SLRNK

Super-Linker

USER'S GUIDE

Copyright (c) 1984 by

SLR Systems
1622 North Main Street
Butler, PA 16001 U.S.A.
(412) 282-0864

COPYRIGHT NOTICE

This software product is distributed for the use of the original purchaser only, and no license is granted herein to copy, duplicate, sell or otherwise distribute to any other person, firm, or entity. Furthur, this software product and all forms of the program are copyrighted by **SLR Systems**, and all rights are reserved.

TRADEMARKS

Wherever referred to throughout this manual, CP/M and Z80 are registered trademarks of Digital Research and Zilog, Inc., respectively.

INTRODUCTION

SLRNK is a powerful linking loader for Z80-based CP/M systems. It takes relocatable binary information in either Microsoft or SLR Systems format from a disk file, resolves external and entry point references, and stores the output in memory for execution or outputs it to a disk file.

FEATURES

1) One pass operation

2) Supports both Microsoft and SLR Systems formats simultaneously.

3) Relocatable format allows extended math on externals and relocatable symbols

4) Up to 15 different data, program, and common areas per module

5) High speed operation

6) Optional alphabetized symbol table

7) Binary output to memory or disk (COM or HEX)

8) Supports SUBMIT-type indirect-command files

TABLE OF CONTENTS

Introduction 2

Features 2

Table of Contents 3

Running the Linker 4

Linker Operation 5

Command Line Options 8

Slash Option Summary 12

Error Message Summary 13

Appendix A

SLR Format Description 16

Appendix B

Microsoft Format Description 20

Appendix C

SLRIB - SuperLibrarian 25

Appendix D

CONFIG Utility 28

Appendix E

Warranty 30

RUNNING THE LINKER

To run the linker, type

A>**SLRNK** [COMMAND][,COMMAND]

where the brackets are not really typed, but what is enclosed in them is optional. If no COMMANDS are given on the initial command line, **SLRNK** will prompt for a command line with a percent (%) sign. **SLRNK** allows as many commands to be given as will fit on a 128 character line. Commands are separated by a comma. Note that input from the prompt is via the Read Console Buffer system call, so that commands may be passed through a SUBMIT file. However, there is a better way...

COMMANDS are defined as follows

[<Filename>][<switch>]

where <Filename> is the filename to be used in the command (optional depending on <switch>).

<switch> is zero or more command modifiers consisting of a slash (/), a letter, and optional parameters effecting the use of the <Filename> if present.

For example, to link a file named TEST.REL, and create an output file TEST.COM, the command line would be

A>**SLRNK TEST,TEST/N/E**

Before we go into detail about the COMMANDS and <switch>es, let's get an overview of the actual workings of **SLRNK**.

LINKER OPERATION

This Linker is a high-speed memory based linker. It reads relocatable files, resolves any external-global references, and produces an output file. The processes involved can be quite complex; this text is not a tutorial. The reader is assumed to be familiar with relocatability, address spaces, etc. Here we describe how **SLRNK** views these items.

SLRNK recognizes three separate address spaces, these being PROGram space, DATA space, and COMmon space. In the default case, these items are loaded one after the other for each module.

LOADING AREA

SLRNK takes up room in RAM from 100H to about 2000H. Above that area go the symbol table pointers and various buffers. From the top of available memory (TPA) down, the symbol table itself is built, along with temporary file control blocks, disk buffers, etc. Each symbol takes up 6 bytes plus the number of characters in the symbol. If you have a 52k TPA and are linking files producing 200 symbols with 8 characters each, you will have about 40K for your program.

When the first byte of code is loaded, **SLRNK** sets up a logical address space in available RAM at which to load the code. All code must be able to fit into the logical space. For instance, if you load code at location 0, the maximum address at which you can load code is around A000. If the lowest address loaded is 6000H, then you can probably load code clear up to FFFFH. Note that in either case you may ADDRESS areas (such as uninitialized data) outside this range, you just cannot load actual code there.

LOADING ORDER

If no loading directives are given, **SLRNK** links things starting at 103H. For SLR Format files, the relocatable sections are loaded as follows:

1) The program section from the module is loaded first.

2) The data section from the module is loaded next.

3) All common sections from the module not already defined are loaded next.

4) Repeat the above steps for the remaining modules.

The above defaults may be modified at any time to change the way things are loaded. You may force the program, data and common sections to all be loaded in the above manner but at a location other than 0103H with the /A option. For example:

```
/A: 5700
```

forces all of the areas to be loaded from 5700H.

The areas can be separated and loaded in particular places using the /P, /D, and /C directives to origin the program, data, and common areas in a similar manner. For example, if you are burning a bootstrap routine in FROM at location OFOOOH, and want to use RAM at 0000 for data:

```
/P:F000/D:0
```

will do that for you.

Note that in using the /A option, **SLRNK** knows that the different spaces need to be loaded one after the other. But if you use any /P, /D, or /C options to relocate a section or two, **SLRNK** will not keep track of collisions. For example, if you want to put your PROG section at 100H, and your DATA and COMMON areas at 8000H, you should use

```
/A:8000,/P:100
```

to perform that. If you instead used the following

```
/P:100,/D:8000,/C:8000
```

YOU WILL NOT GET THE DESIRED RESULTS! In this case, the DATA and COMMON areas will be load ON TOP OF EACH OTHER instead of consecutively. Be sure to think through what you really want to do.

INPUT FILES

An input file is specified by the name of the file, without the extention .REL followed by any optional "slash" directives. The extension is always .REL. Any options immediately following the file name apply to that file. For example:

```
INIT/P:100/D:8000
```

causes **SLRNK** to set the program load counter to 100H, the data load counter to 8000H and process the file INIT.REL.

After opening the input file, **SLRNK** then determines whether the file is Microsoft format or SLR Systems format.

LINKING

SLRNK performs several steps when it loads a module:

1) The format type is determined.

2) Any entry points are added to the symbol table, and an error message is output if the symbol was already defined.

3) The sizes of program, data and common areas are defined.

4) The code is loaded.

5) Any external symbols are resolved if possible, otherwise they are added to the symbol table.

6) If verbose mode has been selected, the module name, type, and location of program and data sections are displayed on the console. If the module is in SLR Format and the assembly time and date is available, that will also be displayed.

7) If an error was detected at assembly time for this module, then the message *ERR* is displayed on the console.

Let's talk in more detail about the command line options.

COMMAND LINE OPTIONS

A <Filename> given as a command will be processed as a .REL file in most cases. Any trailing <switch> operators are processed first before the file is read. If any of the <switch> options take an optional <filename> as part of their syntax, then the <Filename will be used only as part of the switch option.

SLASH OPTIONS

The operation of the linker is controlled by commands consisting of a slash '/' followed by a letter with optional parameters. In this section we describe the slash options, their syntax and function.

/A:HEXNUM

The A option is used to select a base loading address for ALL address spaces. The address spaces from each following module will be loaded one after the other starting at this address. **SLRNK** performs an implied /A:103 at startup and reset.

/C:HEXNUM

The C option is used to select a separate loading base for any common blocks that follow (not already referenced in a module). Common blocks will be loaded one after the other starting at HEXNUM, until another /C or /A is encountered.

/D:HEXNUM

The D option is used to select a separate loading base for any DATA segments that follow. This is useful in applications where PROG areas are to be stored in PROM and DATA areas are in RAM. Note that this affects only DATA or DSEG spaces. Common blocks will still be loaded according to the last /A or /C.

/E

The E option signifies end-of-link. Several things happen when this option is given. First of all, the symbol table is checked for undefined references. If any are found, then any .REQUEST libraries are searched. If there are still undefined globals, a /U option is performed, and the END operation is aborted. Otherwise the finish table is processed. Depending on the lowest address loaded, several things can occur. If the lowest address is 100H, the code is assumed to be a standard COM file. If the lowest address is 103H (the default initial load address), a jump instruction is placed at the beginning (100H) jump to the defined start address. If the lowest address is 108H, the following sequence is placed in front of the code

```
ORG 100H

LD   HL,(6)    ;CP/M TOP OF MEMORY
DEC  HL
LD   SP.HL     ;SET TOP OF STACK
JP   STARTADR  ;DECLARED WITH AN END STATEMENT
```

Any other LOWEST address is assumed to be a nonstandard CP/M file, and cannot be relocated in memory, it must be written to disk.

At this point, if a file name was given using the /N option, the file is created, and the code is written to it. Note that nonstandard /N files will have the extension CIM and will start with the lowest address loaded in your program.

/F

This option FORCES any undefined globals to a value of zero. This is useful when you get all your modules loaded, but you can't write the file because of undefined references. This should be used sparingly, but is very nice when the undefined labels are non-critical and you don't want to go back, define, assemble, and re-link.

/G

This option is useful only for CP/M applications. The sequence is essentially the same as the /E case except that instead of writing the output to disk, the code is relocated and executed.

FILENAME/H

Name the output HEX file. This option defines the name to be used at /E time. It also sets the output type to an INTEL format HEX file. For COM or CIM output, use /N instead.

FILENAME/I

This option is used to call an indirect command file. An indirect command file is merely a disk file with an EXT of SUB that contains valid linker command lines. This is very useful for development of programs that are composed of many modules. It is similar to a submit file, except that it is nestable, can be used within a submit file, and is much faster.

<FILENAME>/M

This option generates a symbol table map either on the console, or, if a filename is given, it is sent to a file FILENAME.SYM. Symbols are listed, value first, three per line. The output is optionally compatible with ZSID.

FILENAME/N

NAME the output file. This option defines the name to be used at /E time. It also sets the output type to a binary image COM or CIM type. For Hex output, use /H instead.

/O:SYMBOL:HEXNUM

ORIGIN a symbol. This option is used to force the definition of any symbols. For instance, to declare a global symbol CPM (or define one already referenced) with a value of 243Hex, you would use the command:

```
/O:CPM:243
```

/P:HEXNUM

Define PROGRAM space base pointer. This option defines the loading base for subsequent PROG or CSEG areas. Any DATA and COMMON spaces will be loaded at their previous /A,/D, or /C locations.

/Q

QUIT Linking. Aborts linker operation. Same as ^C.

/R

RESET the linker. This option resets the linker to its initial state. All options are set back to their default values, and the load area is cleared. The same as ending operation and reloading the linker.

FILENAME/S

SEARCH mode load. Directs the linker to search the specified file, and load only the module(s) that contain entry points that have already been referenced but are currently undefined.

<FILENAME>/U

List UNDEFINED symbols. This option is identical to the /M option except only undefined symbols are listed.

/V

Select VERBOSE operation. This option causes **SLRNK** to give a bit more detailed information about what it is doing. It causes input lines from indirect command files to be echoed on the console. It also causes, for each module loaded, the module name, module type (M for microsoft format and S for SLR Systems format) along with the size and location of any DATA and PROG areas loaded. If an S-Rel module contains a time and date stamp, the time and date are displayed also.

SUMMARY OF COMMAND LINE OPTIONS

/A:HEXNUM	Select common loading base for ALL types
/C:HEXNUM	Select loading base for COMMON types
/D:HEXNUM	Select loading base for DATA types
/E	END of link, generate output
/F	FORCE all undefined symbols to Zero
/G	End of link, GO execute program
FILENAME/H	Declare name of HEX output file
FILENAME/I	Call INDIRECT command file
<FILENAME>/M	Generate symbol table MAP
FILENAME/N	Declare NAME of COM output file
/O:SYMBOL:HEXNUM	ORIGIN a symbol
/P:HEXNUM	Select loading base for PROG types
/Q	QUIT. Abort. Stop already.
/R	RESET linker
FILENAME/S	SELECTIVE Search mode
<FILENAME>/U	Generate UNDEFINED symbol table
/V	Select VERBOSE operation

ERROR MESSAGE SUMMARY

Throughout the Linking process, **SLRNK** can discover several error conditions and notify you about them. This is a list of possible error messages and their meanings. For each error, **SLRNK** supplies as much information as possible about the source of the error, including any applicable filename, module name, load address and symbol name.

Abort - Disk Full!

Error encountered in writing file to disk, like no room. Remove some unnecessary files and try again.

Bad Chain

While trying to resolve a linked list (chain) of addresses referencing the given ENTRY point, an address was found that is out of range. Usually caused by loading code over top of previously loaded code, thereby contaminating the linked list. Check your load address commands. Quite often caused by having an ORG 100H in a module, then loading other relative modules with /A still at 103H.

Bad Drive

The given drive specifier is invalid.

Bad External Number

An External was referenced (S-REL) with a higher number than how many are defined. Can occur if more than 255 externals are used in one module.

Bad File

Again a corrupt file, linker is confused.

Bad Hex Digit

Something other than a valid Hex digit was found after the colon in your slash option.

Bad Type 4

A Special Item type 4 (M-REL) was encountered that is undefined. Should not occur unless M-REL file is corrupt.

Byte Out Of Range

This can occur in several places (Release 1.0 will tell at what address), depending on the instruction involved. For relative jumps involving externals, and also indexed addressing, the value generated must lie between -128 and +127. For Bit #'s involving relocatable expressions, the bit must be 0-7. For RST instructions, 0, 8, 10h, etc, are required. Finally, for IM instructions, the result must be 0, 1, or 2. Something other than these was calculated.

Can't Open File

Self-explanatory, the given filename could not be found with the .REL extension or .SUB extension, depending on command.

Common Undefined

This means that your relocatable file (M-REL) is corrupt, in that a common block was referenced before it was defined. Certain M-REL 'compatible' assemblers generate output that does that.

Duplicate Symbol

The entry point encountered (defined in this module) was previously defined in another module. Change somebody's name.

File Name Too Long

The filename given has more than 8 letters in it. CP/M filenames are limited to 8 characters preceding the extension.

File Name?

The /E command must write code to disk, but no filename has been defined with the /N or /H options.

Finish Error

Probably a linker bug. Something illegal was found in the finish table, which is built by the linker. Try to determine what may have led up to that and call us.

Line Too Long

The input line read from your indirect command file was longer than 128 characters. Each line must be less than 128 characters to prevent overflowing internal buffers.

Must Origin At 100H

The /G option is valid only when the lowest loaded location is 100h, 103h, or 108h.

OUT OF MEMORY !

In allocating space for the current module. **SLRNK** ran out of room. **SLRNK+** builds your output file 'virtually' on disk at this point so that the code won't need to be resident in RAM at all times, but with **SLRNK**, you are out of memory.

: Expected

A colon is expected after certain slash options. Either a colon is missing, or something previous confused the scanner.

Appendix A - **SLR Format**

SLR Systems Relocatable Format for object modules is described in this section. This format provides the mechanism for writing and assembling modules whose actual execution addresses are not known until "link-time". The format is byte-oriented, meaning that each item or token in the format starts and stops on a byte boundary. The format may be generated and processed one byte at a time, rather than one bit at a time.

There are two types of items available, absolute load and special items. The first byte of the item determines the item type. A number in the range of 00H through 7FH signifies an absolute load item. That correctly implies that 80H through FFH signify special items.

Absolute load item:

A number in the. range of 00H through 7FH signifies an absolute load item. If N is the first byte of the item, then the next N+1 bytes are loaded starting at the current load pointer. For example:

```
00H,CDH      : a zero means load the next byte absolute.
7FH,.....    : a 7F means load the next 128 bytes.
```

Special Item:

If the first byte of an item is 80H or bigger, then it signifies the start of a special item. Special items can be from one to actually infinity in length. This section describes the special items.

If the first byte is from 80H to 0DFH, then it is a special type 1 item.

TYPE 1

Special Type 1 items are determined by breaking the byte into two nibbles, 4 bits each. The most significant 4 bits determine the Type 1 operation, and the second nibble determines the specifics. There are sixteen data types supported by SLR format. Types are determined by the lower nibble in relevent cases. The Data types are defined as follows:

Type	Description
0	This is an absolute 16-bit word, i.e., no relocation is involved. Its value is defined in the next two bytes.
1	This is a program relative 16-bit word. Its value is determined by adding the current PROGRAM base to the next two bytes.

2 This is a data relative 16-bit word. Its value is determined by adding the current DATA base to the next two bytes.

3-E Types 3 through 14 are common relative 16-bit words. They are determined by the appropriate COMMON base to the next two bytes. With this format, up to 12 common blocks may be referenced in a single module.

F Type 15 is a special type used to reference an external value. If the next byte is not a zero, then it determines the external referenced. Up to 255 Externals may be declared using the FC option, and thereafter be referenced by number. If the defining byte is zero, then the external symbol ASCII name follows, terminated by a -1.

This table describes the Type 1 operations.

Upper Nibble Description of Action

8 Generate a 16-bit word defined by the lower nibble and the next byte(s).

9 Place the 16-bit word defined by the lower nibble and the next byte(s) onto the expression stack.

A Set the load pointer to the 16-bit value defined by the lower nibble and the next byte(s).

B Perform the operation defined by the lower nibble on data in the expression stack. See operation table.

C External + Offset. Add the given 16-bit value to the current memory location after all externals are resolved.

D External - Offset. Subtract the given 16-bit value from the current memory location after all externals have been resolved.

E E0-FF are special items. Each has its own definition, and they are defined as follows:

E0 Pop 16-bit word from expression stack. This is used to generate a 16-bit word from an expression. If the result is defined, the value is stored when the E0 is encountered. If the expression cannot yet be resolved, the stack is stored in the FINISH table, to be resolved later.

E1 Pop Byte from stack. Similar to E0, this generates an 8-bit number from the expression stack, if possible. Otherwise the stack is saved in the FINISH table. The 16-bit value popped off the stack must be in the range of -128 to +255 or an error message will be generated.

E2 Pop Relative byte from stack. Identical to E1 except that the value must be in the range of -128 to 127. This is used for JR instructions and offsets from index registers.

E3 Generate Byte from external. Similar to 8F, this takes the external number from the next byte (unless it is zero), pushes it on the stack, and performs an E1.

E4 Generate Relative byte from external. Same as a 9F code followed by an E2.

E5 Chain address. Currently not used.

E6 & E7 - Reserved for expansion.

E8 Generate BIT-type instruction. This causes **SLRNK** to pop a bit # from the stack, range-check it, shift it around and OR it with the next byte to generate the proper BIT, RES, or SET instruction.

E9 Generate an RST instruction. Used when an RST instruction contained an external reference. Restart address is on the top of the expression stack.

EA Generate an IM instruction. Valid numbers are 0, 1, & 2.

EB through F5 - Reserved for expansion

F6 Assembly Time Error. Syntax error was detected during assembly.

F7 Time and date. The next 4 bytes match the time and date structure returned by CP/M call 105 at assembly time.

F8 Chain External. Currently not used.

F9 Module Definition. Includes module name (0 to 16 characters followed by a -1), Program area size in bytes, and Data area size in bytes. This must be the first record in the relocatable file.

FA Entry Definition. Includes Entry point name (0 to 16 characters followed by a -1), and the 3-byte value defining it (first byte is type, next two are the offset). Any Entry definitions must immediately follow the Module Definition record.

FB Common Definitions. Common definitions appear after any Entry Definitions. The definition includes the common name (0 to 16 characters terminated by a -1) followed by the block length.

FC External Declaration. Declares an external reference by name (0 to 16....). Each external is assigned a number from 1 to 255 in the order in which they are declared, and they are referenced later by number.

FD Select Library. Declares a library file (0 to 16...) to be searched in case of undefined externals. Multiple libraries may be selected, they will be searched in the order selected.

FE End of module. Declares the end of the current module. The next three bytes declare an optional starting address.

FF End of file. That is it for that file.

OPERATION TABLE

B0 Add top of stack to next on stack
B1 Subtract top from next on stack
B2 Unsigned Multiply top and next
B3 Unsigned Divide top of stack into next on stack
B4 MOD function, do B3 and save remainder
B5 Bitwise AND operation
B6 Bitwise OR operation
B7 Bitwise XOR operation
B8 SHR Shift right inserting zeros
B9 SHL Shift left inserting zeros
BA Bitwise NOT operation
BB NEG (Unary minus) operation
BC Extract HIGH byte
BD Extract LOW byte
BE not used
BF Leader byte for more operations
  00 EQ
  01 NE
  02 LT
  03 LE
  04 GT
  05 GE

Appendix B - **M-REL FORMAT**

Microsoft Relocatable Format for object modules is described in this section. This format provides the mechanism for writing and assembling modules whose Actual execution addresses are not known until 'link-time'. The format is bit-oriented, meaning that byte boundaries are not observed. The format must be read and written one bit at a time, which makes it very slow to process (unless you are using **SLRNK**). Unlike the reasons given for its design, it also usually takes up more room than a comparable S-Rel file.

There are many types of items available in this format which we will divide into 'absolute byte' and 'other'. The items are better called tokens. The first bit of the token determines its type, as far as our above division is concerned. If the first bit is a 0, the token is an absolute byte, consisting of the next 8 bits. Otherwise, the token is an 'other'.

'other'

The next two bits are needed to furthur define the token. The two bits are interpreted as follows:

00	Special 'other'
01	Program Relative value. The next sixteen bits are added to the current PROG base and then loaded as two absolute bytes.
10	Data Relative value. The next sixteen bits are added to the current DATA base and then loaded as two absolute bytes.
11	Common Relative value. The next sixteen bits are added to the current COMMON base and then loaded as two absolute bytes.

Special 'other'

The Special 'other' tokens are furthur defined by reading the next 4 bits to determine the exact token referenced. The tokens may have parameters with them in the form of what we will call PARAM-A and/or PARAM-B.

Before defining the tokens, let's define the PARAM fields.

PARAM-A

PARAM-A is an 18-bit item just like defined above except that if the first two bits are 00, then it is considered an absolute item. (Note that for M80-L80 compatibility, these items are not always treated as the format defines. Those particular places will be mentioned at the appropriate token definition.)

PARAM-B

PARAM-B is a variable length item defined as follows. The next three bits determine the number of 8-bit bytes that follow. This is used for global names, etc. Note that the name could be 0-7 characters in length. Actually, if you assume no zero length labels (?) you could use the 0 value to mean a length of 8 characters. This is exactly what **Z80ASM** and SLRNK do, except in the case of common names (Fortran uses zero length for blank common name). You will see under token #4 why you are still limited to only 7 characters.

Special 'other' continued...

The 4 bits discussed above determine which of the following 16 tokens we are processing. The tokens are grouped nicely.

The token types 0 through 4 have a PARAM-B only.

0 Entry Symbol. This token is used to tell a linker that the given symbol is defined somewhere in this module- It should appear before any actual code in the file. Linkers use this during /S searches to determine whether or not to load the module.

1 Select Common Block. This selects the common block to be referenced in any subsequent 11(one-one) two bit patterns.

2 Program Name. This is the module name. For reference purposes only.

3 Request Library. This is the library name to be searched at the end of the link for any undefined externals.

4 Extended Token. This item opens up a whole list of items, mostly reverse polish expression items. The particular item is defined by the first character or two. See 'SPECIAL SPECIAL"

The token types 5 through 7 have a PARAM-A and a PARAM-B.

5 Define COMMON size. This item declares a common block name and its size. Note that the two-bit type in the PARAM-A is ignored.

6 Chain External. This item declares an external symbol name and the address in this module where it was last referenced. This is a pointer in a chain. The location pointed to also contains a pointer to another place that referenced that external. In other words, externals are referenced as a linked list. This guy just points to the head of the list. The end of the list is a zero.

7 Define Entry Point. This item declares a global symbol and its value.

The token types 8 through 14 have a PARAM-A only.

8 External - Offset. This token means subtract this PARAM-A item from the value stored at the current load pointer after all externals, etc have been resolved. Used by Microsoft for relative calls and jumps in their 8086 cross-assembler.

9 External + Offset. This token means add this PARAM-A item to the value stored at the current load pointer after all externals, etc have been resolved. Used when you say LD HL,EXTERNAL+5 ;

10 Define Data Size. Defines the size of the DATA section of the module. **SLRNK** requires this item to appear before any reference is made to a DATA load item.

11 Set Load pointer. PARAM-A is the new load location.

12 Chain address. PARAM-A is the head of the chain. Replace everybody in the linked list with the value of the current load pointer. Used by one-pass compilers for forward referenced labels. Works just like item 7, the end of the chain is a zero.

13 Define Program Size. Defines the size of the PROG section of the module. **SLRNK** requires this item to appear before any reference is made to a PROG load item. **Note** the two bit data type is ignored by **SLRNK** but must be set to a 01 to work with L80 (?).

14 End of module. This signifies the end of the current module. The PARAM-A is the start address if not zero. **Note** This token forces the input stream to a byte boundary...

The token type 15 takes no parameters.

15 End of file. Finish processing.

You guessed it, the end of module and end of file are both necessary, but the end of file appears only after the last module in the file (for library file support).

'SPECIAL SPECIAL'

The special type 4 operations are quite varied. The first character of the PARAM-B field defines another subdivision. The ones that **SLRNK** recognizes are

```
'A'     41H     ;the byte that follows defines an
                arithmetic operation (defined below).

'B'     42h     ;the bytes that follow (up to 7...)
                define an external symbol whose value is
                to be pushed on the expression stack.

'C'     43H     ;the next three bytes that follow define
                a 16-bit value to be pushed on the
                stack.  The first byte (only bits 0 & 1)
                define the two-bit type as above.
```

Other identifiers can be there, but that is all that **SLRNK** recognizes. For instance, COBOL compiler has some undocumented overlaying functions, etc.

Arithmetic Operations

The next byte after the "A" defines one of several operations. Some are Microsoft standard, some have been added by SLR Systems (extensions). The extensions are followed by an '*'.

Operator	Definition
01	Pop Byte. A byte result is popped off the expression stack.
02	Pop Word. A 16-bit result is popped off the expression stack.
03	High Byte. Place the high byte of the TOS (Top of Stack) on the stack.
04	Low Byte. Place the low byte of the TOS on the stack.
05	Bitwise NOT. Place the 1's complement of TOS on the stack

06	Negate. Place the 2's complement of TOS on the stack.
07	Subtract. Subtract top two items on stack. Place result back on stack.
08	Addition. Add the top two items, placing result on stack.
09	Multiply (unsigned).
0A	Divide (unsigned).
0B	Modulo (unsigned remainder).
10 *	Bitwise AND.
11 *	Bitwise OR.
12 *	Bitwise XOR.
13 *	Bitwise SHR.
14 *	Bitwise SHL.
15 *	BIT-Type Instruction. The next byte (9-bits) is the mask for the instruction, TOS is the bit #.
16 *	RST Instruction. TOS is the Restart address.
17 *	IM Instruction. TOS is the interrupt mode.
19 *	EQ.
1A *	NE.
1B *	LT.
1C *	LE.
ID *	GT.
1E *	GE.

APPENDIX C - **SLRIB LIBRARIAN**

SLRNK has the ability, by using the /S option, to scan a file that contains multiple REL modules, extracting only those modules which are needed, modules that contain definitions of undefined globals. If you have several often used subroutines, much disk space can be saved by combining the separate REL modules into a single library file. Also, since most of the time required to link a small separate module is the file opening and reading, it is much faster to open one library file and scan it than to open several separate files.

The real point here is not to justify the use of libraries, but to tell you how to use **SLRIB**, the librarian that helps you create and maintain SLR-Format libraries.

The **SLRIB** command line is much like **SLRNK** in that the command can be given on one line, or **SLRIB** will prompt you. To build a library called FLOATS.REL from several modules, this is a possible command sequence:

```
A>SLRIB FLOATS/N,FMULT,FDIV,FSUB,FADD,/E
```

SLRIB will read the files FMULT.REL, FDIV.REL, PSUB.REL, and FADD.REL, and combine them into one file called FLOATS.REL.

That is the simplest brute-force method of building a library. Once you have your library built, you will probably want to update it later. **SLRIB** provides convenient ways to do that also.

SLRIB Options

SLRIB Options are invoked much like **SLRNK** options; they are slash options. Here we describe the separate options.

/A ASK. This option causes **SLRIB** to prompt the console before including a particular module in an operation. For instance, if you want a new library called FLOAT consisting of the modules in FLOATS except for FDIV, you could use the line:

```
A>SLRIB FLOAT/N.FLOATS/A,/E
```

SLRIB will read FLOATS.REL and prompt you for the inclusion of each module:

```
FMULT (Y/N)?Y
FDIV (Y/N)?N
FSUB (Y/N)?Y
FADD (Y/N)?Y
```

You answer the prompt with a single letter, Y or N.

/E END. This option causes **SLRIB** to end its operation. If no library was being built, **SLRIB** just goes away. If a library was being built, it is renamed to the name previously specified with the /N command.

/I INDIRECT. Like **SLRNK**, this option causes **SLRIB** to read its commands from the preceding file name.SUB.

/L LIST. This option causes **SLRIB** to list names, and program and data area sizes of each module in the selected list.

/M MAP. This option forces an implied /L, and also lists all of the GLOBAL symbols defined and EXTERNAL symbols referenced in each module.

/N NAME. This option names the new library. It can also specify a drive on which to place the library. Note that the new library is not given this name until /E time.

%C:NEWLIB/N

/Q QUIT. This is similar to /E except that any library file in the process of being built is deleted from disk.

/R RESET. Identical to /Q, except that **SLRIB** is still running.

/U UNDEFINED. This option causes **SLRIB** to scan the given module list and report any possible BACKWARD references, items that might not be defined in one pass through the library.

MODULE LISTS

What is a module list? It is a way of selecting particular modules from a library file.

We have already seen the /A method of selecting modules from a library file. Let's be more specific about module specification.

To include all the modules in a given library file, just use the library file name:

%FLOATS

includes all the modules contained in FLOATS.REL in the module list.

If you wanted to include just the module FMULT, you could use a /A and only answer Y to FMULT (Y/N)?, but there is an easier way.

%FLOATS<FMULT>

causes just the module FMULT to be included in the module list.

To include everything but FMULT, you could use

%FLOATS<FDIV..>

which means include FDIV and everything after it.

%FLOATS<..FSUB> includes everything up to FSUB (inclusive).

%FLOATS<FADD,FDIV> includes FADD and FDIV, in that order.

These options make it very easy to update an existing library. To replace the module FSUB in FLOATS with an updated version in a file called FSUB1.

A>**SLRIB FLOATS/N,FLOATS<..FDIV>,FSUB1,FLOATS<FADD>,/E**

will accomplish just that. **SLRIB** will prompt you before it destroys the original copy of FLOATS.REL.

APPENDIX D - **CONFIG UTILITY**

The configuration utility CONFIG.COM provided with **SLRNK** allows easy customization of **SLRNK** to a particular set of requirements. This section briefly describes its use.

To run CONFIG, type

A>**CONFIG SLRNK.COM**

where SLRNK.COM is the name of the file that you want to customize.

CONFIG will respond with a series of questions requesting flag settings or item values. In parentheses is the current value of that item. You can modify the current value by typing a new one, or the item can be left the same by just entering a CR. The process can be aborted by typing a ^C. After the last question is answered, CONFIG will write out the new settings.

Now we will explain each question that CONFIG asks.

Disable Interrupts (N) -

SLRNK was written to take full advantage of the Z80 architecture and instruction set for reasons that are self-explanatory when you see it run. Certain systems that are 'Z80' CP/M machines do not provide a true Z80 environment. If your system is interrupt driven and destroys any Z80 registers on interrupts, a Y here will cause **SLRNK** to disable interrupts whenever any special Z80 registers are in use. Interrupts will always be enabled during any system calls.

Take advantage of multi-sector I/O (Y) -

System call 12 tells **SLRNK** whether multi-sector I/O is available or not, at least usually. Certain 'CP/M compatible' operating systems return a code saying they are MP/M compatible, but they don't support multi-sector I/O. If that is your case, you will need to answer N here.

Use TAB as separator bet-ween Symbols in .SYM (Y) -

SLRNK can create a symbol table on disk for use in many applications. If you need ZSID compatibility, Y causes TABs to be used as separators between Symbols, and also truncates symbols to a maximum length of 15 characters. An N causes spaces to be used between symbols, and they can be the full length. Some emulators require the spaces.

Number of bytes per line of HEX output (33) -

This option allows you to specify the number of bytes per line when generating a .HEX file. Certain emulators, PROM programmers, etc. are very particular about the format for the 'Intel Standard' HEX format.

The default extensions for various files can also be modified. They are self explanatory.

CONFIG - SLRIB

CONFIG may also be used to set options in **SLRIB**. To use it just type

A>**CONFIG SLRIB.COM**

where SLRIB.COM is the name of your librarian file.

The questions available are either self-explanatory or the same as described above.