# LINKER

## CDL Z80 LINKER

## User's Manual

Version 1.0

Sept 30, 1979

# Table of Contents

# 1 Introduction

LINKER is a CDL utility program that can bind together individually compiled modules of a program into a single file that may be loaded and executed by the operating system. INTEL HEX and CDL REL files may be created as well. LINKER is available in versions for the CP/M (*) and TPM (*) operating systems, and requires a Z80 processor with at least 32K of memory and a floppy diskette drive.

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 programmer can gradually build up a library of general purpose routines, and avoid the useless effort of solving the same programming problems over and over again.

Using a linkage editor eases the problems encountered when several programmers must work together on a large program. Once the programming problem has been broken down into separate modules, each programmer may work relatively independently.

CDL's Z80 Macro Assembler and FORTRAN 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.

LINKER is a two-pass linkage editor. That is, each of the input modules is read twice. Since the output file is built up on disk, LINKER has the ability to create 60K programs that completely fill the address space of the machine, unlike other linkage editors which require that the output file share memory space with them and their tables. It is even possible to use a small machine to create programs intended for execution on a larger one.

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. Subsequent sections describe options which cause LINKER to deviate from the standard linkage edit process.

(*) CP/M is a trademark of Digital Research and TPM is a trademark of Computer Design Labs.

## 2 Overview of LINKER Concepts

LINKER accepts as input "relocatable" or REL files. The term "relocatable" refers to the fact that the machine code in these files may be "relocated" by the linkage editor; that is, the code can be made to execute at any memory address in the computer. Thus, the programmer is freed from having to choose memory locations himself, and the same code can be loaded at different addresses in different programs. Each REL file may 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. FORTRAN assigns module names 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 linkage edit. After LINKER is aware of what modules are to be included in the program, it assigns memory addresses 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 linkage edit 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 4.4), 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 4.3).

# 3 LINKER Input Format

## MODULE, SEGMENT, and SYMBOL identifiers

An identifier is a sequence of letters, numbers, or special characters, except for the following:

```
"(),/;=
```

Normally, an identifier consists of no more than six characters. However, an identifier for the .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     - begins with a number
CHECKERS  - too many characters
NIM A     - contains a space
PROG/1    - contains an illegal character
```

## FILE NAME

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

```
[device:]name[.type]
```

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 ".type" 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 following are examples of legal file names:

```
PROG1.REL
PROG2
A:CHESS.COM
```

The following are incorrectly formed file names:

```
Q:FILE.REL    - invalid drive name
PROG.1.2      - contains 2 periods
CHESS.PROG    - file type is too long
```

## 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
11B       - 11 binary
```

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>

where <cr> means to press the RETURN key. 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.

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 type 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 operating system. 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 type 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 E of the CDL Z80 Macro Assembler User's Manual).

REL - CDL standard relocatable object file. With this file type, the input files are in effect merged together to create another linkable file which could serve as input to another linkage edit. See section 6 for details. The /XLINK option may be used to create a non-linkable relocatable file.

If the .<type> 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. |

If the output file name (and following equal sign) are omitted, the name of the first input file is used for the output file, and a type 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.

If the file type 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 General Purpose Options

This and the following sections describe options that may be specified to alter the linkage edit process. While those of the following sections may be used only with certain output file types, these may always be used.

## 4.1 /MAIN

This option specifies the main module of the program. When a .COM file is created, execution of the program will begin at the starting address defined for the main module. For other types of files, the starting address is written at the end of the file (see Appendix E of the CDL Z80 Macro Assmbler User's Manual).

The format of this option 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. FORTRAN automatically supplies a starting 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. It is error if no such module can be found.

## 4.2 /MAP Option

The /MAP Option may be used to obtain various reports on the list device which describe the results of the linkage edit. 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 included.

The format of the /MAP option is:

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

The <flag>s select the desired reports, 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. Symbols created via the /DEFINE option or automatically supplied by LINKER listed separately.

M - Modules. Each module is listed, along with its ID number, version and revision number, and date and time assembled. This information is available only for CDL Z80 Macro Assembler modules, and is created by using the .PROGID pseudo op in those programs.

E - External Symbols. This option may be used only when a linkable REL file is being created. External symbols and their assigned relocation base numbers are printed in alphabetical order.

If no <flag>s are given, /MAP A is assumed. When a REL file is created, the memory map reflects the fact the addresses are no longer absolute, but are relative to a relocation base number. Relocation base 1 is the .PROG. segment, and is indicated by a single quote (') following the address. Relocation base 2 is the .DATA. segment, and is indicated by a double quote (") following the address. Other bases are used for common blocks and external symbols, and are indicated by a ":nn" following the address, where "nn" is the base number.

Symbols in the A report are given with the segment they were defined within. Addresses relative to that segment are indicated with a single quote, while addresses of absolute symbols have nothing following them.

If a REL file is created with the /XLINK option, all relocatable quantities are relative to relocation base 1, and are indicated as such by a following quote. All of these conventions are similar to those used in CDL's Z80 Macro Assembler.

## 4.3 /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. Note that /EXTERN symbols are not searched for. The format of the option is:

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

Each <library> has the same syntax as regular input files, and INCLUDE or 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 them.

When FORTRAN 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.

## 4.4 /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.

## 4.5 /ASCII

HEX and REL files may be created in two formats: INTEL HEX and binary. In INTEL HEX format, the entire file is in printable ASCII with all hex characters expanded to 2 ASCII characters, and carriage return and line feed characters inserted periodically, enabling one to print these files on a console or printer. However, the INTEL format required much more diskette space, and is normally used only for serial devices such as paper tape and cassette.

HEX files are created in INTEL HEX format by default. REL files are created in binary format by default, but the /ASCII option may be used to force an INTEL file to be built. It has no arguments.

Complete information on the format of HEX and binary files may be obtained in appendix E of CDL's Z80 Macro Assembler User's Manual.

## 4.6 /BINARY

This option is the converse of the /ASCII option above, and forces HEX and REL files to be created in binary format. This is the default for REL files.

## 4.7 /NOLOAD

This option is used when one or more modules are to be included in the linkage edit, but none of the code and data contained in them are to be written to the output file. That is, the internal symbols of these modules are defined, and space assigned for all segments, but a hole is left in the output file. The format is:

.NOLOAD <module>, <module>, ... ,<module>

The purpose of this option is to provide a primitive overlay capability. One linkage edit is used to create the main program, with all overlayed modules deleted via the /NOLOAD option. Separate files for each overlay are created by using /NOLOAD on the main program. Thus, the program files are kept small, but all inter-module symbol references are maintained. At run time, an overlay loader in the main program reads in the overlay files as required.

# 5 Non-relocatable File Options

The options of this section may be used only in COM, HEX or REL files with the /XLINK option set.

## 5.1 /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.

LOCATEs of individual segments always override a LOCATE to .PROG. or .DATA.

## 5.2 /ACTUAL

As discussed in section 2, each program segment is normally relocated 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.

If an ACTUAL of .PROG. is done, all .PROG. segments are assigned sequential execution addresses starting with the given one. An ACTUAL of .DATA. has the same result with the .DATA. segments. ACTUALs of individual segments override these global ACTUALS.

## 5.3 /XLINK

Normally, when an output file type of REL is specified, a linkable file is produced as described in the following section. When this option is used, however, the linkage information is not placed into the output file, creating a non-linkable variant of a REL file which is still relocatable. This file format is recognized by CDL's ZAPPLE monitor, a simple non-disk operating system. CDL's interactive debuggers, DEBUG I and II, also recognize this file format. Each of these programs offer the ability to load the REL file at any address in memory, unlike a COM file which is always loaded at address 100 hex.

/XLINK has no arguments, and may be used only when the output file type is REL. All segments are merged together and assigned relocatable addresses beginning with zero (unless overridden by the /LOCATE or /ACTUAL options). Relocatable addresses are indicated in the memory map reports by following them with a single quote (').

# 6 Linkable File Options

The options of this section are provided to assist in the creation of a linkable output file. That is, the output file will be indistinguishable from an ordinary linkable file produced by CDL's Z80 Macro Assmbler, and may be used as input to a subsequent linkage edit. In effect, the input REL files are "merged" together to create an output REL file.

To accomplish this, the output file type must be .REL, and the /XLINK option must not be used (it causes a relocatable but non-linkable file to be produced).

When a linkable file is created, the following actions occur by default:

1 All .PROG. segments of all modules are concatenated together to produce the .PROG. segment of the output module.

2 All .DATA. segments and common blocks are concatenated together to produce the .DATA. segment of the output module. The .BLNK. common, however, is output by itself. Other commons are output by themselves only when included in the /COMMON options.

3 On the memory map, .PROG. segments are given addresses relative to relocation base 1, .DATA. segments to relocation base 2, and the .BLNK. common to relocation base 3. Commons mentioned in the /COMMON option, and external symbols created with the /EXTERN option, are assigned higher relocation base numbers.

The following options are used to create internal, entry and external symbols, create common blocks, and define other parts of the linkable output file. They are similar in their action to the similarly named pseudo operations provided by the Z80 Macro Assembler.

## 6.1 /IDENT

This option is used to create the output module name. The format is:

/IDENT <module>

If omitted, a module name of .MAIN. is assumed.

## 6.2 /PROGID

This option is used to define the program identification information which is listed in the M report in the memory map. The format is:

/PROGID <name> [,<version> [,<revision>]]

with optional input indicated by brackets []. The <name> is a 6 character identifier, and the <version> and <revision> are values lying in the range 0 - 255. The <name> defaults to blanks, and the version and revision numbers default to zero.

## 6.3 /COMMON

As explained above, common blocks are normally merged in with the .DATA. segments when a linkable output file is created, and are then invisible to other modules the output file may ultimately be linked with. The /COMMON option is used to force common blocks to remain separate and linkable in future linkage edits. The format is:

/COMMON , , ..., 

## 6.4 /EXTERN

This option is used to create external symbols, or symbols that are not defined by any modules in the linkage edit. Of course, these symbols will have to eventually be defined when the output module becomes input to a subsequent linkage edit. The format is:

/EXTERN <symbol>, <symbol>, ... , <symbol>

All symbols which are not /EXTERNs must be defined by some module in the program, must be /DEFINEd symbols, or must be one of the special symbols automatically defined by LINKER.

## 6.5 /INTERN

This option is used to define those symbols which will be global in the output file. All other symbols will be invisible to other modules when the output module is used in a subsequent linkage edit. The format is:

/INTERN <symbol>, <symbol>, ... ,<symbol>

## 6.6 /ENTRY

The /ENTRY option is used to create entry symbols in the output file. Entry symbols are just like /INTERN symbols, but they are also sensitive to library search mode. That is, if the output module were to be part of a library search in a subsequent linkage edit, the module would be included if the /ENTRY symbol was undefined but needed by other modules.

The format is:

/ENTRY <symbol>, <symbol>, ... <symbol>

## 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.

Errors marked as "diagnostic" indicate that a bug in LINKER has occurred through no mistake on the user's part. Try running LINKER again, in case the error was a temporary hardware failure. If the error persists, please collect the relevant information (error message, LINKER version, input files, etc.) and write the manager of Technical Services at

Computer Design Labs
342 Columbus Ave.
Trenton, N. J. 08629

### 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 an identifier. See Section 3 for an explaination of correct identifier format.

6 - Invalid options for output file type. You used /ACTUAL or /LOCATE in a linkable REL file, or /INTERN, /DEFINE, /EXTERN, etc. in a non-linkable file.

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 type. A file type must consist of no more than three characters. An output file may only have types HEX, COM or REL.

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.

16 - .DATA. may not be used in the /COMMON option. All .DATA. files are merged together when a linkable REL file is created.

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

18 - Expecting byte value. The value used by this option must lie in the range 0 - 255 (or -128 - 127 in signed interpretation). That is, it must be possible to represent the value in 8 bits.

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.

30 - (diagnostic) Relocation base was undefined.

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 4.4.

32 - (diagnostic) Segment not found.

33 - (diagnostic) End of segment table.

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

35 - (diagnostic) Segment was defined twice.

36 - (diagnostic) Undefined segment address.

37 - Too many external bases. You have too many external symbols (/EXTERN option) and common blocks (/COMMON option) defined. The maximum total of both allowed is 251.

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

41 - Output file too large. The output file exceeded the maximum file size allowed on your system.

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 already contains the maximum number of files allowed.

44 - Protection failure. A previous version of the output file is already on the disk and can't be deleted because it is protected.

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 C). Thus, nothing may be loaded below this address. This error may be caused by a /LOCATE to an address below 100H. Also, you may have done a .LOC to an address below 100H in a Z80 Macro Assember program.

47 - Invalid external byte reference. In the Z80 Macro Assmbler module named, you made an external byte reference (whose relocation base number is supplied) to a non-absolute symbol. External byte references may be made only to symbols defined with an absolute address having a high order byte equal to 0 or 0FFH (i.e. a byte value in the range 0-255 or -128 - 127).

49 - (diagnostic) Too many files opened, or too many buffers allocated.

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, or named in the /EXTERN 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.

61 - (diagnostic) End of input file table.

62 - (diagnostic) Module not included or excluded.

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.

67 - (diagnostic) Fortran code loader failed.

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

71 - (diagnostic) Symbol not found.

72 - (diagnostic) End of symbol table.

73 - Symbol is self defined. You defined the given symbol relative to a symbol in another module, and that symbol to yet another symbol, and so on, until the original symbol was reached again. Thus, a circular chain was created, with no symbol actually ever being defined.

74 - .MAIN. was an absolute symbol. The starting address of a program must be relative to the main module of the program, not an absolute address.

75 - The starting address of the main module was an external symbol to that module. That is not allowed! The starting address must be inside the main module.

76 - /EXTERN, /INTERN or /ENTRY conflict. The same symbol can not be an /EXTERN and also be an /INTERN or /ENTRY. The first is undefined in the output module, while the second is defined.

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. This error is just a warning: LINKER will attempt to continue with the linkage edit.

85 - End of input file. The end of the indicated file was reached unexpectedly. Did you use a file copying program on it which assumes that a ^Z is the end of the file? This typically results in the truncation of binary REL files. Try re-compiling.

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 linkage edit 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

.END. - This symbol has as its value 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. is done (i.e. data segments are being assigned to a separate memory location), .END. is set to the address of the first free byte above the data area.

.FREE. - This symbol points to a word which contains the value of .END. .FREE. is located at address 10C hex, in the initialization area for COM files (see Appendix D). It is not created for other types of files.

If FORTRAN 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 CDL's SSS FORTRAN User's Manual for details.

## Appendix C - COM File Format

COM files created by LINKER are constructed so as to appear as shown in figure 1 when loaded into memory 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. This initialization code is not created for any other kind of output file. See Appendix E in the CDL Z80 Macro Assembler User's Manual for the format of HEX and REL files.

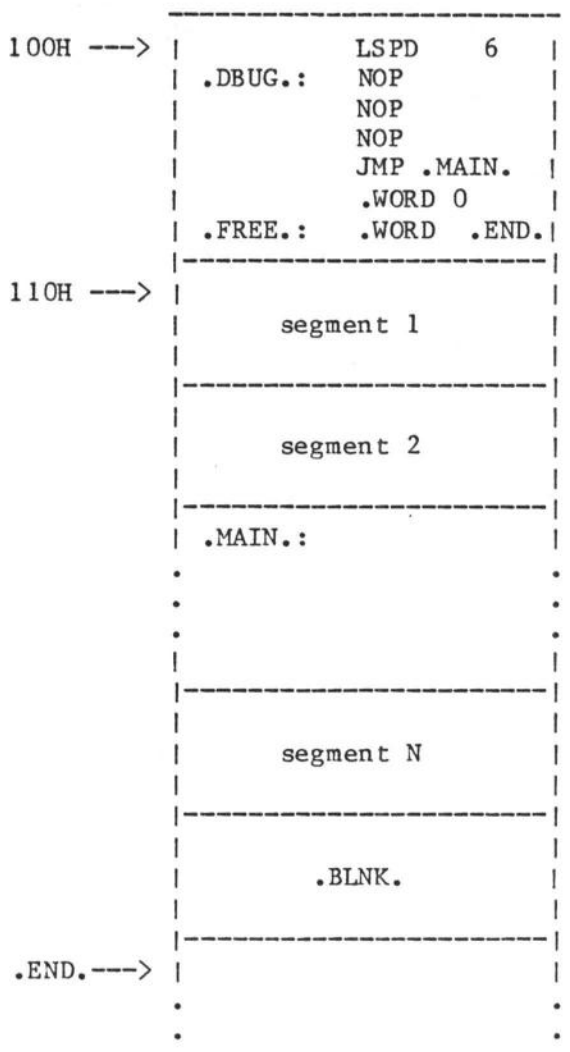


Figure 1

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. Notice that the unlabeled common .BLNK. is always located at the end of the program (or at the end of the data area if a /LOCATE .DATA. is done). Symbol .END. points to the first byte following this. Whenever FORTRAN is used, module .EMUL. (the emulator) is loaded from LIBRARYS.REL, and it will be loaded on a page (256 byte) boundry due to an efficiency trick which makes use of this fact.

## Appendix D - 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

Suppose a program has been created consisting of three modules: MOD1, MOD2, and MOD3. Each of these modules exists in separate disk files, called MOD1.REL, MOD2.REL and MOD3.REL. MOD1 is the module where execution is to begin. To create a COM file ready for execution, execute LINKER, and in response to the prompt, enter the following command:

```
PROG.COM = MOD1, MOD2, MOD3
    /MAP M A
    /MAIN MOD;
```

Two memory map reports will be obtained on the printer. When LINKER has finished, enter Q (followed by a carriage return) to terminate it.

### Example 3

Suppose that it is desired to add an I/O driver for a line printer to the system. The I/O driver is to be loaded high up in memory, so that it will not interefere with normal user programs. When executed as a normal program, the driver is to automatically load itself into the correct address and stay there until the computer is powered off. This is easily accomplished by using some of LINKER's special options.

The program will consist of two modules: the driver itself, and a loader module. Suppose that the printer is interfaced through a single port, number 90. An input from this port gives the printer status: a zero indicates that the printer is ready to accept another character, while anything else indicates that the printer is not ready (power off, out of paper, etc.). Characters are printed by writing them to port 90. The driver is used by calling it with a character to be printed in the A register. The following Z80 assembler code makes up the driver module:

```
        .IDENT  DRIVER
        .ENTRY  PRINT
        ;
PRINT:  MOV     C,A             ;SAVE CHARACTER
..WT:   IN      90H             ;WAIT FOR READY
        CPI     0
        JRNZ    ..WT
        ;
        MOV     A,C             ;RESTORE CHARACTER
        OUT     90H             ;OUTPUT IT
        RET
        .END
```

A module is produced with a single entry point, PRINT. Next, the loader module:

```
        .IDENT  LOADER
        .EXTERN LODADR
        ;
START:  LXI     D,BEGIN         ;BC:=DRIVER SIZE
        LXI     H,END
        CMP     A
        DSBC    D
        MOV     B,H
        MOV     C,L
        ;
        LXI     H,BEGIN         ;MOVE DRIVER TO
        LXI     D,LODADR        ;CORRECT ADDRESS
        LDIR
        ;
        JMP     0               ;RETURN TO
BEGIN:                          ;OPERATING SYSTEM
        ;
        .LOC    .BLNK.
END:
        .RELOC
        .END    START
```

The LOADER module computes the size of the driver and moves it to LODADR, which is an externally defined symbol. Notice that LOADER has a defined starting address of START.

Now for the LINKER input. The following LINKER command is placed into a file called PRINTER.LNK:

```
PRINTER.COM =
    LOADER,DRIVER
    /ACTUAL DRIVER = 0F800
    /DEFINE LODADR = 0F800
    /MAP
    /MAIN LOADER
```

Finally, after assembling LOADER and DRIVER, typing:

```
LINKER @PRINTER
```

causes the desired program, PRINTER.COM, to be created. LODADR is the main module, and gains control when printer is

executed, at label START. Since DRIVER follows LOADER in memory, label BEGIN points to the beginning of DRIVER. Since common .BLNK. is always loaded at the end of the program, label END points to the end of DRIVER. Thus, LOADER is able to find DRIVER and move it to LODADR, which is defined to be 0F800 hex. Although DRIVER is not actually loaded at 0F800 hex at linkage edit time, it is designed to execute at that address via the /ACTUAL option.

If at a later time it is desired to load the printer driver at a different address, this may be accomplished by simply changing LODADR and the /ACTUAL value in PRINTER.LNK, and running LINKER again. Neither of the modules would have to be re-assembled. Also, DRIVER could be made larger and more sophisticated (for example, by adding a "printer not ready" message) without having to make any changes to LOADER.

## Appendix D - 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.

### .LOC Pseudo Op

Do not use the .LOC pseudo op with a number, but only with an external symbol (see the COMMON BLOCKS discussion below) or a symbol defined relative to that module. If you want to load something at an absolute memory address, make it a separate module or common block, and use the /LOCATE option of LINKER to put it where you want it. Each line of code on your assembly listing should have a quote (') after its address. If it doesn't, you are probably doing something wrong.

### 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. Make sure that the starting symbol is defined in that module. It MAY NOT be an absolute symbol. 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. Alternatively, individual REL files may be concatenated into a single file by most file copy utility programs. One has to be careful, however, that ^Z is used as an end of file condition for /ASCII files, but physical end of file is used for /BINARY 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,

```
            .EXTERN TABLE
                    .
                    .
                    .
            .LOC    TABLE
A:          .WORD   5
B:          .BLKB   10
C:          .ASCII  "ABCDEFG"
            .RELOC
```

declares a common named TABLE consisting of A, a word, B, 10 bytes long, and C, an ASCII string. Remember that FORTRAN will name a common .BLNK. if the programmer does not give it a name.

## DATA AREA

Objects are placed into the data segment of a module by preceding them with a .LOC .DATA. The programmer may .LOC .DATA. over and over again in the program: each definition is added on to the end of the previous ones. For example:

```
            .LOC    .DATA.
FOO:        .WORD   0
BAR:        .BYTE   55H
            .RELOC
                    .
                    .
                    .
            .LOC    .DATA.
PTR:        .WORD   TABLE
TABLE:      .BLKW   100
            .RELOC
```

reserves space for four variables in the data segment. LINKER can be instructed to load the data segments and common blocks in a separate area of memory using the /LOCATE option.

## Appendix F - Date and Time

The date and time are required by LINKER when a linkable file is created. In the CP/M (*) version, the operator is asked to enter the date and time on the console, in the form MM/DD/YY HH:MM. All spaces should be filled in underneath the prompt message exactly as shown.

In the TPM version, LINKER will attempt to obtain the date and time through a special system call provided by TPM, and will request them from the operator only if they appear to be invalid. The operator can avoid this occurance by using the SET-TIME utility program before executing LINKER.

(*) CP/M is a trademark of Digital Research and TPM is a trademark of Computer Design Labs

## Appendix G - LINKER Version 1&2 Differences

This is the second version of LINKER. Many new features have been added:

- The /ASCII and /BINARY options allow more control over what kind of files may be created.

- The /NOLOAD and /XLINK options were added.

- The /ACTUAL option was expanded to provide the capability of /ACTUALing all .PROG. and .DATA. segments as a group.

- The ability to create relocatable and linkable files was added, along with the /IDENT, /PROGID, /COMMON, /EXTERN, /INTERN and /ENTRY options.

- A new disk driver module minimizes floppy diskette drive head movement, greatly increasing execution speed. Large FORTRAN programs will link up to two or three times as fast.

Several bugs were fixed:

- Empty .BLNK. or .DATA. segments caused an error #30 in the first version. This problem has been corrected.

- An error #30 was caused when an internal symbol of a module was defined relative to an external symbol in another module. This problem has been corrected as well, and chains of up to 256 symbol definitions may be created.

For the most part, version 2 is upward compatible with version 1. An exception is that the initialization code created for COM files (see Appendix C) is no longer created for HEX and REL files, it being assumed that these files are not being created for standard execution under the operating system.