The diskette for **HACKER'S KIT #1** contains the following:

directory of . 11:28:10

Owner	Last modified	attributes	sector	bytecount	name
0	84/01/31 1126	d-ewrewr	A	120	CMDS

directory of CMDS 11:28:21

Owner	Last modified	attributes	sector	bytecount	name
0	84/01/31 1126	--e-rewr	13	A26	disinp
0	84/01/31 1126	--e-rewr	1F	A7	memlist
0	84/01/31 1126	--e-rewr	21	9B	memload
0	84/01/31 1126	--e-rewr	23	97	resmem
0	84/01/31 1126	--e-rewr	25	2C2	split
0	84/01/31 1126	--e-rewr	29	AA	filter
0	84/01/31 1126	--e-rewr	2B	DA	rewrite

The CMDS directory contains only executable programs, refer to the documention with this package for their use. You will probably want to copy these to your working CMDS directory on you system disk.

HACKER'S KIT #1

The utilities included in this package will be of use to those programmers who have at least a fair knowledge of assembly language and the ways of the OS-9 operating system. Haphazard use of some of them (mainly memload) can cause you to crash your system. (If you write over active program code with memload).

To make full use of this package you need to understand the use of pipes and I/O redirection as you will almost always need to use one or the other with these utilites, for example; the memlist utility writes a portion of memory to the standard output with no formatting. Unless you now that the area of memory you are listing contains ascii text you cannot just let it go to the screen. If the area contains binary data you could "pipe" the data into the DUMP command to get a hex dump or (>) redirect it into a file for some other use.

Similarly the DISINP disassembler command disassembles data from the standard input. If you use it without I/O redirection it will be trying to disassemble the what ever the binary values of the keys you type on the keyboard are (likely not what you had in mind). By having its source as the standard input however you gain versatility by using redirection.

Example, two ways to disassemble assembly code in the file "code" would be:

```
disinp <code      and

list code ! disinp
```

both would produce the same result. If the code you want to disassemble is in memory all you need is a way to move the data from memory to the input of DisInp. (HINT: see memlist).

DISINP

SYNTAX: DISINP [hex_offset]

Disassembles bytes from the standard input to standard output, after skipping hex_offset bytes from standard input (default hex offset = 0). If more than $EF0 bytes are to be skipped or disassembled then use shell to expand the data memory beyond #4K.

Examples: LIST /D0/CMDS/DIR ! DISINP AB

(list command and pipe send the binary contents of dir into disinp, disassembly starts at an offset of AB bytes from the beginning).

MEMLIST E015 45 ! DISINP >/P

The above example uses memlist to list 45 hex bytes starting at absolute memory location E015 hex, these bytes are piped into DISINP which disassebles them with the disassembly listing sent to the printer (>/p).

NOTE: When trying to disassemble an executable OS-9 assembly language module, remember that the executable code does not start at the beginning of the module. The first bytes contain the module header. At an offset of 9 from the beginning of the module header is the offset to the execution address of the module, (normally the first instruction code bytes). Therefore to disassemble a program module first use dump or debug to find the hex execution offset (two bytes starting at byte 9 in the dump), then use this offset with disinp to start the disassembly at the actual execution address of the module.

The memory locations shown in the disassembly listing will begin at some even page boundary dependent on where the data memory is that is used by disinp plus any disassembly offset given with the command.

FILTER

SYNTAX: FILTER [-opts] hex_character_value

FILTER copies the standard input to the standard output removing all occurances of the given character value.

Options have form -A or -An where n is a decimal number associated with that option. Options are single letters A through Z (upper or lower case). Multiple options can be written as -A -B-C or -ABC .

OPTIONS: -An If the A options is specified "n" bytes after each occurance of hex_character_value are romoved from the stream also.

Example: filter 0A <text file >cleanfile

Filters linefeed characters ($0A) from text_file to new file cleanfile.

EXAMPLE: OS9: filter -a1 de <docfile >docfile.f

When ever the hex character $DE is encountered in docfile, it and the one byte after it are skipped, everything else is copied to docfile.f (I use this to remove printer control codes from text I edit with Dynastar).

MEMLIST

SYNTAX: MEMLIST beginning_address [bytecnt]

Memory is listed in unformated binary beginning at beginning_ address for bytecnt bytes to the standard output. If bytecnt is ommitted 100 hex is assumed. Both numbers are given in hex.

EXAMPLES: memlist 0 ! dump

Gives you a hex dump of the first $100 bytes of ram in system (system direct page).

memlist 40a7 3FC >tempfile

Copies $3FC bytes of memory beginning at $40A7 to tempfile.

MEMLOAD

SYNTAX: MEMLOAD starting_hex_address

Memload reads the standard input into memory beginning at the absolute hex starting address given until eof is encountered on the input or until the system crashes, which ever comes first!!!

It is your responsibilty to determine that the memory area you are loading into is free memory, the safe way is to enter debug, use resmem to reserve the required size area of memory, then execute MEMLOAD from within debug. Usually memload will have its standard input redirected to a file.

EXAMPLES: memload 1000 </d0/os9boot

The entire contents of the os9boot file is loaded into memory starting at location $1000. (NOTE: the file loaded does not need to contain program modules with valid CRC's as required by the LOAD command).

RESMEM

SYNTAX: RESMEM hex_memory_size

RESMEM is meant to be executed under DEBUG to reserve an area of memory for special use i.e. usually for using MEMLOAD.

To use under debug:

```
OS9:load resmem debug
OS9:debug
D: lresmem
D: eresmem 800      (example showing reserving 800 hex bytes of ram)
D: G                  (debug will stop with error #13 after the g
 Error #013           command is given and display a register
                      dump... the U register contains the address of
                      the beginning of the reserved memory area)

                      To load a file into this memory invoke memload
                      while still in debug by using the "$" to
                      invoke a shell command line.

D: $memload A00 </d0/junkfile

                      The above assumes that "A00" was the beginning
                      of the memory area to load junkfile into as
                      given by the U-register in the previous
                      operation, at this point debug commands could
                      be used to patch the file in memory and then
                      it could be saved to disk again with memlist
                      as follows:

D: $memlist a00 14b >/d0/newjunk
D: q
OS9:

                      The above will save 14B hex bytes starting at
                      memory location A00 hex to the file newjunk,
                      if newjunk is suppossed to be an executable
                      program the verify command will need to be
                      used to update the module crc and then
                      attr to set the file execute permission bit
                      before it could be called as a command.
```


REWRITE

SYNTAX: REWRITE pathname [hex offset]

Writes the standard input to the pathname in update mode beginning at the optional hex offset into the pathname (usually a disk file). The default offset is zero. REWRITE writes over exsisting data in the file given by pathname.

Examples:

Rewrite /d0/os9boot 7FC </d0/patch

The above writes the contents of patch into the os9boot file beginning at an offset of 7FC from the beginning of the os9boot file. (NOTE: This will likely make your disk unbootable unless you are really sure what you are doing, remember when tampering with an executalbe module in a file to use the VERIFY command with the Update switch to copy that file to a new one with a corrected CRC.

SPLIT

SYNTAX: SPLIT <input_path [-opts] [[cnt path] [cnt path]...] >output_path

Data read thru the standard input path is split into one or more output paths. The first "cnt" bytes or lines of input data are written to the first path given, the next "cnt" bytes or lines to the next path, etc., any remaining data is then written to the standard output path. The counts are hex bytes by default but can optionally be lines.

Options have form -A or -An where n is a decimal number associated with that option. Options are single letters A through Z (upper or lower case). Multiple options can be written as -A -B-C or -ABC .

Options:

- Bn Use a buffer size of nK bytes. (Default is approximately 3K).
- D Counts on command line are in decimal (default is hex).
- L Counts are by line instead of the default by bytes, and data is read and written a line at a time.

Examples:

SPLIT <BIG_TEXT -LD 500 TEXT1 350 TEXT2 >TEXT3

Creates 3 new files, TEXT1 contains the first 500 lines of BIG_TEXT, TEXT2 contains the next 350 lines from BIG_TEXT, and TEXT3 contains any remaining lines.

SPLIT

Copies the standard input to the standard output

SPLIT <BINFILE -B32 AC2 BINFRONT >JUNK

Copies the first AC2 (hexidecimal) bytes of BINFILE to BINFRONT and the remaining bytes into JUNK. A 32K buffer is used by the split program, in this case the same effect as the -b32 switch could have been obtained by #32K on the command line, if the input to split was a pipeline from another program the -b32 would have been the only way to supply more memory to split.