1020

Bell Laboratories **Cover Sheet for Technical Memorandum**

The information contained herein is for the use of employees of Bell Laboratories and is not for publication (see GEI 13.9-3)

Title- Structured File Scanner

Date- January 4, 1974

TM- 74-1271-2

Other Keywords- Debugging
Structured Files
File Maintenance

Author(s)	Location and Room	Extension
L. A. Dimino	MH 2C-514	2390

Charging Case- 39199

Filing Case- 39199-11

ABSTRACT

The UNIX command SFS (Structured File Scanner) provides a versatile debugging facility for those who desire to use complex data structures on random access mass storage.

Some features of SFS:

1. It provides both interactive and preprogrammed operation.
2. It provides expression evaluation and branching.
3. It provides the ability to assimilate a rather arbitrary set of hierarchical structure definitions.
4. It provides the ability to locate, to dump, and to patch specific instances of structure in the file. Furthermore, in the dump and patch operations the external form of the structure is selected by the user.
5. It provides the ability to escape to the UNIX command level in order to allow the use of other UNIX debugging aids.

Pages Text 22 Other 7 Total 29

No. Figures ____ No. Tables ____ No. Refs. ____

Address Label

J. Morangano

MH 2C-512

DISTRIBUTION
(REFER GEI 13.9-3)

COMPLETE MEMORANDUM TO

CORRESPONDENCE FILES -HO

OFFICIAL FILE COPY
(FORM E-7770) - PLUS
ONE WHITE COPY FOR
EACH ADDITIONAL
FILING CASE
REFERENCED

DATE FILE COPY
(FORM E-1328)

13 REFERENCE COPIES

127 SUP
CCFDDB

AHO,A V
ALEXANDER,J L JR
ALMQUIST,R P
ARNOLD,S L
BARTLETT,WADE S
BECKETT,J T
BERRYMAN,R D
BIREN,MRS EMMA E
BLUE,J L
BOLSKY,MORRIS I
BOWYER,L RAY
BOYCE,W M
BOYLE,GERALD C
BROWN,R STANLEY
BROWN,WILLIAM R
CEMASHKO,FRED
CHEN,STEPHEN
CHERRY,MS I L
CLIFFORD,ROBERT M
COOPER,DENNIS W
CORASICK,MISS M J
CRUME,LARRY L
DAVIS,R L JR
DENTON,R T
DEVLIN,MRS SUSAN J
DI MARSICO,BRIAN J
DIETZEL,R ROBERT
DIMINO,L A
ESTVANDER,ROBERT A
FLEISCHER,HERBERT I
FOUGHT,B T
FRASER,A G
FREEMAN,R GLENN
GEARY,M J
GEYLING,F T
GOLDSTEIN,A JAY
GRAHAM,R L
GUNDERMAN,R
HALL,ANDREW D JR
HAMILTON,MISS P A
HAMMING,R W
HANNAY,N B
HESS,MILTON S
IVIE,EVAN L
JOHNSON,STEPHEN C
KEESE,W M
KERNIGHAN,BRIAN W
MAHLER,G R

COMPLETE MEMORANDUM TO

MC CRACKEN,MARYLOU
MC GILL,ROBERT
MC GONEGAL,MISS C A
MC ILROY,M DOUGLAS
*MCDONALD,H S
MORGAN,S P
NIMTZ,R O
NOGLANDI,J
NOSAL,PAUL E
OPITZ,R
OSSANNA,J F JR
PATTON,GARDNER C
PINSON,ELLIOTT N
*PRIM,ROBERT C
ROBERTS,CHARLES S
ROCHAN,MRS PATRICIA A
RODRIGUEZ,ERNESTO J
ROSLER,LAWRENCE
SATZ,L F
SCHLEGEL,C T
SEMMELMAN,C L
SHORTER,J W
SIPES,J C
SMITH,K H
SMITH,LESLIE T
SPANG,THOMAS C
SPIRES,R J
STRCHECKER,CARY A
TANIS,MRS PATRICIA J
TERRY,M E
THOMPSON,K
THOMSON,K-L
TRACY,MRS C E
*TUKEY,JOHN W
VAN HAUSEN,J DAVID
WALFORD,ROBERT B
WALKER,MISS E A
WARNER,JACK L
WASSERMAN,MRS Z
WATSON,C S
WAX,MRS RUTH
WEBBER,SUSAN A
WEXELBLAT,RICHARD L
WITKOWSKI,MISS JOAN D
WOLFE,ROBERT M
YAMIN,MRS E E
94 NAMES

COVER SHEET ONLY TO

CORRESPONDENCE FILES -HO

5 COPIES
PLUS ONE COPY FOR
EACH ADDITIONAL
FILING SUBJECT

127
12 DIR
13 DIR
CCFDEC

COVER SHEET ONLY TO

ABRAHAM,STUART A
AHRENS,RAINER B
ALCALAY,DAVID
AMITAY,NOACH
AMOSS,JOHN J
ANSELL,H G
ARMBRUSTER,MISS M E
ARMSTRONG,DOUGLAS B
ARNDT,DENNIS L
ARNOLD,GEORGE W
ARNOLD,THOMAS F
BACURA,DENNIS C
BAUER,MISS H A
BAUGH,C R
BEYER,JEAN-DAVID
BILOWOS,RICHARD M
BIRCHALL,R H
*BITTRICH,MRS M E
BLINN,JAMES C
BLY,JOSEPH A
BODEN,F J
BOHACHEVSKY,I O
BRADFORD,C E
BRANDT,RICHARD B
BRISSON,R J
BUCHSBAUM,S J
CAMPBELL,STEPHEN T
CANADAY,RUDD H
CARAWAY,R E
CARUSO,A F
CASPERS,MRS BARBARA E
CASTELLANO,MRS M A
CAVINESS,JOHN D
CHAMBERS,J M
CHRIST,C W JR
CLAYTON,DANIEL P
COBEN,ROBERT M
COLE,LOUIS M
CONNERS,R R
COOPER,A E
COPP,DAVID H
COULTER,J REGINALD
CUTLER,C CHAPIN
D ANDREA,MRS LOUISE A
DESMOND,J P
DEUTSCH,DAVID N
DIMMICK,JAMES O
DOLOTTA,T A
DOMBROWSKI,F J
DRAKE,MRS L
DRYDEN,JOHN J
DUFFY,FRANCIS P
EDELSON,D
EDMUNDS,T W
EILBOTT,MRS JOAN
ELLIOTT,R J
ELY,T C
ERDLE,K W
ESSERMAN,ALAN R
FABISCH,MICHAEL P
FAULKNER,R A
FELDMAN,STUART I
FELS,ALLEN M
FIGLIUZZI,MISS M E
FISCHER,H B
FORT,JAMES W
FOWLER,BRUCE R

COVER SHEET ONLY TO

FOX,PHYLLIS
FOY,J C
FRANK,MISS A J
FREEMAN,R DON
FROST,H BONNELL
FULTON,ALAN W
GABBE,JOHN D
GARCIA,R F
GATES,G W
GATES,ROBERT K
GAY,FRANCIS A
GELBER,MISS CHERON L
GIBB,KENNETH R
GILBERT,MRS HINDA S
GILLETTE,DEAN
GIMPEL,JAMES F
GITHENS,JOHN A
GLUCK,F
GNANADESIKAN,R
GOLABEK,MISS R
GOODRICH,LEWIS M
GORDON,BRIAN G
GORMAN,JAMES E
GOTTDENKER,ROBERT G
GRACE,KENNETH JR
GRAFTON,V
GROSS,ARTHUR G
GUERRIERO,JOSEPH R
GUMMEL,HERMANN K
HALE,A L
HALL,MILTON S JR
HALL,W G
HANSEN,R J
HARASYMIW,J
HARKNESS,MRS CAROL J
HARRISON,NEAL T
HARRIS,S D
HARTMANN,ROBERT H
HARUTA,K
HAUSE,A D
HAWKINS,DONALD T
HAWKINS,RICHARD B
HEATH,SIDNEY F III
HELD,RICHARD W
HEMMETER,RICHARD W
HEMMING,C JR
HEROLD,JOHN W
HINDERKS,L W
HOCHBAUM,MISS FRANCES
HOEHN,MISS MARIE J
HONIG,W L
HOOVER,MRS E S
HOYT,WILLIAM F
HO,MISS J
HUDSON,E T
HUNNICUTT,CHARLES F
HYMAN,B
IPPOLITI,O D
IRVINE,M M
JACKOWSKI,D J
JACOBS,H S
JAKIELSKI,C E
JAMES,DENNIS B
JENSEN,P D
JORDAN,MRS E
JOYNT,L
KACHURAK,JOSEPH J

COVER SHEET ONLY TO

KAISER,J F
KALRO,ASHOK L
KANE,J RICHARD
KANE,MRS ANNE B
KAYEL,R G
KEARNEY,ROBERT
KENNEDY,ROBERT A
KERTZ,DENIS R
KILLMER,JOHN C JR
KIRSCHNER,I B
KLAPPROTH,F F
KLEINER,RICHARD T
KNOWLTON,KENNETH
KRUSKAL,JOSEPH B
LEE,MISS B N
LEGENHAUSEN,S
LESK,MICHAEL E
LESK,MS ANN B
LIEBERT,THOMAS A
LINDERMAN,J
LIN,SHEN
LOMUTO,N
LUDERER,GOTTFRIED
LUMMIS,ROBERT C
LUTZ,KENNETH J
LYCKLAMA,HEINZ
LYONS,STEVEN H
MAC WILLIAMS,
MADDEN,MRS D M
MALAVESE,E P
MALCHESKI,W J JR
MALCOLM,J A
MALTHANER,W A
MARSH,BRENT L
MASHEY,JOHN R
MATHEWS,MAX V
MATTAVI,RONALD A
MC CABE,PETER S
MC EOWEN,JAMES R
MC MULLEN,E C
MC TIGUE,G E
MEERES,MRS EILEEN M
MENIST,DAVID B
MENNINGER,R E
MENON,P R
MERCADANTE,P F
METAXIDES,A
METZ,ROBERT F
MILLER,ALAN H
MILLER,G C
MILLER,GERALD L
MILLER,S E
MILLS,MISS ARLINE D
MITCHELL,OLGA M M
MOLINELLI,JOHN J
MOLTA,J W
MORGAN,DENNIS J
MORRIS,ROBERT
MURATORI,RICHARD D
MUSA,J D
NEHRLICH,W R
NELSON,N-P
NEWBURGER,J A
NINKE,WILLIAM H
NORTON,HERBERT O
NOWITZ,D A
O BRIEN,J A

* NAMED BY AUTHOR

> CITED AS REFERENCE SOURCE

...
318 TOTAL

DIMINO,L A
MH 2C519

TM-78-1271
TOTAL PAGES

TO GET A COMPLETE COPY:

1. BE SURE YOUR CORRECT ADDRESS IS GIVEN ON THE OTHER SIDE.
2. FOLD THIS SHEET IN HALF WITH THIS SIDE OUT AND STAPLE.
3. CIRCLE THE ADDRESS AT RIGHT. USE NO ENVELOPE.

PLEASE SEND A COMPLETE COPY TO THE ADDRESS SHOWN ON THE OTHER SIDE.
NO ENVELOPE WILL BE NEEDED IF YOU SIMPLY STAPLE THIS COVER SHEET TO THE COMPLETE COPY.
IF COPIES ARE NO LONGER AVAILABLE PLEASE FORWARD THIS REQUEST TO THE CORRESPONDENCE FILES.

Bell Laboratories

subject: Structured File Scanner - Case 39199-11

date: January 4, 1974

from: L. A. Dimino

TM-74-1271-2

MEMORANDUM FOR FILE

INTRODUCTION. The debugging facilities currently available to those who desire to use complex data structures on random access mass storage most often either lack the generality to handle more than a small set of predefined structures, or provide that generality at the expense of forcing the user to deal almost exclusively with octal or hexadecimal data representations. The lack of an adequate debugging facility for dealing with structured files can obviously be taken as having an inhibitory effect on the use of structured files, and a negative influence on programmer productivity when such files are used.

The author has provided a UNIX command SFS (Structured File Scanner) to fill this gap in the debugging facilities available with that system. The program is written in the C language which currently is available on the PDP-11 running under UNIX and the HIS-6070 running under GCOS. However, the program could be translated with a relatively small effort into a number of other procedural languages which provide both recursive procedure entry, and pointer and character valued variables.

Some features of SFS include.

1. It provides interactive operation, but in addition it also provides for a preprogrammed mode of operation.
2. It provides for elaborate address computations by supplying arithmetic expression evaluation. Furthermore, the values of addresses and of expressions have sufficient range to address all standard mass storage devices used with the hardware.
3. It provides the ability to assimilate a rather arbitrary set of hierarchical structure definitions. In addition, a convenient mechanism is provided for naming instances of substructure in the defined structure(s) in the file.
4. It provides the ability to locate, to dump, and to patch specific instances of structure in the file. The user input and output required by these operations are in a form natural to the structure under consideration rather than some single form arbitrarily chosen for all structures.
5. It provides the ability to escape to the UNIX command level in order to allow the use of other UNIX debugging aids.

In the following description a formal syntactic description of the SFS debugging language is interspersed with an English description. The English description is essentially complete unto itself, and the reader unfamiliar with the metalanguage of the formal description may ignore that description, and work out any ambiguities in the English description to his own satisfaction through interaction with SFS.

The formal syntax is essentially BNF with the metacharacters ::= and newline designating definition and alternation respectively. For those unfamiliar with BNF the following offers a simple but detailed description of the metalanguage.

Each piece of text enclosed in angles <> is a symbol (nonterminal symbol) in the metalanguage, and defines a set of character strings. The text enclosed between the angles is intended to express the meaning attached to the strings in the set.

The nonterminal symbol to the left of the defining symbol (::=) is taken to be defined by the text to the right. This text takes the form of a sequence of strings of symbols of the metalanguage separated by newlines. Each line of the definition provides an alternate form for a string in the defined set. The character strings defined by a line of the definition are just those formed by selecting a string from each set named on the line and concatenating them together. For example,

```
<abc>  ::=  <a><b>
            <c>
```

defines the set <abc> to consist of those character strings formed either by selecting a string from set <a>, say "a", and another from set <b>, say "bb" and concatenating them together to form a single string, "abb", or alternately by a single string selected from <c>.

The metalanguage has some additional restrictions imposed on the definitions. No nonterminal symbol may appear on the left side of more than one definition; every nonterminal symbol must appear on the left side of some definition. The first restriction insures that the definitions are unique, and the second insures that each nonterminal symbol of a finite length text in the metalanguage is ultimately definied in terms of a class of primitive sets of strings (terminal symbols).

In addition, there must be only one nonterminal symbol which does not appear on the right side of some definition. This insures that the definitions ultimately form a definition of a single set of strings.

In the following description the terminal symbols of the metalanguage include:

1. the unitary sets containing the strings consisting of a single printing ascii character, these are represented by the character they contain (Ascii space is not considered a printing character in this context.);
2. <newline>, which contains the single string consisting of an ascii newline character;
3. <spaces>, which consists of those strings formed from one or more ascii space characters;
4. <sn*>, which contains all the strings which can be formed from the ascii characters space and newline;
5. <octal digit>, which contains the strings "0" "1" ... "7";
6. <decimal digit>, which contains the strings "0" "1" ... "9";
7. <null>, which contains the null or empty string;
8. <name>, which contains all non null strings of length less than 64, which begin with an alphabetic character, and which only contain alphabetic characters and decimal digits;
9. <format name>, which contains only those strings in <name> which refer to structures previously defined in SFS;
10. <comment>, which contains all those strings which can be formed from the ascii character set excluding <newline>, and which have a length less than 150 characters (this includes the null string.);
11. <file name>, which consists of those strings which form UNIX filenames;
12. <UNIX command>, which consists of those strings which form UNIX commands;
13. <ascii list>, which contains arbitrary strings of ascii characters which conform in length to a previously defined, and currently referenced, ascii structure according to the rules given in the section on ascii input below.

It should be noted that <ascii list> necessarily depends upon the structure reference with which it is associated, if the formal syntax is to define the grammar of the language unambigously. Similarly, to define an unambiguous grammar the symbols <octal list>, <decimal list>, <floating octal list>, <floating decimal list>, and <patch values> must all depend on the structure reference with which they are associated. Consequently, they also should be terminal symbols. Nevertheless, definitions of these symbols are included in the syntax to elucidate their rules of formation.

SFS COMMAND.

sfs filename [-]

provides an interactive program for scanning and patching structured files. The file is assumed to reside on a random access device (i.e., a device for which the SEEK system call is appropriate). The second (optional) argument indicates that the file is on a block (512 byte) addressed device. If this argument is omitted, the file is assumed to be byte addressed (i.e., an ordinary UNIX file). In either event, the file will always be read and written in units of 512 bytes, aligned at the byte addresses 0,512,1024,..., with only one block residing in core at any one time.

The input to SFS consists of a sequence of commands, each of which contains addressing arguments followed by a one or two character command identifier and the particular arguments required for the execution of that command.

<command> ::= <addressing><command_id and arguments><newline>

Each command is entered on a separate line. Moreover, the processing of a command may be terminated abruptly by typing an interrupt (rubout/delete character).

The addressing consists of zero, one, or two arithmetic expressions separated by a comma.

<addressing> ::= <null>
<address>
<address>,<address>

<address> ::= .
<constant>
(<expression>)

The two expression address specifies a starting and stopping address. While the one expression address specifies a starting address only, the same value is used for the stopping address when it is required. The null addressing is equivalent to taking the current address (.) for both addresses.

Note that if either of the expressions in the addressing is more complicated than a single integer value or the value of the current address, it should be parenthesised.

Although most of the commands require a zero or one expression address, one is allowed to to supply the additional irrelevent information. Furthermore, although several of the commands may read or write the file, all will restore the block containing the current address to core upon commmand termination.

Some examples of the addressing forms include:

.	The current address.
7492	A decimal byte address.
07740	An octal byte address.
(block.head),(block.head + block.length *2)	A range defined by the structure of the file.

COMMANDS. This section defines the nonterminal symbol <command_id and arguments>. However, for convenience in exposition, rather than give the formal syntax and the English explaination separately, they are presented together in a tabular form.

The commands include the following:

<null>	A null command prints the value of (.), the current address.
q	Quit. That is, exit from SFS.
"<comment>	This causes the succeeding characters (<comment>) up to the next newline to be ignored. This command is also useful as a no-op, when used in conjunction with the jump command below.
r	Set (.) to the starting address, and reinitialize the core buffer. This command offers the only exception to the addressing interpretation given above. The null address is interpreted as though it was the expression ((.+511)&037777777000). That is, the address is taken to be the first byte in the next block.
w	Write the core buffer to the block containing the starting address.

SFS has a 32 bit current address register, eight global 32 bit registers (named $0 $1 ... $7), and eight local 32 bit registers (named '0 '1 ... '7). The global registers are always available to the user, like FORTRAN COMMON variables, but a new generation of local registers is provided at each subroutine entry, like ordinary ALGOL variables.

The next three commands provide a means to manipulate these registers through assignment, and the fourth command provides expression evaluation without assignment.

=<expression>	Set (.) to the value of the arithmetic expression, and initialize the core buffer. This is equivalent to '(expression)r'.

$<octal digit>=<expression>
Assign the value of the arithmetic expression to the named global register.

'<octal digit>=<expression>
Assign the value of the arithmetic expression to the named local register.

:<expression> Evaluate the arithmetic expression and print the result.

The next two commands allow the user to return to the UNIX command level of control. However, the special facilities offered by the shell (UNIX command interpreter) are not provided so that if they are required, the shell (UNIX command sh) must be called explicitly.

!<UNIX command> Escape to UNIX, and wait for the UNIX command to complete execution.

&<UNIX command> Initiate a UNIX process, and resume activity immediately.

The next command provides a subroutine capabitity for SFS. This allows complex or repetitive interactions to be programmed, and stored for future reference on a file. When the subroutine is entered the local registers are initialised in sequence by the starting address, the stopping address, and the values of the expressions which follow the filename. If more than one expression is supplied, they must be separated by commas.

%<file name><spaces><list of expressions>
The command stream is to be taken from the named file. Every command may be executed from this file, including another %<file name> command. Recursive subroutine entry is permitted.

<list of expressions> ::= <null>
<expression>
<expression>,<list of expressions>

The next command permits the user variation in the sequencing of commands when executing in the programmed mode. This jump command is considered an error if issued interactively.

j<jump condition><expression>
This command compares the values of the starting and stopping addresses according to the specified jump condition, and performs the jump (transfer of control) only if that condition is satisfied. If the condition is omitted, the jump is always executed.

The jump conditions include =, !=, >, >=, <, <=, and ->, which stand for equals, not equals, greater than, greater than or equals, less than, less than or equals, and bitwise logical implication. Bitwise logical implication may also be interpreted as bit containment. (i.e., a->b is true if the ones in the binary representation of a are also ones in the representation of b.)

If the jump is to be executed, the expression is evaluated. Its value gives the offset, positive or negative, in lines from the current command line to the next one to be executed.

<jump condition> ::= <null>
=
!=
>
>=
<
<=
->

The next five commands are related to format (structure) definitions. The form of the definitions is described in the section FORMAT DEFINITION below.

f<format> Define a format (structure).

u<format name> Undefine a format. The ability to undefine and then redefine a parameterised format, makes it feasible to work with many structures which have a variable configuration.

l List the names of the formats which are presently defined.

l<format name> Print the named format definition.

l-<format name> Print the length of core storage alloted to the format definition, and the length of the structure defined by the format.

The o (offset) command is a convenient method for determining those offsets into a structure which are useful for address computations. The form of a format reference is defined in the section FORMAT REFERENCES below.

o<format ref> Print the length of the substructure referenced, the offset in the structure to the first item referenced, the offset in the structure to the last item referenced and the

address of the first item referenced relative to the current address.

o+<format ref> Prints the offsets like the o command above, but it also sets the value of the current address (.) to the address of the first item referenced.

The d, p, and s (dump, patch and search) commands which follow form the core of the debugging facility.

d<format ref> Dump the referenced structure based at the starting address. The data is dumped in a format determined by the format definition referenced.

p<format ref><patch values>
Patch the referenced structure based at the starting address. The list of patch values must agree in length and form (ascii, octal, decimal, floating point) with the format reference. Within the patch values a single newline character serves as a signal for the program to cue the user as to the type and form of the next value required, and two successive newline characters (empty line) signal the program to leave the value in question unchanged. Note that interrupting the p command does not restore the partially patched structure to its original form.

<patch values> ::= <list of values><sn*><patch values>

<list of values> ::= <ascii list>
<octal list>
<decimal list>
<floating octal list>
<floating decimal list>

s<format ref><conditions>
Test the structure based at the starting address to determine whether the value defined by the format reference satisfies the conjunction of the conditions specified. If the conditions are all satisfied, the base address of the structure is printed, otherwise an error is signaled. The command also sets the value of global register zero. This is set to zero if the test (search) is successful and to minus one if it fails.

The value of the format reference is restricted to an array of bytes, words, or single or double precision numbers. These are interpreted respectively as a byte string of unsigned

values, a multi-length twos-complement integer and a floating point number with a multi-length mantissa.

The conditions which can be tested include equals, not equals, less than, less than or equals, greater than, greater than or equals and bit inclusion (bitwise logical implication).

The list of conditions (see the section CONDITIONS below) must begin with either a newline character or a (?). If the newline is chosen, the program will cue the user as to the length and form of the value to be tested. Note that each value specified in a condition must agree in length and form with the format reference.

Within the value specified in a condition two successive newline characters (empty line) are interpreted by the program as a request to repeat the last byte, word or floating point value input.

The command normally ends with the newline following the end of a condition. However, if the user types a backslash (\) before that newline, the program will cue him with an asterisk (*). He may then continue entering conditions or initiate the test by typing a newline character.

s-<format ref><conditions>

Search the array of structure based at the starting address and extending to the stopping address. (See the s command above.) The base of each of the instances of the structure which satisfy the conditions is printed.

s+<format ref><conditions>

Search the array of structure based at the starting address and extending to the stopping address. (See the s command above.) The base of the first instance of the structure which satisfies the conditions is printed, and the current address (.) is set to this value.

The syntax for the arguments to the search command is given below in the CONDITIONS section, this section also contains two examples of the use of the command.

The next commands permit the user to turn on and off the printing of labels in the patch and dump operation, and the prompting in

the search operations. The inhibition of this printing is frequently desirable when executing programmed input.

These commands also inhibit the printing produced by the s+ and o+ commands when executing a program. They have no effect on these commands when execution is interactive.

v — Turn on the printing of labels, and prompting. This is the default condition.

v+ — This is identical in action to the v command above.

v- — Turn off the printing of labels, and prompting.

The last two commands permit a SFS program to communicate with the user, although in an admittedly restrictive manner.

c<comment> — This is a printing comment. That is, the comment is printed at the users teletype. Within the comment certain character sequences following a backslash (\) are given a special interpretation.

\<space> is output as a literal space character. If this was not available, it would be impossible to output multiple spaces.

\. is replaced in the output by the value of the current address.

\$<octal digit> is replaced in the output by the value of the corresponding global register.

\'<octal digit> is replaced in the output by the value of the corresponding local register.

\(expression) is replaced in the output by the value of the expression.

Lastly, if the comment ends with the backslash character, a newline character will not be appended to the output line.

i<register><comment> — This interaction command will print the comment with the same interpretation as given in the c command above, but then it reads an expression from the user's teletype, evaluates it, and assigns the value to the named global or local register.

FORMAT DEFINITION. The format definition, as the name implies, bears a structural and functional resemblance to the FORTRAN FORMAT statement. The following table enumerates the analogous structure in the two types of format.

FORTRAN	SFS
Parentheses used as list delimiters.	Braces or brackets used as list delimiters in order to prevent confusion with parentheses used as expression delimiters.
Statement number and keyword.	The keyword is superfluous, and the statement number is better rendered as a format name.
Primitive format item: count-type-width.	Primitive format item: storage unit-external form-count. The width is superfluous since the storage unit and the external form determine it uniquely. The count is placed last as is the more common practice for dimensioning information.
List format item.	List format item.
Literal item.	Skip external form for specification of constant or uninteresting fields in the structure.

The SFS format has two additional list items which are crucial to this application. There is a format name item, which allows one to represent a complex structure within the format as a single item. Without this feature formats would be essentially useless for representing complex structures.

The other additional format item is a label item. This is a name which is given to a position in the format definition. These labels provide convenient handles for referencing (naming) a substructure in the format definition. It should be noted, however, that although it is sometimes convenient and almost always harmless to think of a label as a name for the following format item with a storage representation, this is not the case at all. The only proper name for format item is the composite name formed by the format reference (see the next section) which generates the substructure containing just that item.

Labels may be placed anywhere in a list, and they need not be unique within a structure.

The SFS format has another useful feature which is not found in the FORTRAN FORMAT statment. It permits the item count to be any

valid parenthesised expression. This enables one to write parameterised format definitions thereby extending their applicability to structures with a variable configuration.

To be specific, a format definition consists of a name (a character string begining with an alphabetic) followed by a list of format items within braces (brackets may be substituted for braces). This list when expanded defines the extent and form of a structure which may then be accessed on the file. The base (position of the structure within the file) is not part of the format, and the list when expanded from left to right provides positive addressing offsets from the base to the individual elements of the structure.

```
<format> ::=        <format name><open><format list><close>

<format list> ::=   <format item>
                    <format item><fsep><format list>

<fsep> ::=          <sep>
                    ,

<sep> ::=           <null>
                    <sn*>

<open> ::=          {
                    [

<close> ::=         }
                    ]
```

A format item may be a primitive item which specifies a storage unit (byte, word, floating or double precision floating; denoted respectively by B, W, F and D) and an external form (ascii, octal or decimal; denoted respectively by A, O and D). The default form for the byte, word and floating point numbers are respectively A, O and D.

There is also a skip form (S). This form indicates that the associated storage space has no significance to the user, and may be disregarded in the dump and patch operations. A primtive item of the skip form is an error in the search operation.

There is also a null primitive item denoted N which does not define any storage space. This item is sometimes useful for filling out a format list to a desired length in order to make certain forms of format references (see the next section) easier to use.

A format item may consist of a list of format items within braces. An item may also be composed from a format name by placing that name between two periods. In the latter case, the effect

is the same as if the definition of the named format had been copied in place of the reference to its name.

A label is also a format item, and a single null label may be appended to any other item. The label item is formed by placing a name between a period and a colon (e.g., .here:), and a null label is just a colon.

Each format item is implicitly an array of substructure, and consequently, it may have a count (dimension) appended. The default count is always assumed to be one. (e.g., BD is a byte with a decimal form, and B8 is a array of eight ascii bytes. Note that while B8 and {B}8 define the same storage structure, eight consecutive bytes, the latter is defined as an array of eight arrays of one byte, and consequently, the two structures are treated differently.)

The count, if present, may be a constant or a parenthesised expression.

The existence of a reasonable interpretation for the count of a format item which does not specify storage space is in doubt, so a count supplied for such an item is treated as an error.

```
<format item> ::=  <item><count>
                   : <item><count>

<item> ::=         . <label> :
                   . <format name> .
                   . <label><open><format list><close>
                   <open><format list><close>
                   <primitive item>

<label> ::=        <name>

<count> ::=        <constant>
                   ( <expression> )
                   <null>

<primitive item> ::= n
                   <storage unit>
                   <storage unit><form>

<storage unit> ::= b
                   w
                   f
                   d

<form> ::=         a
                   o
                   d
                   s
```

The following is a set of format definitions which define the structure of a UNIX load module (*.o file).

```
lm{ .h:.header. .t:.text. .d:.data. .r:.relocation. .s:.symtable.}
```

The load module (lm) consists of a header, text (code), initialised data, relocation information and a symbol table.

```
header{ ws .t:wd .d:wd .bss:wd .s:wd ws3 }
```

The header contains the lengths in bytes of the text, initialised data, uninitialised data and symbol table. (The zero in the following definitions specifies the base address of the header - see FORMAT REFERENCES below.)

```
text{ w(0 header.t /2) }

data{ w(0 header.d /2) }

relocation{ w((0 header.t + 0 header.d)/2) }
```

There is one word of relocation information for each word of text or of initialised data.

```
symtable{ .symbol.((0 header.s)/12) }
```

Entries in the symbol table are 12 bytes long, and consist of an eight byte name followed by a word of flags and a value.

```
symbol{ .name:b8  .flag:w .value:wd }
```

The following illustration should clarify the structure of the load module.

```
lm    .h:.header.              ws
                       .t:     wd
                       .d:     wd
                       .bss:   wd
                       .s:     wd
                               ws3

      .t:.text.        w((0 header.t)/2)

      .d:.data.        w((0 header.d)/2)

      .r:.relocation.  w((0 header.t + 0 header.d)/2)

      .s:.symtable.    .symbol.((0 header)/12)   .name: b8
                                                 .flag: w
                                                 .value:wd
```

FORMAT REFERENCES. While SFS formats are structurally and functionally related to the FORTRAN FORMAT statement. The sole provision in FORTRAN for identifying an item in a format by completely expanding the structure and counting the primitive items, is hopelessly inadequate when dealing with complex structures. However, it is quite feasible to count format items within a list by counting a structure item or a list item as a single unit. Alternately, the labels that have been provided could substitute for such an item count.

If one locates the structure one desires to reference in its most immediately enclosing list or format item, one can then name it relative to that list or format item. Similarly this list or format can be named relative to its enclosing item, and the names concatenated to form a compound name for the originaly desired structure. Continuing in this manner until one reaches the level of structure definition (base level) that is convenient to address directly, one constructs a compound name (format reference) for the desired structure relative to this base level structure. A detailed description of the form used for such compound names is given below.

A substructure of a format may be described by selecting format items of succesive list items to whatever level (depth) is required. Since each level is an implied array of substructure including the zeroth level (the format itself) and the last level, which need not be a primitive list item, each level selection may be suffixed by a bracketed ([]) index specification consisting of zero, one, or two indices separated by a comma. The two index specification, identifies successive instances of the substructure at that level, starting with the instance given by the first index and continuing to the instance given by the second index. The zero and one index specification identify a single instance of the substructure. In particular, the zero index specification identifies the zeroth or initial instance.

A period (.) is used to separate two different level specifications within the format reference.

```
<format ref> ::=    <fname ref>
                    <fname ref> . <level specification list>

<fname ref> ::=     <format name><indexing>

<level specification list> ::=  <level spec>
                    <level spec> . <level specification list>

<level spec> ::= <item locator><indexing>

<indexing> ::=      <null>
                    [ <expression> ]
                    [ <expression> , <expression> ]
```

Items may be located within a list by label (i.e., a name), by item (exclusive of labels) count (i.e., an expression), by item count as an offset from a label item (i.e., a name followed by a colon followed by an expression), or by an item count as an offset from a label identified by a count of labels only (i.e., an expression followed by a colon followed by another expression).

Note that the use of an expression in the label count is restricted to those forms which do not begin with an alphabetic in order to avoid confusion between a label and a label count. Also note that in either count, zero denotes the initial item.

```
<item locator> ::= <label>
                   <non label item locator>
                   <label locator> : <non label item locator>

<label> ::=        <name>

<label locator> ::=  <label>
                   <restricted expression>

<non label item locator> ::= <expression>
```

It is important to note that a format reference, even if it is valid, does <u>not</u> identify any storage space until a base address has been supplied. In addition, it should be noted that the program will always use the value of the current address register as the base address if the user did not explicitly specify one, either through the use of explicit command addressing or as part of a based format reference (see EXPRESSIONS below).

The following are two examples (see also the example of the previous section) of valid format references:

```
symbol.name[0,4]
```

the first five characters of the name of some symbol table entry,

```
lm.s.0[0,9].name[0,4]
```

the first five characters of the names of the first ten symbols in the symbol table of some load module.

The construction of the format reference in this latter example proceeds as follows. The symbol table in the load module is located by the label "s", so that one may reference it by lm.s . The symbol table contains an array of symbols, format item number zero, so that the first ten entries in the table may be referenced by lm.s.0[0,9] . Each symbol in the table has the format name "symbol" and contains a name (ascii byte item) which can be located by the label "name", so that the first five characters of the desired symbols can be referenced by lm.s.0[0,9].name[0,4] .

This example illustrates the fact that a based format reference need not reference a single contiguous segment of storage. SFS will allow references which require more than thirty implied iterative statements (do loops) for their storage access.

EXPRESSIONS. An expression evaluates to a 32 bit twos complement integer. It may utilize the contents of the file by incorporating a based format reference. The reference is restricted in that it must evaluate to a single value of a primitive item. (A byte or word will not have its sign extended, and a floating point value will have its most significant four bytes interpreted as a twos complement integer.) A null base for the reference is interpreted as the current address (.).

The operators + - * / used in expressions are self explanatory. The other operators % | & ~ < > respectively stand for remainder from a division, bitwise or, bitwise and, bitwise addition, bitwise complement, left shift and right shift.

All expressions are evaluated left to right, with four levels of binding. The unary operators + - ~ are most closely bound, followed by the bitwise operators | & ~ and then by multiplication, division and shift. Addition and subtraction are the most loosely bound operators.

The difference between an expression and a restricted expression is simply that the restricted expression may not begin with an alphabetic character.

```
<expression> ::=        <E1>
                        <format ref>

<restricted expression> ::=  <E1>

<E1> ::=                <E2>
                        <E1><add op><E2>

<E2> ::=                <E3>
                        <E2><mult op><E3>

<E3> ::=                <E4>
                        <E3><bit op><E4>

<E4> ::=                <unary op><E4>
                        .
                        <constant>
                        <register>
                        <base><format ref>
                        ( <expression> )

<base> ::=              .
                        <constant>
                        <register>

<constant> ::=          <decimal integer>
                        0 <octal integer>

<decimal integer> ::=   <decimal digit>
                        <decimal digit><decimal integer>

<octal integer> ::=     <octal digit>
                        <octal digit><octal integer>

<add op> ::=            +
                        -

<mult op> ::=           *
                        /
                        %
                        <
                        >

<bit op> ::=            &
                        |

<unary op> ::=          +
                        -
                        ~

<register> ::=          '<octal digit>
                        $<octal digit>
```

CONDITIONS. The <conditions> in a search operation consists of a list of conditions all of which must be satisfied for the search to succeed. Each condition in turn consists of a test specification followed by a list of values which agree in length and form with the substructure specified by the format reference supplied. The conditions may be separated by spaces or the two character sequence backslash-newline (\<newline>).

The list of conditions must begin with either a newline or a question mark.

The test specifications include equals (= or <null>), not equals (!=), greater than (>), greater than or equals (>=), less than (<), less than or equals (<=), and bit wise logical implication (-> and <-). Bitwise logical implication may also be interpreted as bit inclusion, and both forms of the test are included since the user does not have the ability to specify the order of the two values to be tested.

Further information on conditions and the search command can be found above in the COMMANDS section. Also following the syntax below are two examples of the use of the search command.

```
<conditionals> ::= <newline><conjunction of conditions>
                   ? <conjunction of conditions>

<conjunction of conditions> ::=  <condition>
                   <condition><csep><conjunction of conditions>

<csep> ::=         <spaces>
                   \<newline>
                   <null>

<condition> ::=    <test><sep><multi-length value>

<test> ::=         >
                   <
                   >=
                   <=
                   =
                   !=
                   ->
                   <-
                   <null>

<multi-length value> ::=  <ascii list>
                   <octal list>
                   <decimal list>

<octal list> ::=   <octal value>
                   <octal value><sn*><octal list>

<decimal list> ::= <decimal value>
                   <decimal value><sn*><decimal list>
```

```
<floating octal list> ::=    <floating octal value>
                             <floating octal value><sn*><floating octal list>

<floating decimal list> ::=  <floating decimal value>
                             <floating decimal value><sn*><floating decimal list>

<octal value> ::=  <octal integer>
                   - <octal integer>

<decimal value> ::=  <decimal integer>
                     - <decimal integer>

<floating octal value> ::=   <sign><fixed octal>
                             <sign><fixed octal> o <sign><octal integer>

<fixed octal> ::=  <octal integer>
                   . <octal integer>
                   <octal integer> . <octal integer>

<sign> ::=         +
                   -
                   <null>

<floating decimal value> ::= <sign><fixed decimal>
                             <sign><fixed decimal> e <sign><decimal integer>

<fixed decimal> ::=  <decimal integer>
                     . <decimal integer>
                     <decimal integer> . <decimal integer>
```

As an example of the use of conditions, consider the following command which locates all symbols in the ascii collating sequence between ia... and iz.... The current address is taken to be at the start of the symbol table.

```
.,(.+0 header.s)s-symbol.name[0,1] ? >=ia <=iz
```

For a second example of the use of the search command, consider the following short program which locates and dumps the external symbols in the symbol table of a load module. The program saves the value of the current address register, so that it can be restored upon termination. It then initialises the current address to the start of the symbol table, and the first local register to the end of the table. It then enters a loop which locates and dumps a symbol table entry. The entries represent external symbols if they have flag bit 5 (value 040) on. Finally, the loop exits when no additional external symbols have been found.

This program assumes that two formats given in a previous example have been defined, although this assumption could have been dispensed with by the use of the "f" and "u" commands.

```
'2=.
.=(0 header.t + 0 header.d)<1 +16
'1=.+0 header.s
          .,('1)s+ symbol.flag? <- 040
          ($0),0j  +4
                d  symbol
          (.+12)r
                j  -4
.='2
```

ASCII OUTPUT. The program prints a single ascii value as a two position character sequence. This enables all 256 byte values to be represented. The first position is the prefix, and may contain any combination of the three characters space, backslash (to be read control), and underscore (to be read shift).

The control prefix indicates that the value of the character represented is to be anded with octal 37, and the shift indicates that octal 200 is to be added to the character's value. The space used as a prefix, of course, indicates the indicated value is to be unmodified. (e.g., ' A' denotes the value 101; '\A' denotes the value 1; '_A' denotes the value 301, and '\A' denotes the value 201.)

Because the two ascii characters rubout/delete and space do not have a convenient representation in this scheme, the compound character \ has been adopted for rubout/delete, and the compound character _ has been adopted for space.

ASCII INPUT. The ascii character set is not able to represent all 256 possible byte values. Furthermore, not only is it inconvenient to transmit some characters, but the characters space and newline are not available in ascii input, because they serve as separators. These problems are solved by treating the backslash (to be read as control) character specially.

In the input, the control character acts exactly as if the control key of the teletype were depressed along with the character following the control character. That is, it ands the following character with octal 37. This is true except when the character following control (\) is one of the following, 0123456? \ or space. Control (\) followed by an octal number of up to three digits length yields the value of that number, and control space yields space while control backslash yields backslash.

FOR NEW USERS. Although a conscious effort was made to keep the SFS language simple and straightforward, it is sufficiently complex that the effort required to learn its use may dismay the casual user. He can, however, get by quite well with just the following twelve commands.

<null>	print the current address.
q	quit.
=<expression>	set the current address.
:<expression>	print the value of an expression.
!<UNIX command>	escape to UNIX.
f<format>	define a format.
l	list formats defined.
l<format name>	list a format definition.
d<format ref>	dump structure.
p<format ref>	patch structure.
s+<format ref>	search for structure and set "." .
s-<format ref>	search for structure.

The quickest way for a new user to become acquainted with these commands is to copy a load module and experiment with the commands using the definitions in the above examples.

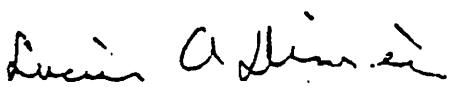
The new user accustomed to FORTRAN indexing, may also have some difficulty at first with the zero base indexing used in SFS.

While it does take some daring to program a complex function in SFS, an example in the appendix is given as a demonstration of its feasibility.

MH-1271-LAD

Attached
Appendix

L. A. DIMINO



APPENDIX

Four useful programs have been written in SFS to provide a debugging facility similiar to that provided by the UNIX DB command. They illustrate the feasibility of constructing fairly complex debugging facilities upon the basic structure provided by SFS, and they did not present any particular programming difficulty.

sfsprg	This examines the subject file and determines whether it is a load module with, or without, relocation information, or a core image file. It then defines the structures appropriate to the file and initialises the global registers. It is normally a prerequisite to the use of the other three programs.
sfsoav	This prints a 16 bit value in octal and ASCII. It will also print the relevent relocation data, when this is available and requested.
sfsda	This is a deassembler for the PDP11/45. It is a fairly complex program which throughly exercises the jump command.
sfssym	This searches the symbol table of a load module for the symbol of a specified type which most closely approximates the given value.

Listings of sfsprg and sfssym are provided below as examples of the use of SFS. The other two programs are not included because their translating and formating function are less typical of the anticipated use of SFS. However, all four programs are presently available under the catalog /usr/lad on the center 127 UNIX system.

```
"       initialise for load module or core image
('2),0j!=19
        i'2Parameter? If you don't understand, type 0<nl>
        ('2),0j!=17
        cResponse is -1 if minimum output is desired
        c       +1 for report of register initialization
        c       +2 for report of formats defined by name
        c       +4 for report of format definitions
        cFor example,
        c5 requests reports for registers and format definitions
        c
        cthe parameter the can also be given in the call
        c%/usr/lad/sfsprg 5
        c
        canother parameter (file length) can be given for core images
        c
        c%/usr/lad/sfsprg 5,5120
        c
        i'2Parameter?
        ('2),0j=175
r w{w}
(0 w.0),0407j!=100
        $7=16
        '7= -1
        f header{ws .t:wd .d:wd .bss:wd .s:wd ws3}
        '4=(0header.t)>1
        ('4),0j=3
                '7='7+2
                f text{w('4)}
        '5=(0 header.d)>1
        ('5),0j=3
                '7='7+4
                f data{w('5)}
        '6=(0 header.s)/12
        ('6),0j=4
                '7='7+6
                f symtable{.symbol.('6)}
                f symbol{.name:b8 .flag:w .value:wd}
        (14w.0),0j!=39
                j('7)
                        f lm{.h:.header. .t:.text. .r:.relocation. }
                        j10
                        f lm{.h:.header. .d:.data. .r:.relocation. }
                        j8
        f lm{.h:.header. .t:.text. .d:.data. .r:.relocation. }
                        j6
        f lm{.h:.header. .t:.text. .r:.relocation. .s:.symtable.}
                        j4
        f lm{.h:.header. .d:.data. .r:.relocation. .s:.symtable.}
                        j2
r lm{.h:.header. .t:.text. .d:.data. .r:.relocation. .s:.symtable.}
                f relocation{w((0 header.t+ 0 header.d)>1)}
                $2=2
                $3=16+(2 w.0)
                $4=$3+(4 w.0)
                $5=($4-16)<1+16
```

```
        $6=$5+(8 w.0)
        cload module with relocation
        ('2),(-1)j=136
        ('2&1),0j=9
                cGLOBAL registers
                c$2=2   relocation flag
                c$3=$3  start of data segment in file
                c$4=$4  start of relocation in file
                c$5=$5  start of symbol table
                c$6=$6  length of file
                c$7=16  addressing offset for text or data
                c
        ('2&2),0j=4
                cFormats by name
                l
                c
        ('2&4),0j=122
                cFormat definitions
                l relocation
                j35
no relocation data on the file
j('7)
        f lm{.h:.header. .t:.text. }
        j10
        f lm{.h:.header. .d:.data. }
        j8
        f lm{.h:.header. .t:.text. .d:.data. }
        j6
        f lm{.h:.header. .t:.text. .s:.symtable. }
        j4
        f lm{.h:.header. .d:.data. .s:.symtable. }
        j2
        f lm{.h:.header. .t:.text. .d:.data. .s:.symtable.}
$2=0
$3=16+(2 w.0)
$4=$3+(4 w.0)
$5=$4
$6=$4+(8 w.0)
cload module without relocation
('2),(-1)j=99
('2&1),0j=8
        cGLOBAL registers
        c$2=0   no relocation flag
        c$3=$3  start of data segment on file
        c$5=$4=$4       start of symbol table
        c$6=$6  length of file
        c$7=16  addressing offset for text or data
        c
('2&2),0j=4
        cFormats by name
        l
        c
('2&4),0j=86
        cFormat definitions
        l lm
        l header
```

```
                ('4),0j=2
                        1 text
                ('5),0j=2
                        1 data
                ('6),0j=78
                        1 symtable
                        1 symbol
                j75
"       core image file
('3),0j>5
        i'3File length? type 0 <nl> if you don't know
        ('3),0j>3
        cType !stat <file name> to get file length and try again
        j69
$6='3
$5=$6-512
$7=0
1 core{ .t:.text. .s:.stack. .c:.cntrl. }
f cntrl{ .us:w .usp:w .sp:w .break:w w .pse:w
  .core:w w(217) .fdr:w12 .fr:w12 .r:.regs. }
f regs{ .r5:w .r4:w .r3:w .r2:w .r1:w .r0:w .pc:w .ps:w }
ccore image -
j(($5cntrl.pse)&017<1+1)
        ctype 0
        j30
        cbus err
        j29
        cilleyal inst
        j26
        ctrace or bpt trap
        j24
        ciot trap
        j22
        cpower
        j20
        cemt trap
        j19
        csys call err
        j17
        cprogrammed interrupt
        j15
        cFP exception
        j12
        cmemory
        j10
        cquit
        j9
        cinterrupt
        j7
        ckilled
        j5
        ctype 16
        j2
        ctype 17
        c fault
$3=$5 cntrl.break
```

```
$4=$5cntrl.sp
$2=$3-$4
f text{ w($3>1)}
f stack{ w(($5-$3)>1)}
('2),(-1)j=22
('2&1),0j=10
        cGLOBAL registers
        c$2=$2  address correction (-offset) for stack references
        c$3=$3  address of program break
        c$4=$4  address of the end of the stack, $4+$2 is the address
        c               of the end of the stack in the file
        c$5=$5  address in the file of the per user area
        c$6=$6  length of the file
        c$7=0   addressing offset for text or data
        c
('2&2),0j=4
        cformats by name
        1
        c
('2&4),0j=7
        cFormat definitions
        1 core
        1 text
        1 stack
        1 cntrl
        1 regs
u w
```

```
"       search symbol table for entry with closest value
"       start,end%sfssym value,type
"       start is address in table where search begins
"       and end is the address where it ends
"       if start is not supplied, search begins at first symbol
"       type =0 or <null> if any symbol will do
"       type =1 for absolute symbols, =2 for text symbols
"       =3 for data symbols, =4 for bss symbols, and one adds
"       +8 for external symbols only of desired type (0,1,2,3,4)
'3='3+'3&8*3
(('0-$5)%12),0j!=3
('0),($6)j>=2
('0),($5)j>=2
'0=$5
(('1-$6)%12),0j!=3
('1),($6)j>=2
('1),('0)j>=2
'1=$6
'4=256000
        '5='0symbol.value -'2
        ('3),0j=3
        ('3),('0symbol.flag)j->2
        j7
        ('5),0j=9
        ('5),0j>2
        '5= -'5
        ('5),('4)j>=3
        '4='5
        '6='0
        '0='0+12
        ('0),('1)j< -11
j2
'6='0
cSymbol (('6-$5)/12) address ('6)
('6)d symbol
"end of srssym
```