WELCOME

to the

ARIAN OPERATING SYSTEM

We at SUPERSOFT already think of ARIAN as a "friend" and co-worker as well as an integral and indispensable part of our software development division. It is our sincerest wish that you, too, will come to share our feelings.

In the course of this manual you will discover the following:

1. How to use ARIAN ...
   ... as an assembler
   ... as a text editor
   ... as a microcomputer system executive
   ... and as a general software development tool.

2. Gain an insight into and understanding of ARIAN.

3. How to customize ARIAN for your particular applications.

We have written the manual in a readable and enjoyable form without sacrificing content.

... SUPERSOFT, 1978

ERRATA, UPDATES, AND CORRECTIONS

(A NOTE FROM THE DISTRIBUTOR) IT SEEMS LIKE EVERY TIME YOU GET A PIECE OF SOFTWARE THESE DAYS THERE ARE FAMILIAR 'ERRATA' SHEETS! WELL, LET ME TELL YOU, OUR SOFTWARE DEVELOPMENT DIVISION HAS BEEN SO BUSY THAT I CAN'T KEEP UP WITH THEM! CASE IN POINT: THE MANUAL THAT YOU ARE RECIEVING IS THE THIRD (3) RE-WRITE IN AS MANY MONTHS! NOT BECUASE OF ERRORS, BUT BECUASE OF IMPROVMENTS!

SO.... WHAT HAPPENS, I NO SOONER GET THIS MANUAL COMPLETE, BUT THE SOFTWARE GUYS BRING ME A 'NEW, GREAT, YOU GOT TO INCLUDE THIS RIGHT NOW' IMPROVEMENT. (I JUST HAD TO SIT DOWN A WHILE..) I REVIEWED THEIR NEW COMMAND ('DRVE') AND DECIDED THAT THEY WERE RIGHT; HAD TO BE ON ALL DISCS STARTING YESTERDAY, BUT I REALLY WASN'T UP TO A FULL RE-WRITE AND PRINT JOB, SO I HAVE ENCLOSED THIS 'ERRATA' SHEET. HOPE YOU DON'T MIND TOO MUCH, AND HOPE YOU LIKE THE NEW COMMAND.

A NEW COMMAND HAS BEEN ADDED TO YOUR 'ARIAN-2' SYSTEM, THE 'DRVE' COMMAND. THIS COMMAND MAKES IT POSSIBLE FOR YOU TO USE UP TO THE THREE DRIVES THAT NORTH STAR ALLOWS! 'ARIAN-2' DEFAULTS TO DRIVE #1, AS IT SHOULD, BUT ONCE YOU HAVE TURNED ON THE DISC COMMANDS (LEVEL 3) BY USE OF THE 'CMND' COMMAND, YOU CAN 'LOG' INTO WHAT EVER DRIVE YOU WANT. YOU SIMPLY TYPE 'DRVE'. 'ARIAN-2' GOES TO DISC, FINDS THE COMMAND, AND THEN PROMPTLY TELLS YOU WHAT YOUR CURRENT DRIVE LOG IN IS, AND THEN ASKS YOU FOR A NEW DRIVE LOG IN. AT THIS POINT YOU MAY EITHER 'PASS' BY HITTING <CR> OR ENTER THE NUMBER OF THE DRIVE SUCH AS 2. YOU MUST HIT ONLY ONE KEY, AND IT MUST BE A NUMBER BETWEEN 1 AND 4. 'ARIAN-2' THEN LOGS YOU INTO THAT DRIVE, AND ALL DRIVE ACCESSES ARE TO DRIVE 2, UNLESS YOU WANT TO GO BACK TO ONE.

THIS PROCESS IS VERY MUCH SIMILIAR TO THE WAY 'CP/M' WORKS, YOU LOG IN TO A DRIVE, AND CAN LOG IN TO ANY DRIVE IN YOU SYSTEM.

---> BUT; THERE IS A LITTLE TINY THING TO REMEMBER, YOU MUST HAVE THE 'DRVE.CMD' FILE ON THE DISCS WHICH ARE IN THE DRIVES YOU MAY LOG IN TO. HERES WHY: LETS SAY THAT YOU ARE IN DRIVE 1, YOU TYPE 'DRVE' AND LOG INTO DRIVE 2, TO GET BACK TO DRIVE 1, YOU MUST TYPE 'DRVE' AGAIN, HOWEVER, IF THE DISC IN DRIVE 2 DOES NOT HAVE THAT COMMAND ON IT, IT WILL JUST GIVE YOU AND ERROR, AND NO WAY BACK. HENCE, THIS IS ONE UTILITY THAT SHOULD BE ON ALL YOU 'ARIAN-2' DISCS!

THE ABILITY TO LOG INTO THE OTHER DRIVES IS INDISPENSABLE FOR INSTANT BACK UP AND FOR HAVING LOTS OF TEXT ON LINE. BY THE WAY, THE 'APND' C-SPEC WILL ASSUME THE NEXT FILE IS ON THE CURRENTLY LOGGED IN DRIVE.

A R I A N - 2 UPDATES

LEVEL THREE (3) COMMANDS

YOUR DISC HAS ON IT TWO (2) SEPERATE 'ARIAN' ASSEMBLERS. THE ONE NAMED 'ARIAN' IS THE STANDARD VERSION, AND DOES EVERY THING ADVERTISED, BUT YOU ALSO ARE RECIEVING (FREE OF CHARGE) THE BRAND NEW UPDATE 'ARIAN-2'. ARIAN-2 HAS 'LEVEL' THREE COMMANDS! THESE ARE DISC BASED COMMANDS WHICH WORK MUCH THE SAME AS THE COMMANDS CONTAINED IN RAM. THE DISC COMES WITH FIVE: MOVE.CMD, FCPY.CMD, SUBS.CMD, LNAM.CMD, AND DNAM.CMD. WITH THESE COMMANDS YOU CAN MORE EASILY MANIPULATE TEXT, SET UP SUBROUTINES TO BE APPENDED AT A LATER DATE, AND JUST GENERALLY INCREASES THE POWER OF 'ARIAN'. IT MUST BE REMEMBERED, THAT 'LEVEL' THREE IS ACCESSABLE ONLY THROUGH ARIAN-2! ALSO, ARIAN-2 USES MORE MEMORY, AND DOES NOT HAVE 'DEP' OR 'EXAM'. HENCE WE HAVE GIVEN YOU BOTH SYSTEMS. IF YOU BEGIN TO RUN OUT OF MEMORY, OR NEED TO USE 'DEP' OR 'EXAM' THEN USE 'ARIAN', IF YOU WANT TO THE FULL POWER OF 'ARIAN-2' USE IT.

HOW TO USE THE LEVEL THREE COMMANDS

TO USE LEVEL THREE YOU MUST TYPE 'CMND' WHILE IN 'ARIAN-2' , THIS IS THE DISC COMMAND SWITCH, IF YOU TYPE IT AGAIN, YOU WILL TURN THEM OFF. THE REASON FOR THIS TOGGLE IS THAT 'ARIAN-2' WILL ALWAYS GO TO DISC AND LOOK FOR A COMMAND IF IT IS NOT FOUND IN RAM, HENCE EVERY TYPING ERROR (IE: LST FOR LIST ETC) WOULD INITIATE A DISC SEARCH, NOTHING BAD WILL HAPPEN, ITS JUST TIME CONSUMMING.

ONCE YOU HAVE THE TOGGLE IN THE ON POSITION, YOU MAY ACCESS DISC BASED COMMANDS.

THE DISC COMMANDS

MOVE: THIS MOVES GROUPS OF LINES FROM ONE PLACE TO ANOTHER WITHIN THE CURRENT FILE. THERE ARE TWO FORMS: 'MOVE' AND 'MOVEB', BOTH FORM ALWAYS HAVE THREE ARGUMENTS. EXAMPLE: 'MOVE 10 30 100' MOVES LINES 10 THROUGH 30 TO AFTER 100, WHILE 'MOVEB 10 40 100' MOVE THE LINES 10 THROUGH 40 TO BEFORE LINE 100. (REMEMBER, ALWAYS THREE ARGUMENTS)

FCPY: THIS WILL COPY THE CONTENTS OF A SPECIFIED LOCAL FILE INTO THE CURRENT FILE (BOTH FILES MUST BE IN RAM. THERE ARE TWO FORMS: 'FCPY' AND 'FCPYB'. 'FCPY' <FILENAME>

<#>' COPIES THE FILE <FN> INTO THE COURRENT FILE AFTER LINE <#>, WHILE 'FCPYB <FN> <#> COPIES THE SPECIFIED FILE BEFORE LINE <#>

SUBS: (SUBSV, SUBSQ) THIS COMMAND ALLOWS THE SUBSTITUTION OF ONE WORD, CHARACTER OR PHRASE FOR ANOTHER. THE 'SUBS' SUBSTITUTES WITHOUT DISPLAYING THE RESULTS, 'SUBSV' DISPLAYS THE SUBSTITUTIONS, AND 'SUBSQ' PROMPTS THE USER ON EACH SUBSTITUTION. THERE CAN BE UP TO THREE ARGUMMENTS, IF THERE ARE NO ARGUMENTS THE SUBSTITUTION IS DONE OVER THE ENTIRE FILE, IF 1 ARGUMMENT THEN JUST THAT LINE, IF 2 ARGUMMENTS ARE PRESENT, THEN THE SUBSTITUTION TAKES PLACE OVER THAT RANGE.

DNAM, AND LNAM, WORK AS DESCRIBED IN 'ARIAN'.

FOR THOSE INTERESTED, YOU CAN HAVE CUSTOM COMMANDS ON DISC, THEY MUST FIT IN THE TRANSIENT PROGRAM REGION (3500 TO 4000 HEX), AND MUST TERMINATE WITH A 'RET' TO 'ARIAN-2'. ALL DISC COMMANDS MUST HAVE THE FORM 'XXXX.CMD' THE '.CDM' IS THE IDENTIFIER FOR 'ARIAN-2'.

ALSO, REMEMBER THAT THE WORK SPACE FOR 'ARIAN-2' CAN NOT BE LESS THAT 4000 HEX! ALSO, A FILE CREATED BY EITHER 'ARIAN' CAN BE USED BY THE OTHER, THE SYSTEMS ARE COMPLETELY COMPATIBLE.

STATEMENT OF WARRANTY

SUPERSOFT DISCLAIMS ALL WARRANTIES WITH REGARD TO THE SOFTWARE CONTAINED ON DISC OR LISTED IN MANUAL, INCLUDING ALL WARRANTIES OF MERCHANTABILITY AND FITNESS ; AND ANY STATED EXPRESS WARRANTIES ARE IN LIEU OF ALL OBLIGATIONS OR LIABILITY ON THE PART OF SUPERSOFT FOR DAMAGES, INCLUDING BUT NOT LIMITED TO SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR IN CONNECTION WITH THE USE OF PERORMANCE OF THE SOFTWARE LICENSED.

TRANSFERABILITY

SUPERSOFT SOFTWARE AND MANUALS ARE SOLD ON AN INDIVIDUAL BASIS AND NO RIGHTS FOR DUPLICATION ARE GRANTED.

TITLE AND OWNERSHIP OF THE SOFTWARE SHALL AT ALL TIMES REMAIN WITH SUPERSOFT.

IT IS AND HAS ALWAYS BEEN SUPERSOFT'S BELIEF AND INTENTION TO PROVIDE EXCELLENCE IN BOTH DESIGN AND SERVICE. IT IS TO THESE ENDS WHICH WE DEDICATE OURSELVES.

TABLE OF CONTENTS

CHAPTER 1

HOW TO USE ARIAN

Like many other operating systems, one learns how to use ARIAN by working and playing with it. This chapter is designed as an introduction to ARIAN, and, with the aid of this chapter, this manual, and ARIAN itself, the user should easily be able to learn how to write and run programs on any microcomputer system which supports ARIAN.

How to Execute ARIAN

Since ARIAN is designed to run using Northstar's standard DOS for support, execution of ARIAN is very simple. Once the user has booted in DOS, he need only type 'GO ARIAN'. ARIAN should then be loaded and executed immediately. ARIAN will print its opening message and give the user its command prompt ('>').

NOTE: you must have 8K bytes of memory starting at location 0 in order to run ARIAN.

(note: The re-entry Address is 0004 Hex)

The ARIAN Input Line Editor

One of the first things the user should know about ARIAN is how to give it a command. This is done by typing the command on the user's keyboard. Once ARIAN has given the '>' prompt, it is in command mode; ARIAN is ready for the user to type a command to it. For a listing and explanation of the commands, see Chapter 4.

In almost all cases, typing done while in ARIAN is processed by the ARIAN Input Line Editor. This editor collects each character as the user types it and allows the user to correct any typing errors he has made. As each character is typed, it is checked to see if it is a special control character. If it is, the function of the control character is executed; if it is not, the character is saved in ARIAN's input line buffer and printed on the user's terminal. When the user has finished typing his line and is satisfied that it is correct, he may then type a carriage return, which is a special control character that tells the line editor to finish inputing the line and to give the line to ARIAN to interpret and execute.

The following is a list of all the control characters recognized by the ARIAN Input Line Editor:

1. the Escape (<ESC>) key. When typed, <ESC> is printed on the user's terminal as a dollar sign ('$') followed by a carriage return (<CR>). This key tells the editor to delete the line typed in so far and start over with a new line.

2. the Line Feed (<LF>) key. This key echoes as a <CR> and does not affect the line contained in the input line buffer. The sole purpose of this function is to allow the user to continue typing his line on the next physical line of his terminal.

3. the Backspace (<BS>) or Ctrl-H key. This key allows the user to delete the last character he typed. It echoes as the cursor backing up to the previous position on the screen. For example, if the user

has typed "ABCD<BS>", only "ABC" is in the input line buffer: the "D" has been deleted. The user cannot delete beyond the beginning of his line; if he does, an <ESC> is processed, echoing as a '$' and <CR>.

4. the Delete (<DEL>) or Rubout key. This key performs the same function that <BS> does, but it echoes differently. The deleted characters are enclosed in backslashes (<DEL> is for hard-copy terminals). For instance, if the user typed "ABCD<DEL>E", this would be echoed as "ABCD\D\E", indicating that the D was deleted and the string in the input buffer is now "ABCE". If the user types more than one <DEL> in a row, all the deleted characters are enclosed in one set of backslashes. For example, if the user types "ABCDE<DEL><DEL><DEL>ABE<DEL>C", this will appear on his terminal as "ABCDE\EDC\ABE\EC", indicating that "EDC" and then "E" were deleted and the resulting string is "ABABC". This feature is provided primarily to permit the use of a device that does not support hardware backspace to be used as a principal I/O device.

5. the Carriage Return (<CR>) key. Again, the <CR> key always instructs the editor to terminate the input of the line and give the line to ARIAN to interpret.

The input line editor is used in every aspect of ARIAN except for the intra-line editing mode (see the EDIT command), and these commands are in effect whenever the user is typing something. This editor is an extremely useful tool, and with practice it wil soon become very easy and natural to use.

Files in ARIAN

ARIAN supports up to ten text files in memory (the local files) and 64 files (text, binary, or other) on disk. Whenever the user lists a file, assembles a file, edits a line, or uses any of the file modification commands, he operates on the primary file.

The primary file is one of the local files in memory. It is the last file loaded or the last file referenced by the FILE command. The FILE command (discussed later) is used to create files and make a specified local file primary. The FILE <filename> command will create a local file if a local file of the specified name does not already exist or it will make the specified local file primary. Once a file is primary, it may be edited and assembled by the user.

Example: If the specified file does not exist, FILE will create it. The output looks like:

```
>FILE TEST
TEST 3500 3500
>
```

Result: The address range specified by ARIAN shows that the file contains no data (it exists from 3500 to 3500 hexadecimal).

Example: If the specified file does exist, FILE will make it primary. The output looks like:

```
>FILE TEST2
TEST2 3570 3589
>
```

Result: Since there is a non-zero range, the user can see that the specified local file is now primary.

How to Create a Local File

The most common use of ARIAN is to write an assembly language program and execute it. In order to do this, he must know how to create a file. This is done in a number of ways.

The first and easiest way is to use the FILE command. By typing FILE <filename>, like FILE MYPROG, the user can let

ARIAN create a file for him. In response to this command, ARIAN will automatically place the file in memory, initialize the file, and respond with something like

```
MYPROG 3500 3500
```

This indicates that ARIAN has initialized the file and set it to start at location 3500 in memory. Now, to type his program into this file, the user need only use the APND command. By typing APND, the user tells ARIAN that he wants to add a block of lines to the end of the current primary file (the file he just created). ARIAN will then respond with a "?" prompt and permit the user to type the lines of his program. All text files in ARIAN must be numbered, but the APND command puts the user in block line entry mode, which automâtically numbers the line for the user (line numbers do not appear while in the mode). At this point, the user simply types the program text, and, when he has finished, types a Control-C followed by a <CR>. ARIAN will then renumber the file and place the block of lines just entered into the file.

Example: The following is an example of a short program entered by the user. From now on, underlined phrases or symbols in the examples presented indicate that these were typed by ARIAN, and the rest is typed by the user.

```
>FILE TEST
TEST 3500 3500
>APND
?  MVI A,1
?LOOP OUT OFFH : OUTPUT TO PORT FF
?  RLC
?  JMP LOOP : DO IT FOREVER
?^C
>LIST
0010  MVI A,1
0020 LOOP OUT OFFH : OUTPUT TO PORT FF
0030  RLC
0040  JMP LOOP : DO IT FOREVER
>
```

Result: The FILE command created the file TEST at location 3500 and the user then entered lines into this file using the APND command. Note that he terminated the entering of these lines by typing a Control-C (^C) followed by a <CR> (carriage returns are not shown in the above example). He then instructed ARIAN to list the file, and it did, showing the line numbers it assigned to the lines of the file. ARIAN automatically inserts a space between the line number and the first character typed in each line.

The FILE command is one method of creating a file, and the LOAD command is another. The LOAD command simply loads a file from disk and makes it primary. See the description of the LOAD command for more details.

How to Assemble and Execute a Program

Now that the program has been written, the user probably wants to assemble and execute it. This may be done by using the following commands: (1) ASSM, to assemble the file and (2) EXEC to execute the object code from the assembly. These commands have a number of forms, but the simplest forms may be generally used.

ASSM by itself assembles the primary file and places it in memory at an address selected by the memory manager (if no ORG statements were in the code). It is generally a good idea not to use ORG statements if the user is simply debugging a program since debugging time is decreased when the user doesn't have to worry about where the object code is being placed.

EXEC by itself will execute the code starting at the starting address of the last assembly, or, if an ORG was present in the program, at the start of the last ORG.

Example: The following is a sample usage of these commands.

```
>ASSM
ASM PASS 1
ASM PASS 2
6800 6807 0008
>EXEC
```

Result: The ASSM command assembled the file, and, as displayed by the hexadecimal addresses, the file exists from 6800 to 6807 and is eight bytes long. All values are in hexadecimal. The object code is then executed by the EXEC command.

How to Save and Load Programs on Disk

Saving and loading the text of programs with ARIAN is exceptionally easy because of ARIAN's dynamic file capabilities, and it is a very useful feature, especially in cases when the user is going to execute an untested program which may crash the system. He can save the text for the program by simply typing "SAVE PROG", assemble and execute the program, and, if it crashes the system, reboot ARIAN and type "LOAD PROG" to get the text of the program back. That way, if the program destroys ARIAN, he can recover easily. Also, the user may wish to save his programs when he has finished with them or he has to go away and shut down his microcomputer for some reason.

Saving, and later loading, programs is done very easily in ARIAN. In order to save a program, the user may simply type the SAVE <filename> command, like SAVE MYPROG. ARIAN will then save the primary file, regardless of what its local name is, on disk under the name specified ("MYPROG"). Later, when the user returns to the system and wishes to reload his program, he may simply type the LOAD <filename> command, like LOAD MYPROG. The specified file is made primary, and he may go on using it as he normally would.

Example: Saving and loading programs follows.

```
>SAVE IT
$ FILE SAVED
>LOAD IT
IT          683F 683F
>
```

Result: In the above example, the primary file, MYPROG, was saved on disk under the name "IT" and reloaded. Two files now exist in memory -- IT and MYPROG. Both files are exactly the same, but IT is the primary file. The addresses given above indicate the creation of another file upon the load. If a local file with the name "IT" already existed, ARIAN would prompt the user with "REPLACE?", to which he would respond with "Y" to load over the local file and "N" to abort the load.

The CUST Command and its Usage

The CUST command allows the user to add new commands to ARIAN temporarily (until he reboots ARIAN or deletes the commands). Its basic format is "CUST <commandname> <address of command>", where <commandname> contains a 4-letter command name and <address of command> is the starting address of the subroutine which executes the desired command. The command itself is a subroutine (generally). It should not have an overall affect upon the stack, and it must do a RET when done. Aside from these restrictions, the user may make the command do anything he wishes. For instance, he may create a file, assemble it, and create a customized command which executes starting at the starting address of the object code.

The other variations of the CUST command are discussed later.

CHAPTER 2

CUSTOMIZING ARIAN FOR THE INDIVIDUAL USER

There are several user-defined parameters built into ARIAN which the user may set to customize ARIAN for his particular microcomputer system. Briefly, these parameters are:

1. turning paging of the display on and off
2. setting the number of lines to display per page
3. setting the number of nulls to be output after each <CR>
4. setting the end of the user's text workspace
5. setting the address to be branched to by the EXIT command
6. customizing the disk communication utility (especially for interrupt-driven systems)

Turn paging on and off. At address 000C hexadecimal is the switch for paging. It is active zero. For CRT use, paging is usually desired; However, if a long printout is desired or if the text editor is being used as a text processor, no paging may be desired. Simply deposit (using the DEP command) a 01 hexadecimal at this address to turn off paging and a 00 hexadecimal to turn on paging.

Number of lines per page. At address 000B hexadecimal is the switch for setting the number of lines to be displayed per page. Page length can be from 1 to 255. Your disk comes with a default of 15 (0F hexadecimal), and a convenient change for a 24 line display is 17 hexadecimal. It is recommended that the user set this display for 1 less than the number of lines displayable by his CRT.

Number of nulls. At address 000F hexadecimal is the switch for setting the number of nulls output by ARIAN's output driver to the user's terminal. The value may be from 0 to 255; your disk comes with this value set at 0. Generally, a CRT should have a value of 0 and a teletype, like an ASR-33, should have 2 or 3 nulls.

End of workspace. At address 000D to 000E hexadecimal is the switch for setting the end of the text file workspace. Your disk comes with this set at 3CFF hexadecimal (assuming you have 16K bytes of memory from 0 to 3FFF). The workspace is where all text files reside, and the system sets the start of this area of memory at 3500 hexadecimal. The assembler puts the object code it generates starting at the first byte after the end of the text workspace if the user doesn't specify to the assembler where to put this code, so the user should set this boundary at 1 to 2K less than the top of his block of continuous memory from 0 (like, if he has memory from 0 to 6FFF hexadecimal, he should set this value at 67FF for a 2K assembly area). Byte 000D is the low-order of this address, usually FF hexadecimal, and byte 000E is the high-order (67 in the above example).

The EXIT branch address. At address 1DAA to 1DAB hexadecimal is the switch for setting the address to branch to when the EXIT command is given. 1DAA is the low-order part of the address and 1DBB is the high-order part. This is set to 2028 hexadecimal (the entry point of DOS) when you receive ARIAN on disk. 28 is stored at 1DAA and 20 at 1DAB.

Customizing the disk communication return point. At address 0A2B to 0A33 hexadecimal is the switch for entering a customized reset into ARIAN. This is primarily for systems using interrupt-driven I/O. For example, such a user may wish to put an EI instruction followed by a call to reset I/O at this point.

CHAPTER 3

THE ASSEMBLER IN GENERAL

The assembler translates the lines contained in the primary file into object code. The second character following the line number is the first source code character position. Therefore, the character immediately following the line number should normally be a space; the APND and INS commands place a space here automatically, and the user need only be concerned with this restriction if he enters his own lines using the <lnum> <text> command. Line numbers are not processed by the assembler; they are merely reproduced in the listing. The assembler will assemble a source program file composed of statments, comments, and pseudo operations on each line. It does this in two passes. During Pass 1, the assembler allocates all storage necessary for the translated program and defines the values of all symbols used by creating a symbol table. The storage allocated for the object code will begin at the byte explicitly or implicitly specified by the ASSM command unless an ORG pseudo-op is present in the program. During Pass 2, all expressions, symbols, and ASCII constants are evaluated and placed in allocated memory in the appropriate locations. The listing, also produced during Pass 2, indicates exactly what data is in each location of memory. Statements contain either symbolic ARIAN assembly machine instructions or pseudo-ops. The structure of such a statement is: (1) name, (2) operation, (3) operand, and (4) comment.

The name field, if present, must begin in the first assembler character position: this is the second character after the line number. The symbol in the name field can contain as many characters as the user desires, but only the first six characters are used in the symbol table to uniquely define the symbol. All symbols in this field must begin with an alphabetic character and may contain no special characters. Digits are allowed.

The operation field contains either a ARIAN assember operation mnemonic or a system pseudo-op. The ARIAN assembler operation mnemonics and system pseudo-ops are described below.

The operand field contains parameters pertaining to the operation in the operation field. If two arguments are present, they must be separated by a comma. All fields are separated and distinguished from one another by one or more spaces.

The comment field is for explanatory remarks. It is reproduced in the listing without processing. Comment lines must start with either a semicolon or an asterisk; it is recommended that comments at the end of a statement also start with one of these characters, but this is not a restriction.

Symbolic names and addressing are also supported by the assembler. To assign a symbolic name to a statement, the name is placed in the name field. To leave off the name field, the user skips two or more spaces after the line number (one or more spaces in block line entry mode) and begins the operation field. If a name is attached to a statement, the assembler assigns it the value of the current location (program) counter. The program counter holds the address of the next byte to be assembled if the instruction is a machine instruction or pseudo-op. The EQU pseudo-op, however, assigns to its label a value which is defined in the operand field. Note: do not confuse the location counter of the assembler with the "$" symbol discussed later; this location counter's value points to the next instruction to be assembled, while "$" points to the instruction after the current instruction if the current instruction is a normal mnemonic or "$" points to the current instruction if it is a pseudo-op.

Names are defined when they appear in the name, or label, field. All defined names may be used as symbolic arguments in the operand field. The reserved system symbols, however, are defined by the assembler and must not be redefined by the user; a duplicate label error will result if this is done. These reserved system symbols are discussed later.

In addition to the user-defined and the system symbols, the assembler has reserved symbols used to represent [illegible] registers of the 8080. These symbols, like the system reserved symbols, may only be used in the operand field. These symbols are:

1. A -- the accumulator; value 7

2. B -- the B register; value 0

3. C -- the C register; value 1

4. D -- the D register; value 2

5. E -- the E register; value 3

6. H -- the H register; value 4

7. L -- the L register; value 5

8. M -- memory (pointed to by H&L); value 6

9. P -- the program status word; value 6

10. PSW -- also the program status word

11. S -- the stack pointer; value 6

12. SP -- also the stack pointer

The assembler also supports relative symbolic addressing. If the name of a particular location is known, a nearby location may be specified using the known name and a numeric offset. All defined symbols, including "$", may be used in this relative symbolic addressing mode.

Example: LDA $+5

Result: This instruction loads the accumulator with the value of the byte located five bytes after the beginning of the next instruction.

Example: SSPD LOC-7

Result: This instruction stores the value of the stack pointer starting at the byte located seven bytes in front of the memory location pointed to by the symbol "LOC".

The assembler permits the user to write positive and negative numbers directly in a statement. They will be regarded as integer constants, and their binary values will be used appropriately. All unsigned numbers are considered to be positive. Decimal constants can be defined using the suffix "D" after the numeric value, but this is not required since the default is decimal. Hence, 10 and 10D define the constant ten decimal. Hexadecimal constants must start with a digit and end with a suffix "H". Examples of hexadecimal constants are 10H, 0AFH, 00010H, and 00BCH.

ASCII constants may be defined by enclosing the ASCII character within single quotes, i.e., 'C'. Two charcters may be enclosed within single quotes for double word constants.

Assembler Pseudo-ops

The following is a list and a description of the pseudo-ops recognized by the assembler:

1. ORG <operand> -- set the origin at the specified address. This instruction also resets the execution address, the assembly limits, and the location in memory at which the object code is loaded. If an ORG appears in the program anywhere but the beginning, the limits set by the last ORG are reflected in the exection address and assembly limits.

2. DS <operand> -- define storage. This reserves the specified number of bytes starting at the current location of the program counter.

3. DB <operand> -- define one byte. This instruction evaluates the specified operand and loads one 8-bit value into the location pointed to by the program counter.

4. DW <operand> -- define one word. This instruction evaluates the specifed operand, producing a 16-bit value which it loads into memory (low order, high order) at the location pointed to by the program counter.

5. ASC '<string>' -- ASCII string. This is the same as DB, but the specified string of ASCII characters is loaded into memory.

6. <label> EQU <operand> -- the specified label is assigned the computed value of the operand. The computed value is a 16-bit quantity.

7. END -- end the assembly. This statement is not absolutely required; assembly will stop when the end of the file is reached.

All pseudo-ops may be preceeded by a label.

System Reserved Labels

Another feature of ARIAN is its system reserved labels. These labels, which all start with the letter "Z" and are at most four characters long, provide the user with easy access to a host of utility subroutines for functions such as I/O, data conversion, and ARIAN entry points and buffers.

The following is a list of the system reserved labels and a description of their usages.

1. ZEOR -- ARIAN executive entry point. This is a return entry point into ARIAN; if the user wishes to do an immediate return to ARIAN, he may simply have the instruction "JMP ZEOR" in his program.

2. ZLIN -- the input line editor subroutine. This is the input line editor used by ARIAN. The user may execute this subroutine by placing a "CALL ZLIN" instruction into his program. This subroutine will immediately wait for user input from the keyboard, and, as the user types on the keyboard, it will place the characters he is typing into the input buffer. All editing control characters (<CR>, <LF>, <DEL>, <BS>) are effective and will perform their functions as though the user were actually in ARIAN. When the user types a <CR>, the subroutine finishes storing the line and does a simple return. H and L point to the first character typed, and the line in the buffer consists of the valid characters after editing followed by a <CR> character (0DH). The memory location immediately in front of H&L contains a count of the number of characters in the buffer (counting the ending 0DH) plus 1. A, H, and L are affected by this subroutine.

3. ZIBF -- this is the address of the first character of the input line buffer (same as the value passed by H&L after a call to ZLIN). It can be used in such instructions as "LXI H,ZIBF".

4. ZCC -- this is the Northstar DOS Control-C subroutine. It is used like "CALL ZCC". Upon return, the zero flag is set if a Control-C was typed by the user and not set otherwise. Only the A register is affected.

5. ZIN -- this is a system input routine. This routine waits for a character to be typed on the user's keyboard and returns the ASCII value of this character in the A register. Input is routed through the redirectable input driver by this subroutine. Only the A register is affected.

6. ZOUT -- system output routine. This routine, which routes output through the redirectable output routine if one was specified, outputs the value specified in the A register. Only this register is affected. It also does a limited amount of

special-character processing in that it outputs a <CR> as a <CR> <LF> followed by the number of null's defined by the corresponding customized parameter.

7. ZCR -- output a <CR> to the user's terminal. Used like "CALL ZCR", this routine is the same as "MVI A,0DH" followed by "CALL ZOUT". Only the A register is affected by this subroutine.

8. ZCHA -- convert hexadecimal to ASCII. This subroutine converts the low-order nybble of the A register to its corresponding ASCII hexadecimal-character equivalent. It then returns this ASCII value in the A register. For example, if A contains a binary 1 before the call to ZCHA, it contains à binary 31 hexadecimal (ASCII 1) after the call. Only the A register is affected.

9. ZCAH -- convert ASCII to hexadecimal. This routine is the reverse of ZCHA. Assuming that the value in the A register is a valid ASCII character for a hexadecimal digit (i.e., 0-9 or A-F), it converts this ASCII to its binary equivalent in the A register. If a valid character was not input to this routine, a <SP> (20 hexadecimal) is output. For example, if A contained 41 hexadecimal ('A' in ASCII) before the call, it contained 0A hexadecimal after the call. Only the A register is affected.

10. ZEN -- exchange nybbles. This subroutine exchanges the high-order and low-order nybbles of the A register. Only the A register is affected. For example, if A contains 35 hexadecimal before the call, it contains 53 hexadecimal after the call.

11. ZPA -- print the hexadecimal value in A on the user's terminal. This routine prints two hexadecimal digit characters on the user's terminal through the redirectable output driver (see SETC command). For example, if A contains 12 hexadecimal before the call, "12" (ASCII) is printed by this

subroutine. No registers are affected by this subroutine.

12. ZBLK -- print a blank (<SP>) on the user's terminal through the redirectable output driver (see SETC command). Only A is affected by this

13. ZPRH -- print the character string pointed to by H&L until a null (binary 0) is encountered. A, H, and L are affected.

14. ZPRR -- print the character string pointed to by the return address until a null is encountered. A, H, and L are affected. For example, this subroutine is used with the following instruction sequence:

```
CALL ZPRR
ASC 'THIS IS A TEST'
DB 0
```

15. ZPHL -- print H&L as four hexadecimal digits through the redirectable output driver. This is like ZPA, but four digits are printed. Only A is affected by this subroutine.

Operand Evaluation

Operand evaluation is somewhat limited in ARIAN, particularly due to the size restrictions of the system. Parenthesized expressions are not permitted. Only sixteen-bit addition and subtraction are permitted in infix (such as A+B) expressions. Single character strings of the form 'A' are permitted in expressions and unarily.

The EXAM command examines the specified block of memory. The contents of memory between the specified addresses, inclusive, are displayed on the user's terminal or redirected I/O device (see SETC command) in hexadecimal.

DEP DEP <address>

The DEP command allows the user to deposit a string of values into memory starting at the address specified. ARIAN responds to this command with a "?" prompt, and the user is to enter his values as 1 or 2 hexadecimal characters; each value is separated by one or more spaces. Typing a <CR> continues the entry on the next physical line of the I/O device. Entry of values is terminated by a Control-C and <CR>.

Example: EXAM 0 1FF

Result: The contents of memory from hexadecimal locations 0 to 1FF are displayed.

Example: DEP 34

Result: The user deposits a string of values into memory starting at location 34 hexadecimal. See the sample ARIAN session to see how the DEP command is actually used.

FILE FILE <filename>
FILE <filename> <address>

The file command allows the user to create a primary file or make a secondary local file primary. If the file specified does not already exist, it is created; otherwise, the specified file is made primary.

If an address is specified, the new primary file is located at the given address.

EXEC EXEC
EXEC <address>

The EXEC command allows the user to execute the program starting at the specified address. If no address is given, the default address, set by the last assembly (the first address printed after the assembly) is used.

CUST CUST <command name> <command address>
CUSTD <command name>
CUSTL
CUSTN <command name>
CUSTS

The CUST command controls the customized command table. CUST by itself creates the specified customized command to execute at the specified address. If a command of this name already exists, the user is prompted with "REPLACE?", to which he must respond with "N" to abort and anything else to replace the command. Remember: all commands must consist of exactly four letters.

CUSTD deletes the specified command; CUSTL lists all the customized commands currently defined; and CUSTN renames the specified customized command. In response to the CUSTN command, ARIAN prompts the user with "NEW NAME?", to which the user types the new name or just a <CR> to abort. CUSTS scratches (deletes) all entries in the customized command table.

Example: CUST PLAY F000

Result: The new customized command, PLAY, is created. Whenever PLAY is typed the subroutine located at hexadecimal F000 is executed.

Example: CUSTD PLAY

Result: PLAY is deleted.

```
RESE   RESE
```

The RESEt command resets ARIAN. Redirected I/O (see SETC command) is reset and other initializations occur.

```
ASSM   ASSM (<address> (<address>))
       ASSML (<address> (<address>))
```

The primary file is assembled by the ASSM command. ASSM just assembles, ASSML assembles and lists. If no address is specified, the program is assembled at one byte beyond the upper workspace boundary. With one address, it is assembled at the specified address and with two addresses it is assembled to execute at the first address but the object code is placed in memory starting at the second address.

Example: ASSML 0 6800

Result: The primary file is assembled to execute at 0 and the object code is placed at 6800. The assembly listing is generated. This technique is used to prevent damaging ARIAN by assembling the code on top of it.

```
SYMT   SYMT
       SYMTS
```

The SYMT command displays the user's symbol table after an assembly. SYMTS displays the system symbol table.

```
BREK   BREK <address>
       BREK
       BREKD <address>
       BREKL
```

The BREK command controls the user's breakpoints. A breakpoint in ARIAN is a one-byte instruction (RST 1) which transfers control back to ARIAN if executed. Whenever a breakpoint is encountered, the values in all the registers are preserved, allowing the user to continue program execution if he desires. Also, whenever a breakpoint is encountered, control is returned to ARIAN and the breakpoint is reset.

BREK followed by an address sets a breakpoint at the specified address; up to 8 breakpoints may be set at any one time by the user. BREK by itself clears all the breakpoints. BREKD followed by an address resets the breakpoint which resides at the specified address, and BREKL displays the addresses of all breakpoints currently set.

Breakpoints are useful program debugging tools in ARIAN. They are used primarily to determine if a program reaches a specified address, and, with the CONT command (discussed later), the user can continue dynamic testing of his programs.

<u>CONT   CONT</u>
        CONT <address>

The CONT (continue) command allows the user to proceed from a breakpoint. The values of all registers are saved when a breakpoint is encountered, and, after entering ARIAN and working in ARIAN when a breakpoint was encountered, the user may continue his program by simply typing CONT. CONT followed by an address loads the registers with the stored values and continues at the address specified; CONT by itself just restores the registers and continues at the breakpoint (the breakpoint was reset when it was executed).

<u>LIST   LIST</u>
        LIST (<line or starting line number> (<ending line number>))
        LISTF (<line or starting line number> (<ending line number>))
        LISTN (<line or starting line number> (<ending line number>))

The LIST command allows the user to list all or parts of the primary file through the redirectable I/O driver (see SETC command). If no line numbers are specified, the entire file will be listed: if one line number is specified, just that line is listed: and, if two line numbers are specified, that range of lines is listed. LIST lists the file exactly as the user typed it (with line numbers added, of course). LISTF formats the listing (assuming it is an assembly language program). To format properly, all op code must start in column 2 if there is no label and each section of the line (label, op code, operand, comment) must be separated by only one space. LISTN lists like LIST does, but line numbers and the extra space between the line number and the text are not included in the listing.

Example: LIST

Result: The entire primary file is listed.

Example: LIST 100 200

Result: Lines 100 to 200, inclusive, of the primary file are listed.

Example: LIST 100

Result: Only line 100 is listed.

Example: LISTF 300 456

Result: Lines 300 to 456, inclusive, of the primary file are listed in formatted form.

Example: LISTN 200

Result: Line 200 is listed without its line number and the space after the line number.

Note that LISTN lends itself to listing straight text, giving ARIAN the added capability of allowing ARIAN to function as a simple text process, i.e. letter writing and the like! If this is done, it is advised to turn off the paging so the page prompt will not appear on the user's I/O device.

DEL DEL <line or start line> (<end line>)

The DEL command deletes the lines specified from the primary file. The first line deleted is the first line number: if there is no line with this number, the line following this line number is deleted. At least one line number must be specified.

Example: DEL 100

Result: Line 100 is deleted from the primary file. If no line was labelled 100, and, for instance, say the line around 100 were 90, 95, 101, 105, line 101 would have been deleted.

Example: DEL 100 200

Result: Lines 100 to 200, inclusive, are deleted. If lines 101, 120, 145, 195, 199, 201, and 205 were the only lines in the file around this range, lines 101 to 199 would be deleted.

<u>RNUM RNUM</u>
RNUM (<new first line number> (<increment>))

The RNUM command renumbers the primary file. If no arguments are specified with RNUM, the file is numbered starting at 0010 and incrementing by 10. The first argument gives the number to start at and the second gives the increment..

Example: RNUM

Result: The primary file is renumbered, starting at 10 and incrementing by 10.

Example: RNUM 100

Result: The primary file is renumbered, starting at 100 and incrementing by 10.

Example: RNUM 100 5

Result: The primary file is renumbered, starting at 100 and incrementing by 5 (100, 105, 110, ...).

Warning: if wraparound occurs during renumbering, i.e., the line numbers exceed 9999, the error message "LINE NUMBER OVERFLOW" will be printed and the user must then renumber the file with a smaller increment and/or starting line number. He may destroy his file if he tries to work with an improperly-numbered file.

APND APND
APND (<line number>)

The APND command allows the user to append a block of lines to the end of his file (just APND) or insert a block of lines after a specified line (APND <line number>). While in this block line entry mode, the user need only type the text of the lines: ARIAN will place a line number and extra space on the front of each line. The user is prompted with a "?" at the beginning of each line, and he then types the line. The input line editor is in effect, and he may use it to correct typing mistakes. When finished, he simply types a Control-C immediately followed by a <CR>. If the Control-C is the first character of a new line, the previous line becomes the last line of the block to be entered; if the Control-C is the last character of a text line, the Control-C is ignored and that line without the Control-C is entered as the last line of the block. See the sample ARIAN session to view an example of entering lines through block line entry mode with APND.

When block line entry mode is exited, the entire primary file is renumbered with the default starting line number of 0010 and an increment of 10. If the "LINE NUMBER OVERFLOW" message is printed, the user must immediately use the RNUM command to renumber the file until this message does not occur. "RNUM 5 5" is recommended as the command to use (renumber starting at line 5 and incrementing by 5).

Example: APND 100

Result: The following block of lines is inserted after line 100 and before the next line of the file.

Example: APND

Result: The following block of lines is appended to the end of the file.

INS INS <line number>

The INS command is exactly the same as APND, but the block of lines is inserted in front of the specified line. This command was necessary to allow the user to insert a block of lines in front of the first line of the file. Block line entry mode and renumbering is the same in INS as it is in APND.

Example: INS 200

Result: The following block of lines typed by the user is inserted in front of line 200.

FIND FIND
FIND <starting line number of search>

The FIND command searches over the primary file for a string of characters specifed by the user and prints every line which this string occurs in. FIND by itself will search over the entire primary file and FIND <line number> will search starting at the specified line and continue to the end of the file.

In response to the FIND command, ARIAN responds with "SEARCH STRING?", to which the user may simply type a <CR> to abort the command or a string of characters followed by a <CR> to execute the search. The <CR> is not a part of the string. See the sample ARIAN session for an example of the use of the FIND command.

Example: FIND 500

Result: Search for the string specified by the user starting at line 500 and search to the end of the file.

EDIT EDIT <line number>

The EDIT command invokes the ARIAN intra-line editor. This editor allows the user to edit a line that has already been typed without retyping the entire line. If a line number is not specified, the first line of the file will be edited: if a line number is specified, that line, if it exists, or the line that would follow it if it did exist, will be edited.

The intra-line editor is a dynamic editor which permits the user to see the effects of his editing commands immediately after he types them. When a line is edited, it is copied into the editor's old line buffer and then displayed to the user. The editor then does a carriage return and prompts the user with a question mark. As the user edits this line, each character of the new line that is created is placed into the editor's new line buffer; the original line in the old line buffer is not affected. Finally, when editing is finished, the user may type a carriage return to terminate the editing process and replace the original line in the file with the line as it exists in the new line buffer.

The intra-line editor responds to a host of subcommands. The following is a complete list of these commands and their functions.

1. <sp> -- copy the character pointed to by the old line pointer into the character position pointed to by the new line pointer and advance the old line and the new line pointers by one. The space bar, therefore, will simply copy the next charcter from the old line buffer into the new line buffer. after the copy is done, the copied character will be displayed to the user.

2. E -- skip to the end of the line. the rest of the characters in the old line buffer are copied into the new line buffer and both pointers are advanced to point to the non-existent character after the last character copied. The copied characters are displayed to the user as they are copied.

3. D -- delete the character pointed to by the old line pointer (delete the next character in the old line). The character is deleted by advancing the old line pointer by one character position and not affecting the new line pointer. The deletion is displayed to the user as a backslash ("\") followed by the deleted character. If the next command typed by the user is another D, the next deleted character is displayed (without the backslash). This will continue until the user types some other command, in which case a closing backslash will be displayed. In effect, the deleted characters are enclosed in backslashes when displayed to the user.

4. I -- insert a string of characters in front of the the character currently pointed to by the old line pointer. In response to the I typed by the user, the editor types a slash ("/"). The user may then type any string of characters he wishes except for an escape or a carriage return. These characters will be copied into the new line buffer, the new line pointer will be advanced, and each character will be echoed to the user as he types it.

   The escape and carriage return characters are special characters to the insert subcommand. <ESC> instructs the insert subcommand to end the insertion. The editor then types another slash to indicate that the insertion is finished and allows the user to continue editing normally. <CR> instructs the editor to terminate creation of the new line, copy the new line into the primary file, and return to ARIAN command mode. The <CR> is echoed as a slash, a carriage return, and a system prompt (">"), indicating that the user is now in ARIAN command mode.

5. R -- replace the characters pointed to the old line pointer with the following string. Both pointers are advanced and the new characters are echoed to the user. No special character is typed to the user after he types an R, and the <ESC> and <CR> characters respond as the user types his string, each chracter he types replaces the corresponding character in the old line buffer.

6. S<letter> -- skip to the specified letter. This is the only two-character command in the editor; it consists of the letter S followed by a single character. When this command is typed, both the old and new line pointers are advanced and the corresponding characters are typed and copied into the new line buffer until the specified character is encountered or the end of the line is reached. Once the specified character is found, the old line pointer will point to it and this character will not be printed; it will be the next character in the line. The S and the specified letter are not echoed to the user when the command is typed. This command is very useful, particularly when the user [illegible]

insert, delete, or replace at a specified character; he does not have to space over to that character with this command.

7. <DEL> -- the delete key backs up the new line pointer. The characters backed over are enclosed in "<" and ">" (like they are enclosed in slashes in the I command) and deleted from the new line. Only the new line pointer is affected by this command.

8. <CR> -- terminate creation of the new line. This command terminates editing of the line and replaces the original line in the primary file with the line that currently exists in the new line buffer. If <CR> is the first editing character typed, the edit is aborted and no replacement occurs.

9. A -- abort the editing of the old line. This command may be typed whenever the editor is ready to receive a command (i.e., the editor is not in the middle of an insertion or replacement), and it terminates the edit and returns control to ARIAN without affecting the original line.

10. P -- print the new line and edit it. This command will terminate the new line at the current position of the new line pointer, copy the new line buffer into the old line buffer, print the new line, and restart the editing sequence with this new line instead of the original line. The original line as it exists in the primary file is not affected.

11. X -- exit and reedit the old line. The X command terminates the editing done so far and restarts the edit of the original line. If a P command has been previously typed, the last line placed into the old line buffer is reedited.

The editor has three error messages that it may display. These messages are:

1. ?? -- invalid command. A double question mark indicates that an invalid command has been typed.

2. ** -- end of edit line. A double asterisk indicates that the user has tried to go beyond the end of the original line illegally while editing. This error most commonly occurs while using the <SP> command to copy characters beyond the end of the line.

3. *EOL* -- end of line buffer. The length of the new line has just reached the limits of the new line buffer, and the user must reedit the original line.

Example: EDIT 200

Result: Line 200 is printed and the user is prompted with a "?". The user may now edit line 200 using the intra-line editing commands. One useful aspect of this command not yet discussed is that line 200 may be copied as line 201 or any other desired line number by editing only the line number (such as changing the second zero in 200 to a 1, and typing the E (skip to end of line) command followed by a <CR>. Line 200 in the primary file will be unchanged and line 201 will be created; line 201 will be a copy of line 200.

DDIR DDIR

The DDIR command gives a directory of the files stored on disk. The directory listing is paged, which makes it easier to read than the normal DOS listing because no entries go over the top of the page if there are more files on disk than lines on the user's CRT.

This listing contains from 4 to 5 elements per line, depending on the type of file is being listed. The name of the file is given first, followed by the starting disk address of the file in hexadecimal, a hexadecimal value for the length of the file in 256-byte blocks, the type of the file (see the DOS manual), and, if the file is binary (type 1), the execution address of the binary file. ARIAN will only work with type 0 (text) and type 1 (binary) files.

LDIR LDIR

The LDIR command gives a directory of the local text files currently residing in memory. The primary file is named first, and the secondary text files follow. The name of the file and the inclusive memory address limits of the file are given for each local text file. See the sample ARIAN session for an example.

DNAM DNAM <file name>

The DNAMe command allows the user to rename any disk file. The file name specified in the command is the name of the disk file as it currently resides on disk, and, in response to this command, ARIAN prompts the user with "NEW name?", to which he responds with a <CR> to abort the renaming process or the characters of the new name to do the actual renaming. These characters must number from 1 to 8 (8 characters maximum for a file name) and be followed immediately by a <CR>.

Example: DNAM TEXT

NEW NAME?MYFILE

Result: The disk file TEXT is renamed MYFILE.

LNAM LNAM <file name>

LNAMe is exactly like DNAM, but the specified local text file is renamed.

DDEL DDEL <file name>

DDEL deletes the specified disk file. Only the disk directory is affected: no disk file management is done by this command.

Example: DDEL MYFILE

Result: The file "MYFILE" is deleted from the disk.

LDEL LDEL <file name>

LDEL is exactly like DDEL, but it deletes the local text file specified. Also, the memory manager is invoked after the deletion and the remaining local files are packed together. The memory manager always makes sure that the primary file is physically the last file in the file workspace so the primary file can grow as the user modifies it. The memory manager also monitors the growth of the primary file while it is being modified.

LSCR LSCR

The LSCR command scratches the local file directory. All file entries are deleted, and there is no primary file after the command is executed. It effectively clears the file workspace. The files themselves, however, are not touched by this command, and they may be recovered by the RCVR command if the user knows the starting address of the file he wants to recover. See the RCVR command (following).

FCHK FCHK <file name>

The FCHK command checks the validity of the specified local file. It performs an error check on the internal structure of the specified file, and it does not do anything to alter that file. FCHK is used to check to make sure that the specified file is intact after a user error may have affected it, such as a user program running wild.

RCVR   RCVR <file name> <starting address>

The RCVR command, as mentioned under LSCR, recovers the specified file after it has been deleted. This command starts at the address specified, does an internal format check on each line of the file, and looks for an end-of-file mark. If the file checks out as valid, it will make a directory entry under the specified name and make the recovered file primary.

Example: RCVR LOSTFILE 3500

Result: A recovery is attempted on the data starting at 3500 hexadecimal, and, if the data forms a valid file, the file LOSTFILE is created in the local file directory and made primary. The user may now edit this file like any other local file.

SAVE   SAVE <file name>
SAVEB <file name> <start address> <end address>

The SAVE command saves a file on disk. SAVE <file name> saves the primary file on disk under the specified name. If another file already exists with the specified name, the user will be prompted with "REPLACE?", to which he responds with "N" to abort or any other character to do the replacement. The disk file manager is invoked by this command, and this is all the user need do to save the primary file on disk. Note that the response to "REPLACE?" is only one character and a <CR> is not necessary.

SAVEB saves the specified section of memory on disk under the specified file name. Again, the disk file manager is invoked and the "REPLACE?" option may be given if a file already exists with the specified name. See the sample ARIAN session for examples of the SAVE command.

LOAD   LOAD <file name>

The LOAD command loads the specified file from disk into memory. If the file is a text file (type 0), it will be loaded into memory at a location chosen by the memory manager and it will be made into the primary file. This primary file will have the same name as the corresponding disk file..

If the file is a binary file (type 1), it will be loaded into memory at its execution address only. The user must then know what the file's execution address is in order to execute it. This can be discovered by using the DDIR command and reading this address from the directory entry for the loaded binary file.

```
SETC   SETC
       SETCI <address>
       SETCO <address>
```

The SETC command controls redirectable I/O in ARIAN. SETC by itself resets I/O to the I/O routines in Northstar's DOS. SETCI tells ARIAN that all further input is handled by the subroutine starting at the specified address; SETCO tells ARIAN that all further output is handled by the subroutine starting at the specified address. I/O is set or reset immediately after the <CR> is typed on the approriate SETC command.

I/O is always reset to the DOS I/O routines upon initial entry into ARIAN and by the RESEt command. The entry point at location 4 hexadecimal also resets I/O.

The redirectable I/O drivers written by the user (the routines addressed by the SETCI and SETCO commands) must conform to the following rules:

1. No register may be altered by these routines. It is recommended that the user PUSH all registers, including the A register, onto the stack at the beginning of these routines and POP them at the end.

2. These routines must do a simple (or conditional) RET when they are finished.

3. <CR> must be handled as just outputting ODH

(carriage return in ASCII). The output driver interface in ARIAN always checks for a <CR> and will always send out <CR> <LF> and the specified number of nulls in response to a <CR>. The ARIAN output driver outputs all characters it receives exactly as it receives them -- except for <CR>. <CR> is always output as <CR> <LF> followed by the required number of nulls. Hence, "A" (41 hexadecimal) is output as "A" (41 hexadecimal); <CR> (0D hexadecimal) is output as <CR> <LF> (0A hexadecimal) and the required number of nulls (0 hexadecimal).

WORK WORK

WORK <start address> <end address>

The WORK command allows the user to set and display the boundaries of the text file workspace in ARIAN. This workspace is where the memory manager places and plays with all the local text files. The WORK command by itself just displays the boundaries currently set. WORK followed by the two addresses reset these boundaries. The starting address should never be less than 3500 hexadecimal, and the ending address, as a general rule, should be 1 or 2K less than the top of contiguous memory. The restriction on the ending address is made because this area -- to the top of memory -- is used by the assembler to place the object code generated when the user does not explicitly tell the assembler where to place this code (the simple ASSM command).

Example: WORK 3500 67FF

Result: The workspace is set to 3500 to 67FF hexadecimal. In this example the user has memory from 3500 to 6FFF, and he left 2K for the assembler to place the object code in.

EXIT EXIT

The EXIT command simply branches to Northstar's DOS (location 2028 hexadecimal). The user may reset this branch address if he desires (see the page on customizing ARIAN).

CHAPTER 5

A SAMPLE ARIAN TERMINAL SESSION

The following is a reproduction of an actual terminal session with ARIAN. This reproduction was created using the SETCO command with a redirectable I/O driver.

```
>*
>*
>*  THIS IS A SAMPLE ARIAN TERMINAL SESSION
>*    IT IS BEING RECORDED BY A CYBER-175 COMPUTING SYSTEM
>*    THROUGH REDIRECTABLE I/O
>*
>* THIS IS AN EXAMPLE OF EXAMINE AND DEPOSIT
>EXAM 0  10
0000:00  00 00 03  00 04 00 00  08 C3 3F 19  0F 00 FF 67
0010:00
>* THIS IS THE USER-DEFINED PARAMETER REGION
>* LET'S CHANGE THE NUMBER OF LINES TO DISPLAY PER PAGE
>DEP 0B
?16^C
>* WE HAVE CHANGED NUMBER OF LINES PER PAGE TO
>  16 HEXADECIMAL, OR 22 DECIMAL
>DEP 0D
?00 80 ^C
>* WE HAVE CHANGED THE END OF THE TEXT FILE AREA TO BE AT
>  8000 HEXADECIMAL (NOTE LOW ORDER, HIGH ORDER); THE TEXT
>  FILES NOW RESIDE IN THE 3500 TO 8000 REGION OF MEMORY
>  AND THE AUTOMATIC ADDRESS FOR ASSEMBLIES IS 8001
>
>* NOW, LET'S PUT IT ALL BACK THE WAY IT WAS
>DEP 0B
```

```
?0F
?00
?FF 67 0
?^C
>* WE HAVE WRITTEN THE ORIGINAL PARAMETERS BACK INTO <ARIAN,
>  DOING A CONTINUOUS DEPOSIT STARTING AT LOCATION 0B HEXADECIMAL.
>  NOTE THAT ALL HEXADECIMAL ADDRESSES GIVEN IN THE COMMANDS MUST
>  START WITH A DIGIT; IF WE SAID "DEP B" INSTEAD OF "DEP 0B",
>  AN ERROR MESSAGE WOULD BE GIVEN.
>FILE TEST
TEST      3500  3500
>*  WE HAVE JUST CREATED A FILE NAMED "TEST"
>APND
? CALL ZCR : OUTPUT <CR> <LF>
? MVI C,10
? MVI A,30 : PRINT CHARS '0' TO '9'
?LOOP CALL ZOUT : PRINT VALUE IN A
? INR A : INCR A
? DCR C
? JNZ LOOP
? RET^C
>LIST
0010  CALL ZCR : OUTPUT <CR> <LF>
0020  MVI C,10
0030  MVI A,30 : PRINT CHARS '0' TO '9'
0040 LOOP CALL ZOUT : PRINT VALUE IN A
0050  INR A : INCR A
0060  DCR C
0070  JNZ LOOP
0080  RET
>*  NOW FOR A FORMATTED LIST
>LISTF
0010               CALL    ZCR : OUTPUT <CR> <LF>
0020               MVI     C,10
0030               MVI     A,30 : PRINT CHARS '0' TO '9'
0040     LOOP      CALL    ZOUT : PRINT VALUE IN A
0050               INR     A ; INCR A
0060               DCR     C
0070               JNZ     LOOP
0080               RET
>ASSM
ASM PASS 1
ASM PASS 2
6800 680F 0010
>EXEC

 !"#$%&'
>*  WHOOPS!  SHOULD HAVE BEEN 30H, NOT 30 (DECIMAL)
>LIST 10 30
0010  CALL ZCR : OUTPUT <CR> <LF>
0020  MVI C,10
0030  MVI A,30 : PRINT CHARS '0' TO '9'
```

```
>EDIT 30
 0030  MVI A,30 : PRINT CHARS '0' TO '9'
?0030  MVI A,30/H/ : PRINT CHARS '0' TO '9'
>LISTF 30
0030            MVI     A,30H : PRINT CHARS '0' TO '9'
>ASSM
ASM PASS 1
ASM PASS 2
6800 680F 0010
>EXEC

0123456789
>*  NOW, LET'S SPACE OUT THE DIGITS
>LISTF 40 70
0040     LOOP   CALL    ZOUT : PRINT VALUE IN A
0050            INR     A ; INCR A
0060            DCR     C
0070            JNZ     LOOP
>INS 50
? PUSH PSW
? CALL ZBLK : PRINT <SP> BETWEEN EACH DIGIT
? POP PSW^C
>LISTF
0010            CALL    ZCR : OUTPUT <CR> <LF>
0020            MVI     C,10
0030            MVI     A,30H : PRINT CHARS '0' TO '9'
0040     LOOP   CALL    ZOUT : PRINT VALUE IN A
0050            PUSH    PSW
0060            CALL    ZBLK : PRINT <SP> BETWEEN EACH DIGIT
0070            POP     PSW
0080            INR     A ; INCR A
0090            DCR     C
0100            JNZ     LOOP
0110            RET
>ASSM
ASM PASS 1
ASM PASS 2
6800 6814 0015
>EXEC

0 1 2 3 4 5 6 7 8 9
>*  LOOKS GOOD!
>*  NOW, LET'S RUN THIS AS A CUSTOMIZED COMMAND
>CUST PLAY 6800
>CUSTL
PLAY 6800
>PLAY
0 1 2 3 4 5 6 7 8 9
>*  LET'S DO THAT AGAIN
>PLAY
0 1 2 3 4 5 6 7 8 9
>*  NOW FOR ANOTHER FILE
```

```
>FILE TEST2
TEST2     3614  3614
>LDIR
TEST2     3614  3614
TEST      3500  3613
>*  NOTE THE MEMORY MANAGER
>LIST
>APND
? CALL ZCR : <CR>
? CALL ZPER : PRINT THE FOLLOWING STRING
? ASC 'HELLO THERE!'
? DB 0
? RET^C
>ASSM $
WORK
3500 67FF
>ASSM.6C00
ASM PASS 1
ASM PASS 2
6C00 6C13 0014
>CUST MESS 6C00
>CUSTL
PLAY 6800
MESS 6C00
>MESS
HELLO THERE!
>LDIR
TEST2     3614  368A
TEST      3500  3613
>FILE TEST
TEST      3577  368A
>* AGAIN, NOTE MEMORY MANAGER
>LIST$
ASSM 6C80
ASM PASS 1
ASM PASS 2
6C80 6C94 0015
>PLAY
0 1 2 3 4 5 6 7 8 9
>MESS
HELLO THERE!
>LDIR
TEST      3577  368A
TEST2     3500  3576
>* NOW FOR A DISK DIRECTORY
>DDIR
FDOS      0004 000A 1 2000
ARIAN     000E 0020 1 0000
CUTERS    002E 004C 0
SYSLOG    007A 0003 0
SYMSOFT   007D 0005 0
FORMA [illegible]
```

```
VDMDISP   0087 0005 0
VDMDRVR   0096 0005 0
SYSLOGV   009B 0007 0
TEXT      00A2 0004 0
ARIANS    00A6 0020 1 0000
REDIRI    00C6 0001 0
REDIRO    00C7 0001 0
>LDIR
TEST      3577  368A
TEST2     3500  3576
>FCHK
$ VALID FILE
>FCHK TEST2
$ VALID FILE
>*  SAVE PRIMARY FILE ON DISK
>SAVE T1
$ FILE SAVED
>FILE TEST2
TEST2     3614  368A
>LDIR
TEST2     3614  368A
TEST      3500  3613
>FILE
TEST2     3614  368A
>SAVE T2
$ FILE SAVED
>DDIR
FDOS      0004 000A 1 2000
ARIAN     000E 0020 1 0000
CUTERS    002E 004C 0
SYSLOG    007A 0003 0
SYMSORT   007D 0005 0
FORMAT    0082 0005 0
VDMDISP   0087 0005 0
VDMDRVR   0096 0005 0
SYSLOGV   009B 0007 0
TEXT      00A2 0004 0
ARIANS    00A6 0020 1 0000
REDIRI    00C6 0001 0
REDIRO    00C7 0001 0
T1        00E1 0002 0 MORE?
T2        00E3 0001 0
>LSCR
>LDIR
>*   NOTE:  DELETED LOCAL FILES
>RCVR FUN1 3500
FUN1      3500  3500
>LDIR
FUN1      3500  3613
>FILE
FUN1      3500  3613
>*   NOTE:  WE RECOVERED A DELETED FILE
```

```
>FIND
SEARCH STRING? HELLO
>FIND
SEARCH STRING? MVI
0020  MVI C,10
0030  MVI A,30H : PRINT CHARS '0' TO '9'
>LISTF
0010              CALL    ZCR : OUTPUT <CR> <LF>
0020              MVI     C,10
0030              MVI     A,30H : PRINT CHARS '0' TO '9'
0040     LOOP     CALL    ZOUT : PRINT VALUE IN A
0050              PUSH    PSW
0060              CALL    ZBLK : PRINT <SP> BETWEEN EACH DIGIT
0070              POP     PSW
0080              INR     A : INCR A
0090              DCR     C
0100              JNZ     LOOP
0110              RET
>FIND
SEARCH STRING? <SP> BETWEEN EACH
0060  CALL ZBLK : PRINT <SP> BETWEEN EACH DIGIT
>FIND 30
SEARCH STRING? MVI
0030  MVI A,30H : PRINT CHARS '0' TO '9'
>FIND
SEARCH STRING? MVI
0020  MVI C,10
0030  MVI A,30H : PRINT CHARS '0' TO '9'
>LSCR
>*  NOW FOR A LOAD
>LCAD T1
T1       3500  3500
>LSTF
$ INVLD CMND
>LISTF
0010              CALL    ZCR : OUTPUT <CR> <LF>
0020              MVI     C,10
0030              MVI     A,30H : PRINT CHARS '0' TO '9'
0040     LOOP     CALL    ZOUT : PRINT VALUE IN A
0050              PUSH    PSW
0060              CALL    ZBLK : PRINT <SP> BETWEEN EACH DIGIT
0070              POP     PSW
0080              INR     A : INCR A
0090              DCR     C
0100              JNZ     LOOP
0110              RET
>LCAD T2
T2       3614  3614
>LDIR
T2       3614  368A
T1       [illegible]  [illegible]
>LISTF
```

```
0010            CALL    ZCR ; <CR>
0020            CALL    ZPRR ; PRINT THE FOLLOWING STRING
0030            ASC     'HELLO THERE!'
0040            DB      0
0050            RET
>FIND
SEARCH STRING? HELLO
0030  ASC 'HELLO THERE!'
>LOAD T3
T3       368B  368B
$ NO SUCH FILE
>*  WHOOPS!  WE DON'T HAVE FILE T3 ON DISK, DO WE?
>LDIR
T3       368B  368B
T2       3614  368A
T1       3500  3613
>LDEL T3
>LDIR
T2       3614  368A
T1       3500  3613
>*  MAY AS WELL DELETE T2 AND T1 FROM DISK
>DDEL T1
>DDEL T2
>DDIR
FDOS     0004 000A 1 2000
ARIAN    000E 0020 1 0000
CUTERS   002E 004C 0
SYSLOG   007A 0003 0
SYMSORT  007D 0005 0
FORMAT   0082 0005 0
VDMDISP  0087 0005 0
VDMDRVR  0096 0005 0
SYSLOGV  009B 0007 0
TEXT     00A2 0004 0
ARIANS   00A6 0020 1 0000
REDIRI   00C6 0001 0
REDIRO   00C7 0001 0
>LDIR
T2       3614  368A
T1       3500  3613
>*  NOTE: T1 AND T2 ARE STILL LOCAL
>
>*  NOW FOR SOME TEXT PROCESSING
>LSCR
>LDIR
>FILE TEXT1
TEXT1    3500  3500
>APND
?THIS IS LINE 1
?THIS IS LINE 2
?THIS IS LINE 3
?THIS MAY BE LINE 4 ETC
```

```
>LIST
0010 THIS IS LINE 1
0020 THIS IS LINE 2
0030 THIS IS LINE 3
0040 THIS MAY BE LINE 4
>LISTN
THIS IS LINE 1
THIS IS LINE 2
THIS IS LINE 3
THIS MAY BE LINE 4
>LISTF
0010     THIS     IS       LINE 1
0020     THIS     IS       LINE 2
0030     THIS     IS       LINE 3
0040     THIS     MAY      BE LINE 4
>* YUCK!
>EDIT 20
 0020 THIS IS LINE 2
?0020 \THIS\/MY LINE/ IS LINE 2
 0020 MY LINE IS LINE 2
?0020 \MY\/THIS/ LINE IS/ NOT/ LINE 2/2/
>LIST 20
0020 THIS LINE IS NOT LINE 22
>EDIT 20
 0020 THIS LINE IS NOT LINE 22
?0020 THIS \LINE \IS \NOT \LINE \2\2
>LIST 20
0020 THIS IS LINE 2
>LISTN
THIS IS LINE 1
THIS IS LINE 2
THIS IS LINE 3
THIS MAY BE LINE 4
>LIST
0010 THIS IS LINE 1
0020 THIS IS LINE 2
0030 THIS IS LINE 3
0040 THIS MAY BE LINE 4
>RNUM 100
>LIST
0100 THIS IS LINE 1
0110 THIS IS LINE 2
0120 THIS IS LINE 3
0130 THIS MAY BE LINE 4
>RNUM 100 100
>LIST
0100 THIS IS LINE 1
0200 THIS IS LINE 2
0300 THIS IS LINE 3
0400 THIS MAY BE LINE 4
>150 THIS IS LINE 150
>LIST
```

```
0100 THIS IS LINE 1
0150 THIS IS LINE 150
0200 THIS IS LINE 2
0300 THIS IS LINE 3
0400 THIS MAY BE LINE 4
>325 THIS IS LINE 325
>324 THIS IS LINE 324
>326 THIS IS LINE 326
>LIST
0100 THIS IS LINE 1
0150 THIS IS LINE 150
0200 THIS IS LINE 2
0300 THIS IS LINE 3
0324 THIS IS LINE 324
0325 THIS IS LINE 325
0326 THIS IS LINE 326
0400 THIS MAY BE LINE 4
>RNUM 100
>LIST
0100 THIS IS LINE 1
0110 THIS IS LINE 150
0120 THIS IS LINE 2
0130 THIS IS LINE 3
0140 THIS IS LINE 324
0150 THIS IS LINE 325
0160 THIS IS LINE 326
0170 THIS MAY BE LINE 4
>RNUM 1000
>LIST
1000 THIS IS LINE 1
1010 THIS IS LINE 150
1020 THIS IS LINE 2
1030 THIS IS LINE 3
1040 THIS IS LINE 324
1050 THIS IS LINE 325
1060 THIS IS LINE 326
1070 THIS MAY BE LINE 4
>DEL 1070
>LIST
1000 THIS IS LINE 1
1010 THIS IS LINE 150
1020 THIS IS LINE 2
1030 THIS IS LINE 3
1040 THIS IS LINE 324
1050 THIS IS LINE 325
1060 THIS IS LINE 326
>DEL 1010 1030
>LST
$ INVLD CMND
>LIST
1000 THIS IS LINE 1
1040 THIS IS LINE 324
```

```
1050 THIS IS LINE 325
1060 THIS IS LINE 326
>RNUM
>LIST
0010 THIS IS LINE 1
0020 THIS IS LINE 324
0030 THIS IS LINE 325
0040 THIS IS LINE 326
>*  WELL, THAT'S IT FOR NOW
>*    SO LONG, FOLKS
>*    SO LONG, FOLKS
```

All numeric arugments are assumed to be decimal unless the suffix "H" is appended to them. Therefore, 100 is 100 decimal and 100H is hexadecimal 100.

The "$" symbol is used as the value of the program counter. In normal instructions, "$" points to the first byte of the next instruction; in pseudo-ops, "$" points to the first byte of the pseudo-op. This permits relative addressing to take the form of "LXI H,LABEL-$" and pseudo-ops

like "STACK EQU $" to be used.

Finally, if an expression with a value greater than OFF hexadecimal is loaded into an eight-bit register, like "MVI A,1FFH", only the low-order byte of this value is loaded.

Examples of permitted expressions include:

```
LABEL+3
POINT-'A'+60
POINT3-0AFH+6-2
HERE-$-2
```

Assembler Error Messages

The following is a list of the error messages produced by the assembler and their meanings:

1. R -- register error. The register name is missing or invalid.

2. S -- syntax error. The instruction syntax is incorrect.

3. U -- undefined symbol. The referenced symbol is undefined.

4. V -- value error. The computed value cannot be represented as a 16-bit integer or the expression has a syntax error.

5. M -- missing label error. A required label is missing.

6. A -- argument error. The instruction's argument is of the wrong type or generally incorrect.

7. L -- label error. The label of this instruction contains an invalid character.

8. D -- duplicate label error. The label of this instruction has been defined elsewhere.

9. O -- opcode error. The opcode in this instruction is invalid.

CHAPTER 4

THE ARIAN COMMANDS

This chapter of the users' manual describes all of the ARIAN commands in detail and how to use them. The following is a list of these commands (parentheses mean the enclosed item is optional):

1. EXAM <address> <address>
2. DEP <address>
3. FILE <filename>
4. EXEC (<address>)
5. CUST <command name> <address of command>
6. RESE
7. ASSM
8. SYMT
9. BREK <address>
10. CONT
11. LIST (<line or starting line number>) (<ending line number>)
12. DEL (<line or starting line number>) (<ending line number>)
13. RNUM (<starting line number>) (<increment>)

14. APND (<line to append after>)
15. INS (<line to insert in front of>)
16. FIND (<line to start search at>)
17. EDIT <line to edit>
18. DDIR
19. LDIR
20. DNAM <filename>
21. LNAM <filename>
22. DDEL <filename>
23. LDEL <filename>
24. LSCR
25. FCHK (<filename>)
26. RCVR <filename> <starting address of file>
27. SAVE <filename>
28. LOAD <filename>
29. SETC (<address>)
30. WORK <starting address> <ending address>
31. EXIT

The ARIAN Commands in Detail

The most basic of the ARIAN commands is <lnum> <text>. This command, consisting of a line number, a space, and some text, enters that line into the primary file at the correct place. Following are the rest of the commands in detail.

EXAM EXAM <address> <address>