension. A file extension must consist of no more than three characters. An output file may only have extensions ".HEX" or ".COM".

14 - Expecting 16-bit value. A number or literal must appear in the input at this point.

15 - Incorrect INCLUDE or EXCLUDE format. Either you did not give one of the key words INCLUDE or EXCLUDE, or there is an incorrect module ID, or the closing right parenthesis ")" is missing.

17 - "@" inside @ file. Disk files containing commands and used via an "@" may not contain further "@" specifications.

20 - Insufficient memory. There was not enough free memory available for LINKER to use for its symbol and segment tables. Therefore, the program could not be linked.

31 - Duplicate segment. The indicated segment appears more than once in the input modules. Did you remember to use the .IDENT pseudo op in Z80 Assembler programs? Another way this error can occur is if FORTRAN IV is being used and multiple /SEARCH passes are made over LIBRARYS.REL. See section 3.6.

34 - Undefined segment. A segment which you referred to in the /LOCATE or /ACTUAL options was never encountered in the input files.

40 - Can't close output file. Is the disk write protected?

41 - Error in extending file.

42 - No space for output file. There is not enough space on the disk to hold the output file.

43 - No directory space. The disk upon which the output file is to be placed doesn't have enough room in the disk directory.

45 - Can't open output file. This error may be caused by a full directory, or by a protection failure.

46 - Loading below 100H in .COM file. A .COM file is organized so that the beginning of the file corresponds to memory address 100H, since the operating system always loads a .COM file at this address (see Appendix D). Thus, nothing may be loaded below this address. This error may be caused by a /LOCATE to an address below 100H.

50 - Expecting module record. The input file was supposed to contain a module record at this point, but did not. This error often occurs when there is trash at the end of the previous module in a library file.

51 - Invalid record type. The input file contained an incorrect .REL record type at the indicated offset.

53 - Undefined symbols exist. All of the listed symbols will have to either be made INTERNAL symbols of some module or defined via the /DEFINE option.

54 - Missing starting address. You did not use the /MAIN option, symbol .MAIN. did not exist, and none of the program modules had a defined starting address.

55 - The main module (as given by the /MAIN option) has no defined starting address. Be sure to give a starting address with the .END pseudo op in Z80 Assembler programs.

56 - The main module (as given by the /MAIN option) was never encountered in the input files; therefore, no starting address could be determined.

57 - Can't recognize module. There is garbage in the input file at this point. Are you sure this file is a valid .REL file? If all else fails, try re-compiling.

58 - Can't process FORTRAN. The version of LINKER you are using can't link FORTRAN modules.

60 - Duplicate input file. Each input or library file can appear only once in a command.

64 - FORTRAN symbol number out of range. This and the following two errors usually indicate a smashed FORTRAN .REL file. Try re-compiling.

65 - Bad FORTRAN relocation base type.

66 - Bad FORTRAN op code.

70 - Duplicate symbol. The indicated global symbol is defined in more than one module.

79 - Program won't fit into memory. This program won't fit into the address space of a 16-bit micro-computer. Either it is simply too large, or you created large wasted areas of memory by using the /LOCATE option.

80 - Expecting carriage return. The indicated input file was supposed to have a carriage return at the given location, but did not. Are you sure this is a valid .REL file? Try re-compiling the program if all else fails.

81 - Expecting line-feed in input file.

82 - Expecting ASCII character. The input file did not contain a valid ASCII character where it was supposed to.

83 - Bad Checksum. Z80 Assembler ".REL" files contain checksum bytes after each record which are used to validate the data that is read from them. A checksum error usually indicates a file that is corrupted with errors: try re-compiling.

85 - End of input file. The end of the indicated file was reached unexpectedly.

87 - Empty input file. The indicated input file was totally empty, except perhaps for some filler characters.

Appendix B - Pre-Defined Symbols

There are a few global symbols which are pre-defined by LINKER before the linking process begins. They are listed below. The user should not attempt to define these symbols himself, as a duplicate symbol error (code #70) will result. Future versions of LINKER may have more of these symbols. They will be of the form .XXXX., so the use of symbols of this form should be avoided.

Pre-Defined Symbols

.FREE. - This symbol points to a word which contains the address of the first free byte in memory above the program. It is useful when the programmer wishes to make use of free memory at execution time. When a "/LOCATE .DATA. = <addr>" is done (i.e. data segments are assigned to a separate memory location), .FREE. points to the first free byte above the data area.

If FORTRAN IV modules are included in the program, many other symbols will be defined via modules brought in from LIBRARYS.REL. The reader is referred to the TDL FORTRAN IV User's Manual for details.

Appendix C - Syntax Summary

Below is a brief summary of LINKER input syntax, in a modified BNF format. The symbol "::=" should be read as "is defined to be". Angle brackets "<>" delimit meta-linguistic objects, which are themselves defined in a following line. Square brackets "[]" indicate optional input. Curly braces "{}" indicate input which may be omitted or repeated as many times as desired. A vertical bar "|" indicates a choice - the form preceding or following may be used.

```
<LINKER input>      ::= <command> {;<command>} ;Q

<command>           ::= [<output file> =] <input file>
                                {,<input file>}
                                {/<option>}

<output file>       ::= <file name>
                        (the extension, if included,
                         must be .COM or .HEX)

<input file>        ::= <file name> [<module selection>]

<file name>         ::= [<device>:] <name> [.<extension>]

<device>            ::= "A" through "P"

<name>              ::= a string of no more than 8 characters
                        from <fset>, beginning with one of
                        "A" though "Z".

<extension>         ::= a string of no more than 3 characters
                        from <fset>, beginning with one of
                        "A" though "Z".

<module selection>  ::= (INCLUDE <module> {,<module>})
                      | (EXCLUDE <module> {,<module>})

<option>            ::= <main> | <map> | <search> | <define>
                      | <locate> | <actual>

<main>              ::= MAIN <module>

<map>               ::= MAP [A] [S] [G] [M]

<search>            ::= SEARCH <input file>
                               {,<input file>}

<define>            ::= DEFINE <symbol> = <value>
                                {,<symbol> = <value>}

<locate>            ::= LOCATE <segment> = <value>
                               {,<segment> = <value>}

<actual>            ::= ACTUAL <segment> = <value>
                               {,<segment> = <value>}
```

Syntax Summary

```
<module>        ::= <id>

<symbol>        ::= <id>

<segment>       ::= ['']<id>

<id>            ::= a string of no more than 6 characters
                    from <nset>, which does not begin
                    with a number.

<fset>          ::= A-Z, 0-9, #$%*&'?+-\^~{|}!

<nset>          ::= <fset> and <>.:[]_

<value>         ::= <number> | <literal>

<literal>       ::= "<any one or two characters>"

<number>        ::= [-] 0 - 0FFFF[H]
                        | 0 - 65535.
                        | 0 - 177777O
                        | 0 - 1111111111111111B
```

Appendix D - Program Format

The .HEX or .COM file created by LINKER is constructed so that it will appear in memory as shown in figure 1 when loaded by the operating system. The operating system assumes that all programs will begin execution at address 100H. LINKER therefore places a a 16 byte initialization routine at this address which sets up a stack and jumps to the starting address of the program. This area also contains .FREE., and other fields which are used by FORTRAN, or which may be defined in later versions of LINKER. A 3 byte patch area is included for debugging purposes.

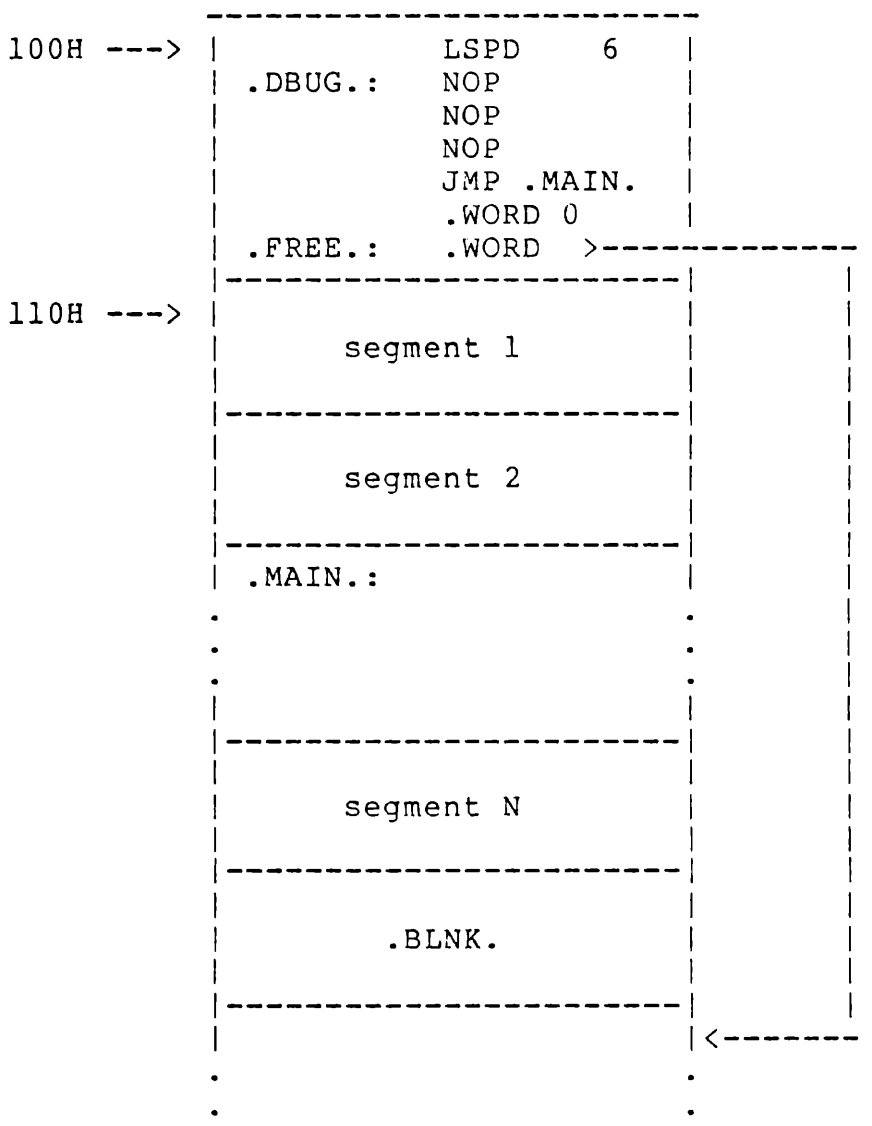

Figure 1

If the initialization routine is not wanted, the /LOCATE option may be used to locate a segment at 100H, overwriting it. Hopefully, this segment would contain the first executable instructions of the program at the very front.

Of course, the program segments may be located anywhere in memory above 100H by using the /LOCATE option

end of the program (or at the end of the data area if a /LOCATE .DATA. is done). Symbol .FREE. points to the first byte following this. Whenever FORTRAN IV is used, module .EMUL. (the emulator) is loaded, and it will be loaded on a page (256 byte) boundry due to an efficiency trick wnich makes use of this fact.

Appendix E - LINKER Examples

Example 1

Suppose you have a program consisting of just one module, contained in file TEST.REL. To produce a file TEST.COM to execute, just type:

LINKER TEST <cr>

Recall that the name of the output file defaults to the name of the first input file (the only input file in this case). This is a simple link, with no memory map or other options. The module must have a defined starting address, and no external symbols.

Example 2

The following command is used to link LINKER itself, and is kept in file LINKER.LNK. Thus, when LINKER must be linked,

LINKER @LINKER

does the job. The command is:

```
linker.com =
    bases, command, console,
    convertr, decimal, diskfile,
    files, fortran, freespac, getfile,
    hash, linker, mainfind,
    map, memory,
    printer, progbild, relread, scanner,
    segments, string, symbols
    /main main /map m a
    /define cpmbas = 100H
    /define month = "3",     *These have to be backwards
            day   = "62",    *since they are stored
            year  = "87"     *as words
    /locate .prog. = 150     *Save patch area
```

There are twenty-three input files, containing one module each - this makes the individual files easy to edit. They all have extension ".REL". The main module of the program is contained in file LINKER.REL, and has module name MAIN. Two memory map reports are to be produced. A symbol CPMBAS is defined, and three symbols MONTH, DAY, and YEAR, which are used to print out the date when LINKER is executed. Thus, the date may be changed without any re-compilation. .PROG. is located at address 150H, which leaves a 64 byte "hole" between there and 110H where the initialization routine ends (see Appendix D). This space is reserved in the event that bugs have to be fixed.

Appendix F - Using LINKER with Z80 Assembler

This appendix is a list of hints which may be of help in setting up Z80 Assembler modules for use with LINKER.

SYMBOLS

Internal and External symbols are created by using the .INTERN and .EXTERN pseudo operations. .ENTRY is used to create entry-point symbols.

SWITCHES

When assembling a module for use with LINKER, do not use the .PABS or .XLINK switches. Do use the .PREL and .LINK switches (these are defaults). You may use the .PHEX switch to get an ASCII .REL file, but using .PBIN (the default) will result in a savings of disk space.

MODULE NAME

Always use the .IDENT operation to give each module a unique name. If you don't, the module will have name .MAIN. Each module in a program must have a unique name.

STARTING ADDRESS

A label should be supplied with the .END pseudo op to define the starting address of the main module of the program. Then use the /MAIN option of LINKER to indicate the main module. Alternatively, make the starting address .MAIN., and declare .MAIN. as .INTERN (FORTRAN does this automatically).

LIBRARIES

Libraries may be created by using the .PRGEND switch. This results in the creation of a new module starting at that point. PIP may be used to create libraries as well, but use the O (object) switch for binary .REL files, and do not use this switch for ascii .REL files. FORTRAN and Z80 Assembler modules, and binary and ascii modules may be mixed together in a library.

MEMORY MAP

If the M report of the memory map is wanted, use the .PROGID pseudo op to define the program name, version number, and revision number.

COMMON BLOCKS

To make a common block, declare the common block name to be an .EXTERN in each module that must reference it. The common should not be declared .INTERN by any module. Then, use .LOC to define the common. For example,