DCE 29-MAY-71 8:53 7049

Note to WLB re 7048

Walt: Re. (Journal, JRNL1, J7048:gw), maybe if you checked to see if the Output Processor did right by the printout of (Journal, 7017,) before WSD has to start worrying about how he fixes up the Journal-entry Processor ??? Thanks, Doug. 1

DCE 29-MAY-71 8:53 7049

Note to WLB re 7048

(J7049) 29-MAY-71 8:53; (Expedite) Title: Author(s): Douglas C. Engelbart/DCE; Distribution: Walter L. Bass, William S. Duvall, James C. Norton/WLB WSD JCN; Clerk: DCE;

DCE 29-MAY-71 9:14 7050

Rough Notes for Talk to Prof. Parker's Information Science Colloquium, Stanford, 27 May 71

Parker was away. Tom Martin (Grad Student) handled the meeting. Don Dunn, Bob Kinchloe, and Hank Epstein were only staff I recognized. Some will organize for group visit here; Martin will coordinate. Epstein interested in ARPA Net possibilities sharing MARC Files, e.g. with Ohio's Library Net, and with others.

DCE 29-MAY-71 9:14 7050

Rough Notes for Talk to Prof. Parker's Information Science Colloquium, Stanford, 27 May 71

DCE 29-MAY-71 9:1h 7050

Rough Notes for Talk to Prof. Parker's Information Science Colloquium, Stanford, 27 May 71

<JOURNAL>7050.NLS;1, 29-MAY-71 9:15 DCE ; Title: Author(s): Douglas C. Engelbart/DCE; Clerk: DCE;

L|O

A Programming Language for the Augmentation Research Center

This file describes a new programming language, L|O, for use on the PDP|O. The language contains some high level features for operations such as string analysis and manipulation which are implemented in the language as calls on library routines. In addition, L|O has basic constructs such as local variables and fields which have been particularly useful. The L|O compiler was written using the compiler-compiler system Tree Meta.

17

18

The definition of the syntax is given as abbreviated Tree Meta parse rules having the following form: 19

A right part of a syntax rule consists of one or more alternatives. If there is more than one alternative, they are separated by slashes (/). Each alternative consists of a sequence of elements. Any subsequence enclosed in square brackets, [and], is optional. All other elements in the sequence must occur in the specified order. 19a

The elements may be any of the following: 19b

the name of a rule; 19b1

a call on a basic recognizer which tests the input for one of the following 19b2

.ID -- recognizes a lower case identifier, 19b2a

.UID -- recognizes an upper case identifier, 19b2b

.NUM -- recognizes a number, 19b2c

.SR -- recognizes a string enclosed in quotes ("), 19b2d

.SR| -- recognizes a single character preceeded by an apostrophe ('), or 19b2e

.CHR -- recognizes any character; 19b2f

a string enclosed in quotes ("); 19b3

a single character string indicated by an apostrophe (') followed by the character; 19b4

a list of alternatives enclosed in parentheses; 19b5

a dollar sign ($) followed by an element, which means an arbitrary number of occurences (including zero) of the element. 19b6

Comments enlosed in percent signs (%) may be embedded anywhere in the rule. The rule is terminated by a semicolon (;). 19c

L|O Documentation

VARIABLE FORMS

20

The rule "lhs" (left hand side) gives the syntax for the variable forms. These variables may be used on the left hand side of assignment statements as well as in expressions and certain other syntactic positions. 20a

```
lhs = (fwlhs / pwlhs) qualifiers;                                  20b

fwlhs = % full word left-hand-side %                               20c

  global / register / local / pointer / unref;                     20c1

pwlhs = % partial word left-hand-side %                            20d

  field / character;                                               20d1
```

Global variables 20e

```
global = (.ID / link) [ '[ exp '] ];                               20e1
```

Global variables may be indexed. They are allocated from fixed locations rather than a stack. Where a link is used instead of a simple identifier, the meaning is exactly the same as if only the lower case identifier in the link had been used. 20e2

```
link = '< [.UID] [',] [.UID] [',] .ID [': $.CHR] '>;               20e3
```

This construct allows use of the NLS linking commands when writing/studying/modifying programs. The first uppercase identifier gives the directory name and the second gives the file name. 20e4

Register variables 20f

```
register = .ID; %declared to be a register%                        20f1
```

A register variable is simply a global variable that is kept in a fast register rather than memory. The variable may be used exactly like any other global variable, except it will result in faster access. There can be only a few register variables in a program since the compiler makes use of several registers for the evaluation of expressions and for stack and record pointers. 20f2

Registers 0 to 6 may be used by the programmer. Registers 7 and 8 may be used if the program does not use the string construction or analysis facilities. Registers 9 and 10 are the stack pointer and mark respectively. Register 11 is used as a record pointer. The remaining registers are used by the compiler for evaluating expressions. 20f2a

Local variables 20g

```
local = .ID;  %formal parameter or local variable%
```
20g1

Local variables include formal parameters of procedures and variables declared at the head of a procedure by the "LOCAL" declaration. Each time a procedure is called, space is allocated for its local variables. The space is released when the procedure returns. Thus procedures may be used recursively and each activation of the procedure will have its own set of locals. 20g2

The local variables of a procedure may be referenced only from within that procedure. The names used for local variables may be used for (different) local variables in other procedures. 20g3

Local variables are limited to scalars, and thus may not be indexed. 20g4

Generalized pointer variables 20h

```
pointer = '[ exp ']';
```
20h1

Generalized pointer variables refer to the location whose address equals the value of the expression. 20h2

In the simplest case where the expression is a (global or local) variable, indirect addressing is used. More complex expressions are evaluated and then indexing is used. 20h3

Unref variables 20i

Global, register, and local variables may be declared to be references by the REF statement. Uses of a reference variable are treated as if they were surrounded by square brackets (i.e. as a pointer). Thus if "x" is a reference variable, then uses of x are treated like *[x]*. The "unref"

form allows the programmer to override the reference mode so as to access the variable itself. 20i1

unref = '& .ID; %where the identifier is a REF variable.% 20i2

Field variables 20j

Fields may be used either as an "unqualified left hand side" or as a qualifier of some other left hand side. 20j1

field = '. fieldname / '↑ fieldname; 20j2

qualifiers = $('. fieldname); 20j3

fieldname = .ID; 20j4

The identifier used as a "fieldname" should be the name of a local or a global variable which holds a field descriptor. Field descriptors are created by FIELD or RECORD declarations or by the MKFD (make field descriptor) primitive. 20j5

A field descriptor consists of the size, position, and address of the field. The size is the number of bits in the field and the position is the number of bits in the word to the right of the field. The address may include indexing and indirection. 20j5a

A fieldname preceded by a period (.) accesses the field described by the fieldname. If preceded by an uparrow (↑), then the descriptor is "incremented" before the access is made. 20j6

A field descriptor is incremented by subtracting the size from the position. If the resulting position is negative, then the address is increased by one and the position is set to 36 minus the size. 20j6a

Any variable form may be qualified by following it by a period and a fieldname. This results in accessing the contents of the named field within that variable. It is possible to refer to a field within a field. For example, to zero the c field of the b field of the variable k the statement 20j7

k.b.c ← 0 20j7a

may be used. 20j8

Eventually a more general data definition facility will be included in L|O. Until that time this limited form is significantly better than nothing. The use of symbolic names for fields has dramatically improved the readability of L|O programs for NLS and has received a "how did we ever get along without it" kind of reception. 20j9

Character variables 20k

character = '* stringname '* '[exp ']; 20k1

stringname = fwlhs; 20k2

The stringname may be an arbitrary (full word) variable form but should result in a reference to a string declared by the "DECLARE STRING" statement, described below. 20k3

The value of the expression following the stringname determines which character in the string is being referenced. The first character in the string has an index of one, the second has an index of two, and so on. 20k4

There are two predefined fields for use with strings. The field denoted L gives the current length of the string; the field M gives the maximum length allowable for the string. 20k5

Example: 20k6

Assume the declaration 20k6a

DECLARE STRING str [20]; 20k6a1

and the execution of the statement 20k6b

str ← "hello". 20k6b1

Then str.M equals 20 and str.L equals 5. 20k6c

Reading a character position beyond the current length of the string returns a special value, ENDCHR (= 377 octal). Writing a character position beyond the current length and not exceeding the maximum length causes that length to be increased to include the new character. 20k7

STATEMENT FORMS

2|

```
labeled = [label] stat;                                          2|a

label = '( .ID ') ':;                                            2|b

stat =                                                           2|c

  assign / multipleassign / conditional / iterative /
  transfer / block / bump / stacksnrings / divid / builtin /
  special / null;                                                2|c|

assign = lhs '← exp;                                             2|d

  The expression is evaluated and then stored into the left
  hand side.                                                     2|d|

multipleassign = '( lhs $(', lhs) ') '← '( exp $(', exp) ');     2|e

  The expressions are evaluated and the values pushed on a
  stack provided by the system.  Then the values are popped
  from the stack and stored into the appropriate left hand
  side.  The order of evaluation of the expressions is left
  to right.  Thus for                                            2|e|

  (a, b) ← (a+b, a-b)                                            2|e2

  the expression a+b is evaluated and stacked, expression a-b
  is evaluated and stacked, the value of a-b is popped and
  stored into b, and finally, the value of a+b is popped and
  stored into a.                                                 2|e3

  Naturally, the number of expressions must equal the number
  of lhs's.                                                      2|e4

conditional = if / case / onsignal;                              2|f

  There are three types of conditional statements, the common
  "IF" statement, a "CASE" statement, and the "ONSIGNAL"
  statement.                                                     2|f|

   if = "IF" exp "THEN" labeled ["ELSE" labeled];                2|f2

   case = "CASE" exp "OF" $casest "ENDCASE" labeled;             2|f3
```

```
casest = [label] binrel $(', binrel) ': labeled ';;          2|f4
```

The CASE-statement provides a means of executing one statement out of many. The expression after the word "CASE" is evaluated and the result left in a register. This is used as the left-hand side of the binary relations at the beginning of the various cases. Several relations may be listed at the start of a single statement; the statement will be executed if any of the relations are statisfied. If none of the relations are satisfied, the statement following the word "ENDCASE" will be executed. 2|f5

Example: 2|f6

```
CASE c OF
   = a: x ← y;                    %c = a%
   > b: (x, y) ← (x+y, x-y);      %c > b%
   ENDCASE y ← x;                 %c # a AND c <= b%          2|f6a
```

If a case ends with an unconditional transfer, the compiler does not produce another (unnecessary) branch instruction. 2|f7

```
onsignal = "ON" "SIGNAL" $sigstatement "ELSE" stat;          2|f8

sigstatement = binrel $(', binrel) ': stat '; ;              2|f9
```

Any procedure may have an active ON SIGNAL statement. The statement becomes active when control reaches it during the execution of the procedure. Only one ON statement is active at a time in a particular procedure. The body of the ON statement is similar to a CASE statement and is executed as a result of a SIGNAL statement (see below) in a procedure reached by some sequence of calls starting in the procedure in which the ON SIGNAL statement is defined. 2|f|0

The binary relations at the front of the statements in the ON SIGNAL statement test the value stored in the system signal variable "sysgnl" by the SIGNAL transfer statement. Additional information may be passed by the signalling mechanism through the system message variable "sysmsg". 2|f|0a

There is an implicit SIGNAL transfer statement following the ON statement; if comtrol falls through the case tests, the SIGNAL will be propogated to earlier procedures. 2|f|0b

A principle use of the SIGNAL mechanism will be error handling. 2|f||

iterative = loop / while / until / do / for; 2|g

loop = "LOOP" labeled; 2|g|

The statement following the word "LOOP" is repeatedly executed until control leaves by means of some transfer instruction within the loop. 2|g|a

while = "WHILE" exp "DO" labeled; 2|g2

Equivalent to 2|g2a

(label): IF NOT exp THEN GOTO out;
labeled; GOTO label; (out): 2|g2a|

until = "UNTIL" exp "DO" labeled; 2|g3

Equivalent to 2|g3a

(label): IF exp THEN GOTO out;
labeled; GOTO label; (out): 2|g3a|

Thus the word "UNTIL" has the same effect as "WHILE NOT". 2|g3b

do = "DO" labeled ("UNTIL" / "WHILE") exp; 2|g4

This is like the above, except that the logical test is made after the statement has been executed rather than before. Thus "DO labeled WHILE disjunct" is equivalent to: 2|g4a

(label): labeled; IF exp THEN GOTO label; 2|g4a|

and "DO labeled UNTIL exp" is equivalent to: 2|g4b

(label): labeled; IF NOT exp THEN GOTO label;. 2|g4b|

Thus the controlled statement is always executed at least once (the first time before the test is made). 2|g4c

for = 2|g5

"FOR" lhs *['← exp]* ("UP" / "DOWN") *[exp]*

"UNTIL" ('= / '# / ">=" / "<=" / '> / '<) exp "DO"
labeled; 2|g5a

If no initialization expression is given then the variable is left with its current value. 2|g5b

If the optional increment expression is not given, a value of one is used. The increment is added to the named variable if "UP" is specified; it is subtracted for "DOWN". 2|g5c

The relation given after the "UNTIL" is used to determine whether or not to do another iteration. 2|g5d

Example: 2|g5e

```
FOR k ← n UP j UNTIL > m*3 DO x[k] ← k;
```
2|g5e1

is equivalent to 2|g5e2

```
k ← n;
GOTO test;
(loop): k ← k + j;
(test): IF k > m*3 THEN GOTO out;
x[k] ← k;
GOTO loop;
(out):
```
2|g5e3

Note that the increment and bound expressions are recomputed on each iteration. 2|g5f

transfer = 2|h

"CALL" fwlhs [args] / fwlhs args / 2|h|

There are two forms of the procedure call statement, one starting with the word "CALL" and optional arguments, and the other without the word "CALL" and manditory argument list. The argument list is described below. 2|h|a

"RETURN" ['(exp $(', exp) ')] / 2|h2

A procedure may return an arbitrary number of results. The order of evaluation of results is from left to right. 2|h2a

"SKIP" "RETURN" ['(exp ')] / 2|h3

This causes the procedure to return to the calling location+2. To test whether a procedure has done a SKIP RETURN, use the SKIP relation described below. 2|h3a

"EXIT" ("CASE" [.NUM] / ["LOOP"] [.NUM]) / 2|h4

This construct provides for forward branches out of CASE or iterative statements. The optional number specifies the number of lexical levels of CASE or iterative statements respectively that are to be exited. If a number is not given then 1 is assumed. All of the iterative statements (LOOP, WHILE, UNTIL, DO, FOR) can be exited by the EXIT LOOP construct. 2|h4a

EXIT and EXIT LOOP have the same meaning. 2|h4b

Examples: 2|h4c

```
LOOP
   BEGIN
   ........
   IF test THEN EXIT;
   %the EXIT will branch out of the LOOP%
   ........
   END;
```
2|h4c1

```
UNTIL something DO
   BEGIN
   ........
   WHILE test1 DO
      BEGIN
      ........
      IF test2 THEN EXIT;
      %the EXIT will branch out of the WHILE%
      ........
      END;
   ........
   END;
```
2|h4c2

```
UNTIL something DO
   BEGIN
   ........
   WHILE test1 DO
      BEGIN
      ........
      IF test2 THEN EXIT 2;
      %the EXIT 2 will branch out of the UNTIL%
```

```
            ........
               END;
            ........
            END;                                              2|h4c3

         CASE exp OF
            =something:
               BEGIN
               ........
               IF test THEN EXIT CASE;
               %the EXIT will branch out of the CASE%
               ........
               END;
            ........                                          2|h4c4
```

"REPEAT" ("LOOP" *[.NUM]* / *["CASE"]* *[.NUM]* *['(exp ')]*) / 2|h5

This construct provides for backward branches to the front of CASE or iterative statements. The optional number has the same meaning as in the EXIT statement. 2|h5a

If an expression is given with the REPEAT CASE, then it is evaluated and used in place of the expression given at the head of the specified CASE statement. If the expression is not given, then the one at the head of the CASE statement is reevaluated. 2|h5b

REPEAT and REPEAT CASE have the same meaning. 2|h5c

Examples: 2|h5d

```
   CASE exp| OF
      =something:
         BEGIN
         ........
         IF test| THEN REPEAT;
         %REPEAT with a reevaluated exp|%
         ........
         IF test2 THEN REPEAT(exp2);
         %REPEAT with exp2%
         ........
         END;
      ........                                                2|n5d|

   LOOP
      BEGIN
      ........
```

```
        IF test THEN REPEAT LOOP;
        %REPEAT LOOP will go to the top of the LOOP%
        ........
        END;                                                    2|h5d2
```

It is worth noting that the availability of EXIT and REPEAT statements has resulted in clearer programs which are generally without labels and GOTO's. The EXIT and REPEAT replace GOTO's to the start or end of the most common compound forms. By providing implicit labels in these positions for use with EXIT or REPEAT, explicit labels are avoided. 2|h5e

"GO""TO" (lhs / "STATE") / 2|h6

Goto provides for unconditional transfer of control to a new location. The statement GOTO STATE transfers control to the location of the most recent STATE definition (see below). 2|h6a

"SIGNAL" *['([exp] [', exp] ')]* ; 2|h7

The SIGNAL statement transfers control to the first active ON SIGNAL statement back in the calling sequence starting with the procedure that called the procedure containing the SIGNAL statement. 2|h7a

If the first optional expression is present in the SIGNAL statement, it is evaluated and stored into the system signal variable "sysgnl". 2|h7a|

The binary relations at the front of the statements in the ON SIGNAL statement test the value stored in the variable "sysgnl". The contents of this variable may be accessed through the use of the primitive SIGNAL. (See primitives below.) Additional information may be passed by the signalling procedure by means of the variable "sysmsg". 2|h7a|a

If the second optional expression is present, it is evaluated and stored into the system message variable "sysmsg". 2|h7a2

procedure call argument list 2|h8

args = '(*[exp $(', exp)] [': lhs $(', lhs)]* '); 2|h8a

The argument list consists of an arbitrary number of expressions separated by commas. This variable may be accessed by the primitive MESSAGE. It is not necessary for the number of arguments to equal the number of formal parameters for the procedure. The argument expressions are evaluated in order from left to right. 2|h8b

Following the arguments there may be a list of locations for multiple results to be returned. The list of left hand sides for multiple results is separated from the list of argument expressions by a colon. The number of locations for results need not equal the number of results actually returned. If there are more locations than results, then the extra locations get an undefined value. If there are more results than locations, the extra results are simply lost. (This format for multiple results is patterned after SPL and QSPL of Peter Deutsch and Butler Lampson). 2|h8c

Example: 2|h8d

If procedure p ends with the statement 2|h8e

```
RETURN (a,b,c)
```
2|h8e|

then the statement 2|h8f

```
q ← p(:r,s)
```
2|h8f|

results in (q,r,s) ← (a,b,c). 2|h8g

block = "BEGIN" labeled $('; labeled) "END"; 2|i

The block statement simply allows the grouping of several statements into one. 2|i|

The hierarchical structure of the NLS file may eventually replace the use of BEGIN's and END's as a means of forming compound statements. 2|i2

bump = "BUMP" ["DOWN"] lhs $(', lhs); 2|j

The BUMP statement increments each of the variables, while the BUMP DOWN statement decrements each one. 2|j|

stacksnrings = %stacks and rings% 2|k

"PUSH" recordname "ON" storename / 2|k|

"POP" storename *["TO" recordname]* / 2|k2

"RESET" storename; 2|k3

recordname = fwlhs; 2|k4

storename = fwlhs; 2|k5

Stacks and ring buffers may be declared by the DECLARE STACK/RING statement. Ring buffers are like stacks except that instead of giving an error on overflow or underflow, they wrap around. Thus overflow moves the pointer to the bottom element and underflow moves the pointer to the top element. 2|k6

The elements that are pushed and popped may consist of more than one word of storage. All elements are of the same size however. 2|k7

The PUSH statement pushes the record from the specified location onto the specified store. 2|k8

The POP statement removes the top record from the store and if a destination is specified stores the record into it. 2|k9

The RESET statement removes all items from the store by reseting the pointer. 2|k|0

The storename may also be used to access the top record on the store. 2|k||

If "stk" is a storename, then *[stk]* is the first word of the top record on the stack. 2|k||a

If the records take a single word, then *[stk-1]* is the next lower record, *[stk-2]* is the one below that, and so on. 2|k||b

Eventually L|O will include more powerful storage facilities modelled after the AED free storage package of Doug Ross. 2|k|2

divid = "DIV" exp ', quotient ', remainder; 2|1

quotient = lhs; 2|1|

```
  remainder = lhs;                                                  2|12
```

The central connective in the expression must be '/. This statement allows both the quotient and the remainder of the division to be saved. 2|13

```
builtin = '! (.UID /.ID) %opcode% [.ID %acc% ',] address;           2|m

  address =                                                         2|m|
    ['@ %indirect%] (adel / '= adel)                                2|m|a
    ['( .ID %index% ')];                                            2|m|b
  adel = .SR / .ID / ['-] literal;                                  2|m2
```

The builtin allows the programmer to use assembly-language-like statements. Hopefully the need for this ability will be small, but a language for systems programming should provide means for handling those cases. 2|m3

The opcode must be defined in the program by means of a SET declaration. Likewise the identifiers used for the accumulator or index fields of the instruction must be defined thru REGISTER or SET declarations. 2|m4

The address part of the instruction may be specified in the following ways: 2|m5

address part	address field of instruction contains	
.....	.....	2|m5a
.ID	the address of the identifier	2|m5b
['-] .NUM	the [negative] number	2|m5c
'= .ID	the address of a literal containing the address of the identifier	2|m5d
'= ['-] .NUM	the address of a literal containing the [negative] number	2|m5e

```
null = ["NULL"];                                                    2|n
```

The null statement is provided as a convenience to the programmer. 2|n|

The following special statement forms are for the most part implemented by means of procedure calls on routines in NLS. The statements allow for complex string analysis and construction as well as providing constructs designed to simplify the specification of feedback and other common tasks within NLS. 2|o

```
special =  %special statements forms%                          2|p

  ccpos / pattern / stringconstruction / inputstat /
  feedbackline / group / state / entity / deletemarker;         2|p1

ccpos = "CCPOS" (pos / '* stringname '* ['[ exp ']]);           2|q
```

This sets up the "Current Character POSition" for string analysis. All string tests start their search from the current character position. 2|q1

```
  pos =   %position in a statement or string%                   2|q2

    "SF(" stspec ') /
      %String Front  --  left of the first character%           2|q2a

    "SE(" stspec ') /
      %String End  --  right of the last character%             2|q2b

    textpointer;                                                2|q2c

  stspec = textpointer / '* stringname '*;                      2|q3

  textpointer = .ID;                                            2|q4
```

A text pointer points between two characters in a statement or string. 2|q5

An alternative convention would have the pointer specify a particular character. This in fact was the initial design. However, by putting the pointers between characters a single pointer can be used both to mark the end of one substring and the begining of the substring starting with the next character. This can result in an appreciable simplification of string manipulation algorithms. (Doug Engelbart suggested this approach). 2|q5a

The variable holding a text pointer is declared by a DECLARE TEXT POINTER statement. There is a special declaration for these because text pointers require more

than a single word of storage. The identifier used as a text pointer may be such a variable or a reference, defined by a REF statement, to such a variable. 2|q6

If a text pointer is given after CCPOS, then the character position is set to that location. 2|q7

Unless the String End option is used in specifying the position, the scan direction is initialized to read from left to right. When the position is specified as a String End, then the scan direction is set right to left. 2|q8

If a stringname is given after CCPOS, then the position is moved to that string. The scan direction is set left to right. 2|q9

Indexing the stringname simply specifies a particular position within the string. Thus *str*[3] puts the current character position between the second and third characters of the string "str". If the scan direction is left to right, then the third character will be read next. If the direction is right to left, then the second will be read next. 2|q9a

If no indexing is given, then the position is set to the left of the first character in the string. This is equivalent to an index of 1. 2|q9b

pattern = "FIND" union; 2|r

This specifies a string pattern to be tested starting from the current character position. If the test succeeds the character position is moved past the last character read. If the test fails the character position is reset to the position prior to the test. 2|r1

union = intersection ["OR" union]; 2|r2

If the "intersection" is false, then the character position is reset to where it was before the "intersection" was tested and the test following the "OR" is made. 2|r2a

intersection = negation ["AND" intersection]; 2|r3

If the "negation" is true, then the character position

```
  is reset to where it was before the "negation" was
  tested and the test following the "AND" is made.          2|r3a

negation = "NOT" negation / alternatives;                   2|r4

alternatives = concatenation [ '/ alternatives ];           2|r5

  If the "concatenation" is false, then the character
  position is reset to where it was before the
  "concatenation" was tested and the test following the '/
  is made.                                                  2|r5a

concatenation = $(nelement / element);                      2|r6

  A concatenation is true if each of its elements are.      2|r6a

nelement =  %executed for effect - are always true%         2|r7

  pos   /
     %Set current character position to this position.
     If the SE option is used, then set scan direction
     right to left.
     If the SF option is used, then set scan direction
     left to right.
     Otherwise the scan direction is unchanged.%            2|r7a

  '<   /
     %Set scan direction to the left.%                      2|r7b

  '>   /
     %Set scan direction to the right.%                     2|r7c

  '↑ textpointer   /
     %Store current scan position into the textpointer%     2|r7d

  '← [.NUM] textpointer   /
     %Back up the textpointer by the specified number of
     characters.  Default value for number is one.
     Backup is in the opposite direction of the
     current scan direction.%                               2|r7e

  "FS" .ID   /
     %Set this flag to true
     the flag may be a global or a local variable%          2|r7f

  "FR" .ID   /
     %Reset this flag to false%                             2|r7g
```

```
'+ (.ID / link)   /
   %Call this procedure then continue%                          2|r7h

"TRUE"
   %Has no effect%;                                             2|r7i

element =  %can be true or false%                               2|r8

.SR /
   %String constant%                                            2|r8a

'* stringname '* /
   %String variable%                                            2|r8b

char /
   %Character constant%                                         2|r8c

charclass /
   %Look for a character of this class%                         2|r8d

"FT" .ID /
   %Test this flag%                                             2|r8e

'( union ') /
   %Look for an occurence of the specified pattern%             2|r8f

'- element /
   %False if the specified element occurs next%                 2|r8g

"INITIALS" ('# / '=) .UID /
   %Initials of user who created the statement
   are tested by this construct.
   Undefined for a string.%                                     2|r8h

"SINCE" date /
   %True if statement was created since
   the specified date.
   Undefined for a string.%                                     2|r8i

"BEFORE" date /
   %True if statement was created before
   the specified date.
   Undefined for a string.%                                     2|r8j

"BETWEEN" pos pos '( union ') /
   %Search limited to between the positions.
```

```
      Scan character position is set to first position
      before the pattern is tested.%                                  2|r8k

   '? (.ID / link) /
      %The procedure is called, then the global
      variable "flag" is tested%                                      2|r8l

   '[ union '] /
      %True if the pattern can be found anywhere
      in the remainder of the statement.
      First searches from current position.
      If that fails then increments the position
      and tries again until the statement is exhausted.%              2|r8m

   .NUM element /
      %Find the specified number of occurences
      of the element%                                                 2|r8n

   bndnum
      %bounded number of occurences%;                                 2|r8o

bndnum = lowbnd '$ uprbnd element;                                    2|r9

   lowbnd = [.NUM];                                                   2|r9a

      Default lower bound is zero.  Must find at least this
      many occurences.                                                2|r9a|

   uprbnd = [.NUM];                                                   2|r9b

      Default upper bound is some very big number.  Will
      look for no more than this many occurences.                     2|r9b|
```

The test for the element is made until it fails or the upper bound is reached. If the upper bound is reached, then the "bndnum" is true. If the test finally fails, then the "bndnum" is true if the number of times the element was found lies within the bounds. 2|r|0

```
date =                                                                2|r||
   '( year '/ month '/ day  [hourminute [': second] ] ');             2|r||a
```

day time

The year, month, day, hourminute, and second are given as numbers. Note that the hour and minute are given as a single number (hourminute in the rule) and are based on a

24 hour day. The year, month, and day are required; the others are optional with defaults of zero. 2|r|2

```
stringconstruction =                                              2|s

  ("ST" (pos / substr) /                                          2|s|

     '* stringname '* ['[ exp "TO" exp']] ) '← stlist;            2|s|a
```

The string to which pos or stringname refers is replaced by the string specified to the right of the arrow. A substring is replaced if a substr or an indexed stringname is specified. 2|s2

```
     ST p| p2 ← string;
     is equivalent to
     ST p| ← SF(p|) p|, string, p2 SE(p2);                        2|s2a

     *str*[lower TO upper] ← string;
     is equivalent to
     *str* ← *str*[| TO lower-|], string, *str*[upper+| TO
     str.L];                                                      2|s2b
```

The new string or substring is specified as a concatenation of string primaries, with the primaries separated by commas. 2|s3

```
   stlist = stprim $(', stprim);                                  2|s4

   stprim =                                                       2|s5

     "NULL" /
        %represents the zero length string%                       2|s5a

     .SR /
        %for string constants%                                    2|s5b

     substr /
        %substring%                                               2|s5c

     '+ substr /
        %substring capitalized%                                   2|s5d

     '- substr /
        %substring in lower case%                                 2|s5e
```

```
'$ substr /
   %substring without markers%                                    2|s5f

'* stringname '* /
   %for string variables%                                         2|s5g

'* stringname '* '[ exp '] /
   %for character variables%                                      2|s5h

'* stringname '* '[ exp "TO" exp '] /
   %substring by indices%                                         2|s5i

exp /
   %value of expression taken as a character%                     2|s5j

"STRING" '( exp ['', exp] ');
   %gives a string which represents the value of the
expression as a signed decimal number.  If the second
expression is present, a number of that base is produced
instead of a decimal number.%                                     2|s5k

substr = pos pos;                                                 2|s6

   This is the substring bounded by the two positions.            2|s6a

If it is preceded by a dollar sign ($), then the substring
is copied without moving any associated markers to the new
position.                                                         2|s7

A construct of the form *str*[i TO j] refers to the
substring starting with the ith character in the string up
and including the jth character.  Thus *str*[i TO i+10] is
the eleven character substring starting with the ith
character of str, and *str*[i TO str.L] is the string str
with the first i-1 characters deleted.                            2|s8

Example:                                                          2|s9

   Let a "word" be defined as an arbitrary number of letter
   digits.  The two statements in this example delete the
   word pointed to by the text pointer "t", and if there is
   a space on the right of the word, it is also deleted.
   Otherwise, if there is space on the left of the word it
   is deleted.                                                    2|s9a

   The text pointers x and y are used to delimit the left
   and right respectively of the string to be deleted.            2|s9b
```

LD is true if the character is a letter or a digit, and SP is true if the character is a space. 2|s9c

```
FIND t < $LD ↑x t > $LD (SP ↑y / ↑y x < SP ↑x / TRUE);
ST x y ← NULL;
```
2|s9d

The reader should work through this example until it is clear that it really behaves as advertized. 2|s9e

The following several statement forms are related to controlling the feedback given to the NLS user and the defintion of "states" within the command language of NLS. 2|t

```
inputstat = "INPUT" inalternatives;
```
2|u

The input statement construction is used to program command interaction in Display NLS. It results in a sequence of calls on routines which implement basic operand interaction such as bug selection on the display screen and text input. 2|u|

Alternatives within an NLS command group (e.g., the option to select or type the second operand in the REPLACE command) are automatically handled by the input statement construction. In addition, backup within the command specification as a result of a backspace typed by the user is also handled automatically. 2|u2

Use of the input statement construction results in a greater consistency in the command language as well as a great simplification in the programming of new commands. 2|u3

```
inalternatives = insequences $('/ insequences);
```
2|v

```
insequence =
    $initem ('( inalternatives ') / $(stat [';]) );
```
2|w

```
initem =
```
2|x

```
("BUG" / "STID" / "LEVADJ" / "TEXT"/
     "NAME" / "WORD" / "NUMBER" / "STRING") fwlhs /
char;
```
2|x|

The left-hand-sides following BUG or STID should specify a TEXT POINTER. Those following LEVADJ, TEXT, etc., should specify a string. 2|x2

Explanation: 2|x3

BUG b| means read a selection made with the cursor and store the resulting TEXT POINTER into b|. 2|x3a

STID b| means input a statement selection and store the TEXT POINTER into b|. A statement selection is either a BUG or a SP followed by a statement name or number. 2|x3b

LEVADJ str inputs a sequence of level adjust characters and stores them into the string str. Level adjust characters are defined to be an arbitrary number of u's or d's optionally terminated by a space. 2|x3c

TEXT str inputs a string of characters and echoes them in the text area of the display. The string is stored in str and is defined to be an arbitrary number of characters up to, but not including, a command accept or center dot. 2|x3d

NAME str inputs a string of characters which are forced upper case, stored in str, and echoed in the name register of the display. Alternatively the name may be selected by a BUG in which case the selected word is forced upper case, stored, and displayed. 2|x3e

WORD str like NAME except does not force upper case. 2|x3f

NUMBER str like NAME except inputs a number either typed or specified by BUG selection. 2|x3g

STRING str like TEXT except echoes in the NAME register. 2|x3h

char succeeds if the specified character is input. 2|x3i

Example: 2|x4

The INPUT statement for the Replace Text command is 2|x4a

```
INPUT BUG b| BUG b2                                    2|x4a|
   (BUG b3 BUG b4 CA rpf ← TRUE; /                     2|x4a|a
   TEXT lit CA rpf ← FALSE) ;                          2|x4a|b
```

The flag "rpf" is used to show which alternative was taken. 2|x4a2

```
feedbackline = "DSP" '( $dsp ');                                    2|y

  The command feedback line the the display is controlled by
  means of this statement.                                          2|y|

   dsp =                                                            2|y2

     '← /
        %move feedback arrow to far left%                           2|y2a

     '↑ /
        %put feedback arrow under start of next word%               2|y2b

     "↑ON" /
        %turn feedback arrow on
        (this is done automatically if specify
        position for the arrow)%                                    2|y2c

     "↑OFF" /
        %turn feedback arrow off%                                   2|y2d

     '? /
        %turn question mark on%                                     2|y2e

     "?OFF" /
        %turn question mark off%                                    2|y2f

     "..." dword /
        %the last word currently in the feedback line
        is replaced by this one%                                    2|y2g

     '< dword /
        %clear feedback line and put this word
        in the first position%                                      2|y2h

     dword
        %put this word in the next position%;                       2|y2i

     Examples:                                                      2|y2j

        DSP(<Insert ↑ *es*);                                        2|y2j|

        DSP(... Character);                                         2|y2j2

   dword = .ID / .UID / .SR / '* stringname '*;                     2|y3

     If an identifier or string is used, then that text is
```

displayed. If a stringname is used, then the contents of that string is displayed. 2|y3a

```
group = "GROUP" '← .ID;
```
2|z

Sets the command group. 2|z|

```
state = "STATE" '← .ID ', .ID;
```
2|a*

Sets the state to this location and sets the group to the second identifer. The first identifier may be used as a label for this location. Control is transferred here by the GOTO STATE command. 2|a*|

```
entity = "ENTITY" '← .CHR ', (.ID / .UID);
```
2|aa

The character is made the entity character for this group and the identifier is made the entity string for this group. 2|aa|

EXPRESSION FORMS

22

Expressions 22a

exp = 22a|

```
"IF" exp "THEN" exp "ELSE" exp /
"CASE" exp "OF"
   $(binrel $(', binrel) ': exp ';)
  "ENDCASE" exp /
disjunct;
```
22a|a

An expression involving logical operators may be used in the place of an arithmetic expression. It has the value | if true and the value 0 if false. 22a2

Example: 22a3

```
flag ← a > b OR x = y
```
22a3a

is equivalent to 22a3b

```
   flag ← IF a > b OR x = y THEN 1 ELSE 0.                    22a3c
```

Likewise, an arithmetic expression may be used where a logical expression is expected. A zero value represents false, and a nonzero value represents true. 22a4

Example: 22a5

```
   IF p() THEN a ELSE b                                       22a5a
```

is equivalent to 22a5b

```
   IF p() NOT= 0 THEN a ELSE b.                               22a5c
```

Logical operators 22b

```
   disjunct = conjunct $("OR" conjunct);                      22b1
```

True if either conjunct is. 22b1a

```
   conjunct = negation $("AND" negation);                     22b2
```

True if both negations are. 22b2a

```
   negation = "NOT" negation / relation;                      22b3
   relation =                                                 22b4
     "SKIP" (lhs args / builtin) /                            22b4a
     string ["NOT"]                                           22b4b
        ('= / '# / "<=" / ">=" / '< / '>) string /            22b4b1
     "FIND" union /                                           22b4c
     "POS" posrel /                                           22b4d
     sum [binrel];                                            22b4e
```

The expression "FIND union" is true if the pattern defined by the union is found. 22b4f

The "SKIP" relation is true if the construct following it results in skipping an instruction when it is evaluated. The construct may be either a call to

```
    procedure or a builtin, such as a "jump to system", that
    may do a SKIP RETURN.                                          22b4g

  string = '* stringname '* / .SR;                                 22b5

    It is possible to compare variable or literal strings.         22b5a

  posrel =                                                         22b6

    pos ["NOT"] ('= / '# / ">=" / "<=" / '> / '<) pos;             22b6a

    This may be used to compare two text pointers.                 22b6b

    If the pointers refer to different statements then all
    relations between them are false expect "not equal"
    which is written '# or "NOT" '=.  If the pointers refer
    to the same statement, then the truth of the relation is
    decided on the basis of their location within the
    statement with the convention that a pointer closer to
    the front of the statement is "less than" a pointer
    closer to the end.                                             22b6c

  binrel = ["NOT"] (                                               22b7

    (">=" / "<=" / '> / '<) sum /                                  22b7a

    ('= / '#) (charclass / sum) /                                  22b7b

    ("IN" / "OUT") intrel;                                         22b7c

  charclass =                                                      22b8

    "CH" /
       %any character%                                             22b8a

    "ULD" /
       %uppercase letter or digit%                                 22b8b

    "LLD" /
       %lowercase letter or digit%                                 22b8c

    "LD" /
       %lowercase or uppercase letter or digit%                    22b8d

    "NLD" /
       %not a letter or digit%                                     22b8e
```

```
    "UL" /
       %uppercase letter%                                    22b8f

    "LL" /
       %lowercase letter%                                    22b8g

    "L" /
       %lowercase or uppercase letter%                       22b8h

    "D" /
       %digit%                                               22b8i

    "PT" /
       %printing character%                                  22b8j

    "NP"
       %nonprinting character%;                              22b8k
```

This provides a simple way to test the common classes of characters. 22b81

Example: 22b8m

```
       char = LD                                             22b8m1
```

is true if the variable "char" contains a value which is a letter or a digit. 22b8m2

```
intrel = ('( / '[) sum ', sum ('] / '));                     22b9
```

This provides for the common operation of testing whether the value of some expression lies within a particular interval. Each side of the interval may be "open" or "closed". In other words the value which determines the boundary of the interval may be included within the interval (by using square brackets) or excluded (by using parentheses). The IN relation means that the value is in the interval; OUT is equivalent to NOT IN. 22b9a

Example: 22b9b

```
       x IN [1,100)                                          22b9b1
```

is the same as 22b9b2

```
       x >= 1 AND x < 100.                                   22b9b3
```

Arithmetic operators 22c

```
sum = prod $('+ prod / '- prod);                                    22c1
```

The plus (+) and minus (-) represent addition and subtraction. 22c1a

```
prod = bitor $('* bitor / '/ bitor / "MOD" bitor);                  22c2
```

The star (*) and slash (/) represent multiplication and division. 22c2a

The form a MOD b gives the remainder of a / b. 22c2b

```
bitor = bitand $(".V" bitand / ".X" bitand);                        22c3
```

The form a .V b gives the logical or of a and b. 22c3a

The form a .X b gives the exclusive or of a and b. 22c3b

```
bitand = factor $(".A" factor);                                     22c4
```

The form a .A b gives the logical and of a and b. 22c4a

```
factor = '- factor / '( exp ') qualifiers / prim;                   22c5
```

Note that it is possible to extract a field from the value of a parenthesized expression. 22c6

Example: 22c7

The assignment statement 22c7a

x ← (b + c).f 22c7b

stores the f field from b + c into the variable x. 22c7c

Primitives 22d

```
prim =                                                              22d1

  lhs [args / '← exp / ":=" exp] /
  rhs /
  ("MIN" / "MAX") '( exp $(', exp) ') /
  "MKFD" '( exp ', exp ', exp ') /
  "READC" ['( exp ')] /"VALUE" '( exp [', exp] ') /
  "SIGNAL" /
```

"MESSAGE" /
"CCPOS" ; 22d1a

When a procedure call is used as a primitive, the value is that of the leftmost result returned by the procedure. 22d2

Two forms of assignments can be used as primitives. 22d3

The form a ← b has the effect of storing b into a and has the value of b as its value. 22d3a

The form a := b has the effect of storing b into a and has the old value of a as its value. (An exchange with memory instruction is done rather than a store). 22d3b

Example: 22d3c

The statement 22d3c1

c ← b := c 22d3c2

exchanges the contents of b and c. 22d3c3

The SIGNAL and MESSAGE primitives were discussed above in the ONSIGNAL statement. 22d4

The first expression in the VALUE primitive evaluates the address of a string. The result is a binary number corresponding to that string as a signed decimal number. If the second expression is present, the number is assumed to be of that base rather than |O. 22d5

```
rhs = '$ (.ID / .SR) / literal;
```
22d6

A string or an identifier preceded by a dollar sign ($) represents the address of that string or identifier. 22d6a

```
literal = .NUM / "TRUE" / "FALSE" / char;
```
22d7

Numbers come in several flavors. A sequence of digits alone or followed by a D is interpreted as base ten. If followed by a B then it is interpreted as base eight. A scale factor may be given after the B for octal numbers or after a D for decimal numbers. The scale factor is equivalent to adding that many zeros to the original number. 22d8

Examples: 22d8a

64 = 100B = B2 22d8a1

144B = 100 = D2 22d8a2

A sequence of digits followed by an M is a mask. The number indicates the number of bits to be turned on in the mask. An octal scale factor may also be present to the right of the M. If he scale factor is absent, the mask is right justified. Thus 18M is an 18 bit mask on the right half of a 36-bit PDP-10 word; 18M6 is an 18 bit mask on the left half word. 22d8b

The words TRUE and FALSE are equivalent to the numbers 1 and 0 respectively. 22d9

The following provide synonyms for commonly used characters. The effect is the same as if the number internally representing the character had been used. 22d10

```
 char =                                                    22d11

   .SR1 /
      %single character preceded by an apostrophe%        22d11a

   "ENDCHR" /
      %endcharacter as returned by READC%                 22d11b

   "CD" /
      %command delete%                                    22d11c

   ("BUG" / "CA") /
      %command accept%                                    22d11d

   "SP" /
      %space%                                             22d11e

   "EOL" /
      %Tenex's version of CR LF%                          22d11f

   "ALT" /
      %Tenex's version of altmod or escape (=33B)%        22d11g

   "CR" /
      %carriage return%                                   22d11h
```

```
"LF" /
   %line feed%                                              22d||i

"TAB" /
   %tab%                                                    22d||j

"BC" /
   %backspace character%                                    22d||k

"BW" /
   %backspace word%                                         22d||l

"C."
   %center dot%;                                            22d||m
```

The builtin functions MIN and MAX have the obvious meaning. 22d|2

MKFD makes a field descriptor with the three expressions giving the position, size, and address respectively. The address may be 23 bits so as to include indirection or indexing. 22d|3

The primitive READC is a special construct for reading characters from NLS statements or strings. 22d|4

The optional parenthesized expression after "READC" gives the address of a work area that determines which character is to be read next. 22d|4a

If the expression is not given, then a character is read from the current character position and in the scan direction as set by the last CCPOS statement or string analysis. 22d|4b

Attempts to read off the end of a string in either direction result in a special "endcharacter" being returned and the character position is not moved. This endcharacter is included in the set of characters for which system mneumonics are provided and may be referenced by the identifier "ENDCHR". 22d|4c

Example: 22d|4d

to sequentially process the characters of a string 22d|4d|

```
CCPOS *str*;
UNTIL (char ← READC) = ENDCHR DO process(char).      22d|4d2
```

(Note: READC may also be used as a statement if it is desired to read and simply discard a character). 22d|4e

When used as a primitive, CCPOS has as its value the index of the character to the right of the current character position. 22d|5

Examples: 22d|5a

If str = "glarp", then after CCPOS *str*, the value of CCPOS is 1 and after CCPOS SE(*str*) the value of CCPOS is 6 (one greater than the length of the string). 22d|5a1

To sequentially process the first n characters of a string (assumed to have at least n characters) 22d|5a2

```
CCPOS *str*;
UNTIL CCPOS > n DO process(READC).
```
22d|5a3

DECLARATION FORMS

23

Declare 23a

```
declare =  (decl/ext/equ/regdec/record/pgdec/refd) ';;
```
23a1

```
decl = "DECLARE" [/"EXTERNAL"/]
```
23a2

```
  (field / string / tp / stores / items);
```
23a2a

If the EXTERNAL modifier is used, then the following names may be referenced from other files. 23a3

```
field = "FIELD" fdef $(', fdef);
```
23a4

```
   fdef = .ID '= '/ address ', size ': position '/);
```
23a4a

The "address" is taken from the syntax for builtin's. 23a4b

```
   size = .ID / .NUM;
```
23a4c

This is the size in bits of the field. If an identifier

is used, it must already have a value as the result of a SET statement. 23a4d

```
 position = .ID / .NUM;                                    23a4e
```

This is the position of the field in the word given as a displacement in bits from the right. 23a4f

```
string = "STRING" astr $(', astr);                         23a5
```

```
 astr = .ID ('[ .NUM '] / '= .SR);                         23a5a
```

Strings defined by this means may be used in those constructs, such as string construction, that call for a "stringname". 23a5b

A declaration such as 23a5c

```
   longstring [200]                                        23a5c|
```

provides for a string with a maximum length given by the number. The current length of the string is set to zero. 23a5d

A declaration such as 23a5e

```
   literalstring = "a silly example"                       23a5e|
```

serves to initialize the string. Such an initialized string may later be assigned another string whose length is no greater than the initial one. 23a5f

In the future the string system will be extended so that the maximum length of strings need not be specified. Instead the system will allocate string storage as required. 23a5g

```
tp = "TEXT" "POINTER" idlist;                               23a6
```

```
 idlist = .ID $(', .ID);                                   23a6a
```

The listed identifiers may be used to hold text pointers for use in string/statement analysis and construction. 23a6b

The text pointer takes more than one word of storage, and therefore references should be passed rather than values in procedure calls and the like. 23a6c

```
stores = ("STACK" / "RING") stckd $(', stckd);                    23a7

   stckd = .ID '[ .NUM [', .NUM] ']';                             23a7a

   Allocates storage for a pointer and stack or ring buffer
   that can hold the number of records specified by the
   first number.  The second number gives the size in words
   of each record.  The default size is one word.                 23a7b

   See the "stacknring" commands PUSH, POP, and RESET.            23a7c

items = item $(', item);                                          23a8

   item = .ID [dimension / value];                                23a8a

   dimension = '[ (.ID /.NUM) ']';                                23a8b

      An array of the specified dimension is allocated.  If
      an identifier is used to specify the dimension, it
      must have already been defined thru a SET statement.        23a8b|

   value = '= ( '( icon $(', icon) ') / icon );                   23a8c

      The specified values are allocated.                         23a8c|

   icon = .SR / .ID / ['-] literal;                               23a8d

      When a string is listed as a component, it is
      produced using as many words as are needed to hold
      the characters packer left justified, four characters
      to a word.  An identifier used as a component results
      in its address being produced as the contents of the
      word.  If the identifier was defined with a SET
      statement, then the value to which it was set is
      produced.  A literal component results in a word
      containing the value of that literal.                       23a8d|

External                                                          23b

   ext = "EXTERNAL" idlist;                                       23b|

   The identifiers listed may be referred to from other
   programs.  Procedure names are automatically set external.     23b2

Set                                                               23c

   equ = "SET" equ| $(', equ|);                                   23c|
```

```
equ1 = (.UID / .ID) '= (.ID / ['-] literal);                    23c2
```

The identifier on the left is defined to have the value specified on the right. If an identifier is used on the right, it must have been already given a value thru a previous set statement. 23c3

This statement may be used to define opcodes for use in builtin's. 23c4

Register 23d

```
regdec = "REGISTER" regdef $(', regdef);                        23d1

regdef = .ID '= .NUM;                                           23d2
```

The number specifies the accumulator to be used to hold the variable. 23d3

Record 23e

```
record = '( .ID ') "RECORD" recdef $(', recdef);                23e1

recdef = .ID '[ .NUM '];                                        23e2
```

The number is the size in bits of the field. 23e3

Page 23f

```
pgdec = "PAGE" [.NUM];                                          23f1
```

This simply moves the location counter to the start of the next "page", or 512 word block. If the program is to loaded starting at a location other than the first word of a page, the displacement into the page where loading will begin must be given following PAGE. 23f2

Reference 23g

```
refd = "REF" idlist;                                            23g1
```

There are two subclasses of variables; those which have been declared to be references by the "REF" statement, and those which have not. 23g2

If a variable, v say, has been declared to be a

reference, then uses of v will be equivalent to uses of *[v]* if v had not been declared to be a reference. 23g2a

In order to allow the programmer to override this reference mode, a reference name preceded by an ampersand (&) is taken to mean the variable itself. 23g2b

This allows a reference variable to be initialized or modified while retaining the ability to make references through it without explicitly using square brackets. 23g2c

Note that the REF statement does not declare any storage, but simply sets an attribute of variables. 23g3

Local variables may be declared to be references by a REF statement following the declaration of LOCAL variables in the procedure. 23g4

Examples: 23g5

```
(p) PROCEDURE(ptr|);
   LOCAL b|, ptr2;  REF ptr|;
   b| ← ptr|;     %stores what ptr| references into b|%
   ptr2 ← &ptr|;     %stores ptr| into ptr2%
   RETURN END.
```
23g5a

is equivalent to 23g5b

```
(p) PROCEDURE(ptr|);
   LOCAL b|, ptr2;
   b| ← [ptr|];     %stores what ptr| references into b|%
   ptr2 ← ptr|;     %stores ptr| into ptr2%
   RETURN END.
```
23g5c

A more realistic example shows the combined use of the "ref" and "unref" forms of a variable in a procedure that searches a table for an entry whose fields satisfy certain constraints. 23g5d

```
(search) PROCEDURE (room);
```
23g5e

```
   %Return the index of the first entry in the array
   "table" whose "exists" field is nonzero and whose
   "size" field is as large as the "room" parameter.  If
   there is no such entry then return -|.  The variable
   "length" specifies the number of entries in "table".%
```
23g5e|

L10 Documentation

```
        LOCAL t, index;                                          23g5e2
        REF t;                                                   23g5e3
        &t ← $table; %set up the pointer%                        23g5e4
        FOR index ← 0 UP UNTIL = length DO                       23g5e5
           IF t.exists AND t.size >= room THEN RETURN (index)
                                                                 23g5e5a
           ELSE BUMP &t;                                         23g5e5b
        RETURN (-1) END.                                         23g5e6
```

PROGRAM

```
                                                                 2h
 program = $parts "FINISH";                                      2ha
 parts = procedure / declare;                                    2hb
 procedure = '( .ID ') "PROCEDURE" ['( idlist ')] '; body;       2hc
 body =                                                          2hd
   $("LOCAL" locd '; / "REF" idlist ';)
   labeled $(';  labeled) "END." ;                               2hd1
locd =                                                           2he
   "STRING" lstr $(', lstr) /
   "TEXT""POINTER" idlist /
   loco $(', loco);                                              2he1
lstr = .ID '/ NUM '/;
  %NUM gives the maximum length of the local string being
declared%                                                        2hf
loco = .ID ['[ .NUM ']];
  %Local declaration of an array of NUM words or a simple
variable%                                                        2hg
Procedure                                                        2hh
```

A procedure may have an arbitrary number of local variables, which include formal parameters, variables, strings, text pointers and arrays declared by the LOCAL statement. 24h1

Note that nonlocal declarations are outside of the procedure body. 24h2

Future versions of L|O will include more general control structures based on processes, ports, and messages. 24i

CODE PRODUCED BY THE COMPILER

25

The following is provided as an admittedly contrived example of the code produced by the compiler. 25a

For the sequence of statements 25b

```
x ← x + a;
y ← 6 - y;
IF x < 0 THEN
   BEGIN
   z ← 1;
   [p] ← x;
   BUMP p;
   END
ELSE z ← -z;
[p] ← x;
a ← z;
```

25c

the compiler produces the following instructions: 25d

```
 MOVE 17,a          % r17 ← a %
 ADDB 17,x          % x ← r17 ← r17 + x %
 MOVEI 16,6         % r16 ← 6 %
 SUBB 16,y          % y ← r16 ← r16 - y %
 JUMPGE 17,.+6      % if r17 (holding x) >= 0 then goto .+6 %
 MOVEI 15,1         % r15 ← 1 %
 MOVEM 15,z         % z ← r15 %
 MOVEM 17,@p        % [p] ← r17 (which still holds x) %
 AOS 14,p           % p ← r14 ← p + 1 %
 JRST .+2           % branch over false part of IF statement %
```

```
MOVNS |5,z        % z ← r|5 ← -z (use same register for z) %
MOVEM |7,@p       % [p] ← r|7 (which still holds x) %
MOVEM |5,a        % a ← r|5 (which holds z since was loaded
                  in both parts of the IF statement) %          25e
```

The compiler remembers simple variables which are in accumulators and carries this information forward through the various paths of the IF statement. Notice that in both the true and false parts of the IF, the variable z is loaded into the same register so that in the following statements z need not be reloaded. 25f

L10 Documentation W H Paxton 12/05/70

<JOURNAL>7052.NLS;1, 29-MAY-71 11:23 WSD ; Title: Author(s): William H. Paxton/WHP; Keywords: L10 Programming Language; Clerk: WSD;
Origin: <PAXTON>DOCL10.NLS;5, 29-MAY-71 10:20 WHP ;
.DIR=1;.HED=".MCH=72; .GCR;.HJH=2;WHP 29-MAY-71 11:20
7052.MCH=65;
.HJH=1;
L10 Documentation W H Paxton 12/05/70";
** .MCH=65; .SNB=0; .SCR=2; .RTJ=0; .PNO=0; .COD['{]=175B;

WHP 29-MAY-71 12:12 7053

A System for Modular Programming

This is an early (12/70) proposal for a modular programming system.

A SYSTEM FOR MODULAR PROGRAMMING 12/05/70 1

The use of terms such as job, process, and module in the following may not correspond with "common" usage. 1a

A job is defined as a dynamic collection of cooperating processes sharing the same address space. A process is the dynamic counterpart of a module. A module is a collection of procedures and data structures. 1b

OBJECTIVES 1c

It must be possible to specify at runtime the set of processes that make up the job, and it must be possible to dynamically modify this set. Jobs involving up to 100 processes should not be unreasonable. 1c1

It must be possible to dynamically allocate the address space. In other words, modules must be logically as well as physically relocatable at runtime. The limit on the totality of modules potentially available to a user should be set be the capabilities of the file system not the size of the address space. 1c2

The interface to a process should be easy to describe. A process should be usable in any configuration that satisfies the restriants of the interface. 1c3

It should be possible to have several processes in a job which correspond to a single module. 1c4

There should be easy-to-use operations for passing messages and control between processes. 1c5

Error handling mechanisms should be provided by the system. A process should be able to catch errors made by another process, have an opportunity to correct the problem, and allow the other process to be restarted. 1c6

It should be possible to modify a module without having to update modules which might make use of it as long as its interface remains unchanged. 1c7

One process should be able to use another without knowing its memory requirements or what other processes it may use. 1c8

Code segments of modules must be sharable by several processes and by several jobs. 1c9

The system must contribute to the development of other high level software augmentation tools such as incremental compilation and source language debugging from NLS. 1c10

MODULES AND PROCESSES 1d

There are two classes of modules: MESSAGE modules and UNIT modules. 1d1

MESSAGE modules contain: 1d2

A code segment which contains procedures which may be activated by the arrival of messages or by calls from within the module. 1d2a

A data segment containing the definitions of data structures which may be accessed within the module and the definitions of ports into the module over which messages may be sent and received. 1d2b

A linkage segment containing the link data neccessary to make symbolic references from this module to others. Dynamic linking is based on the Multics system and is discussed in detail below. 1d2c

The term process is used here to denote the dynamic object corresponding to a module. Thus a module provides a static definition of some process. 1d3

There may be more than one process corresponding to a single message module. Each of these processes has a separate copy of the data segment of the module, however they all share the same linkage and code segments. 1d4

ports in a message process may be connected to ports in other message processes. Message processes communicate and transfer control by sending messages over these ports (hence the name message process). The commands for establishing connections between message processes and for sending and receiving messages are given below. 1d5

UNIT modules contain code, data, and linkage segments much the same as message modules. However, unit modules have no ports and thus may not send or receive messages. The unit module is intended to serve as a repository of procedures used by several modules. 1d6

By convention there is only one process corresponding to a

particular unit module (hence the name unit module). This allows procedures in the module to be called by simply giving the module name and the procedure name. Message processes must be refered to by a process number assigned at runtime since there may be more than one process corresponding to a module. 1d7

The unit module roughly corresponds to the segment of Multics. 1d8

When modules are needed at runtime, the system takes care of allocating memory for them and loading them. Modules may be loaded anywhere in the address space, thus it is possible to dynamically allocate memory. 1d9

A module may be referred to by its symbolic name (i.e. the string of characters making up the name) when it is requested. Likewise, variables or procedures within the modules may be referenced from other modules by their symbolic names. After the first use of these symbolic names information in the linkage segment is modified so that later references may bypass the symbolic names and go directly to the desired location. This operation is called dynamic linking and allows modules to be independently modified and updated. This gives a degree of flexibility in the development of a large system of modules that is unattainable in more conventional systems. 1d10

MODULE FILE 1e

The source language program for a module is kept in a normal NLS file. The other data concerning a module is kept on a separate file. (??? What about keeping it all together ???) This file contains: 1e1

(1) header 1e2

This contains information describing the structure of this file, such as the size and location in the file of the various data segments. 1e2a

(2) a code segment 1e3

pure procedures. sharable at runtime. relocatable in address space since all intrasegment references are made relative to a code segment pointer. 1e3a

(3) a data segment 1e4

data structures for the module. ports for message modules. each process has a copy of the data segment for the module. relocatable in address space since all references to items in the data segment are made relative to a data segment pointer. 1e4a

(4) a linkage segment 1e5

link data for all the external references made from this module. modified as used at runtime. may be shared by all processes based on this module. references to this segment are made relative to a linkage segment pointer. 1e5a

(5) an external symbol list 1e6

list of the external symbols in this module. used to resolve references to names in this module made at runtime. for each name in the module which may be refered to by other modules has the symbolic name of the variable and its location as segment and displacement within segment. 1e6a

(6) descriptive information. 1e7

for use by the incremental compilation system. contains whatever info is needed to do incremental changes and source language debugging with the module. this will include at least a complete symbol table with all local variables, a list of intrasegment references, and a map from source to location in the various segments and vice versa. 1e7a

PROCESSES 1f

For each process in the job, there is 1f1

a code segment (which may be shared), 1f1a

a data segment, 1f1b

a linkage segment (which may be shared), 1f1c

a chain of activation records (or "frames") which is analogous to a call stack for the process 1f1d

a process description record which holds the segment numbers of the various segments making up the process, a pointer to the start of the activation chain, the

current status of the process, and other information about the process. 1f1e

Whenever the process is active the code pointer CP points to the code segment of the process, the data pointer DP points to the data segment, the link pointer LP points to the linkage segment, and the frame pointer FP points to the activation record of the current procedure. These pointers are held in index registers so that the process may access its own information by simply using the indexing abilities of the machine. 1f2

When control is passed to another process, the segment pointers are automatically changed. This is facilitated by saving the process number of the calling process in each activation record. Then when the procedure returns it is easy to check if it is returning to a new process. If that is the case then the process record specifies how the pointers should be loaded. 1f3

The use of these pointers allows dynamic allocation of the address space. This is important when there are a great number of modules available for use, the totality of which exceeds the size of the address space. It would be impossible to determine ahead of time where a particular module should be loaded, so this decision is instead put off until the module is actually needed. The exact location will depend on the set of processes in use by the job at the time the module is required. Thus the module may be loaded in one location in one job and in a different location in another. The use of segment pointers allows the module to be shared by several jobs concurrently even though it is placed it different locations in each address space. 1f4

OPERATIONS 1g

The following is a preliminary specification of the basic operations that will be provided in the modular programming system. Initially the programs to implement these operations will be in the address space of the job. Eventually it may be desirable to make the operations part of the time sharing system itself. 1g1

(1) LOAD module 1g2

The module name is given symbolically. If the module is already loaded then this operation simply returns.

Otherwise the module file is opened and the header read. Based on the information in the header, memory is allocated for the segments of the module and they are mapped into the address space. lg2a

The system must maintain a list of modules currently loaded as well as information describing the state of memory allocation. lg2b

(2) UNLOAD module lg3

The module name is given symbolically. The named module is removed from memory. The space allocated for it may be reclaimed. Links to segments of the module must be restored to completely symbolic form so that they will be recomputed if the module is loaded again later. Any processes corresponding to this module are killed. lg3a

(3) RELEASE module lg4

This command informs the system that the named module is a candidate for unloading if room is needed. If the system runs out of address space it will look for a released module to unload. lg4a

After a module has been used and is no longer needed it should be released rather than immediately unloaded. Then if the module is needed again before the system has had to unload it, the module will still be there with all links to and from it intact. (Will have to bring in fresh copy of data segment or be able to suitably initialize it.) lg4b

(4) USE module1 FOR module2 lg5

All references to module2 go to module1 instead. this operation does an UNLOAD module2 first to reset any existing references to module2. This operation is intended primarily for use in debugging new versions of modules. after the experimental module1 has been checked out then it may replace module2 for general use. may also use this operation if have special versions of modules that have been tailored to certain needs. for example a personalized command module. lg5a

(5) JOIN modules lg6

It will often be the case that a set of modules are used

together. The JOIN command makes it possible to exploit this fact by "pre-linking" the cross references and using the set as a unit. lg6a

The definition of a link (given below) is such that the displacement and segment type (code or data) may have a value even if the segment number does not. Thus when modules are joined, for all the cross references between modules the displacement and the segment type are filled in while the segment number is left undefined. The segment number is then filled in by the usual dynamic linking sequence. lg6b

When any one of a set of joined modules is loaded they all are. Because unloading resets all links to a module to a completely symbolic form, it is possible to replace one of the modules without effecting the modules to which it is joined. This means that modules can be joined without sacrificing any flexibility. Any one of a set of joined modules may be replaced by an experimental version with only the links to and from that particular module being converted back to symbolic form. Then when the new version is debugged, it may be joined to the set by simply pre-linking references between it and other modules. lg6c

This amounts to a facility for creating "macro-modules" with the ability to easily modify and replace components. lg6d

(6) NEW module AS processname lg7

The module must be a message module. The module is loaded and a copy of the data segment is made. A process record for the new process is created with the status set to unstarted. The process number is stored in the specified processname for later references to the process. The processname may be any variable in the process executing this operation. lg7a

(7) KILL processname lg8

The named process is no longer needed. Its data segment may be deleted and the associated storage reclaimed. Any links to the data segment must be reset to symbolic form. Any activation records associated with the process may be reclaimed. Finally the process record itself may be freed. lg8a

If this was the only process associated with the module, then the module is released. lg8b

(8) dynamic linking lg9

This operation involves the conversion of symbolic references to a variable in another process (or unit module) to a segment number and displacement within that segment. This operation may involve loading the module if it is not yet a part of this program. lg9a

Dynamic linking is called for implicitly through references to names in other processes. There are three categories of interprocess references and associated categories of dynamic linking. lg9b

(1) call on a procedure in a unit module lg9c

The call references a record in the linkage segment on the process containing the strings naming the module and the procedure to be called. lg9c1

To translate these into a segment number and displacement it is necessary to first make sure the module is loaded and a process corresponding to the module created. Then the external symbol list for the module is loaded and the procedure name is found in this list. This provides the location of the procedure within the code segment of the module. The segment number of the code segment may be obtained from the process record for the one process associated with this module. lg9c2

The segment number and displacement are saved in the link record in the calling process's linkage segment and the values and the record is marked as linked so that later uses will simply use the previously computed data. lg9c3

The components of the link record are lg9c4

defined flag lg9c4a

true if both segment number and displacement are known for this link lg9c4a1

segment type flag lg9c4b

true if know segment type (code or data) lg9c4b1

segment type lg9c4c

segment number flag lg9c4d

true if know segment number lg9c4d1

segment number lg9c4e

displacement flag lg9c4f

true if know displacement for this link lg9c4f1

displacement lg9c4g

pointer to the symbolic name of the module lg9c4h

pointer to the symbolic name of the variable or procedure lg9c4i

The flags are present because it must be possible to undo the linking and revert to symbolic form when the segment referred to by the link ceases to exist in the job. The pointers to the symbolic names must be preserved even after the values have been determined. The symbolic form may be restored by simply turning on the bit that says the link needs to be evaluated. The flags also play a role in prelinking modules (see the JOIN operation). lg9c5

Two symbol tables are needed in each module to provide for dynamic linking. lg9c6

One contains the symbolic names used by the module in references to other modules. This is called the outsymbol table. For each such name the outsymbol table simply contains the length of the string and the string. lg9c6a

The other table contains the symbols in the module that may be referenced from other modules. This is called the insymbol table. There are two parts to the insymbol table: a hash table and a set of string records. lg9c6b

The hash table may be any length. The first word of the insymbol table gives the length of

the hash table. The compiler will pick an appropriate hash table size depending on the number of symbols in the insymbol table. Each nonzero entry in the hash table points to a list of strings all hashing to a value which equals the index for that entry when taken modulo the table size. Thus to look up a string, hash it, divide the hash by the table length, use the remainder as an index into the table to find a pointer, follow the chain specified by the pointer until find the string. lg9c6b1

Note: multiple entries are chained from a single hash to facilitate incremental changes to the insymbol table. lg9c6b1a

For each string there is a record containing: lg9c6b2

a pointer to the next string on this chain (or 0 if this is the last record on the chain), lg9c6b2a

code or data flag (determines which segment contains this variable), lg9c6b2b

value (displacement), lg9c6b2c

length of string, lg9c6b2d

string. lg9c6b2e

(2) accessing a variable in a unit module lg9d

Like case 1, except the data segment number is saved rather than the code segment number. lg9d1

(3) accessing a variable in a message process lg9e

An example of this is a reference to a particular port in some process. lg9e1

The symbolic name of the variable is kept in the linkage segment. The process number is used to load the symbol list of the corresponding module from which the variable can be found. lg9e2

For connecting ports, the process number and port

number are saved as described under the CONNECT port TO port operation. lg9e3

The displacement corresponding to the variable name depends on the module. Since this category of reference is based on process number rather than module name, subsequent references to this "variable in process" cannot automatically use previous results. Instead, keep the variable's displacement and module name (or number?) in the linkage segment. when next go to access the variable. check if same module. If it is then can use the old value, else start over. lg9e4

The above three categories are the only kinds of interprocess linking. Calls to procedures in message processes are only allowed from within that process. Control may be passed to a message process only by sending a message to it. lg9f

(9) CONNECT port1 TO port2 lg10

Ports must be connected before messages can be sent or received over them. lg10a

The ports being connected cannot be in the same process. Both of the ports may be in processes other than the one doing this operation. The syntax for specifying a port in another process is "portname IN processname". lg10b

The data defining a port is kept in the data segment of the process. Thus there is a unique set of ports for each process. lg10c

A port consists of a flag saying whether the port is connected or not, a process number and a port number for the port to which this port is connected, and if this is an initialization port, a pointer to a starting location within the code segment. lg10d

The first message to a process causes it to be started and must come over an initialization port. The declaration of an initialization port includes the name of a procedure in the module which is used to determine where the process is to be activated. By having several initialization ports, it is possible for a process to be started in different locations depending upon which port is used. lg10d1

The CONNECT operation checks that both ports are disconnected, then stores the appropriate data in each and marks both of them as connected. lg10e

(10) DISCONNECT port lg11

This operation does nothing if the port is not connected. lg11a

Otherwise both this port and the one to which it is connected are marked as disconnected. The port need not be in the process performing the operation. lg11b

(11) REPLACE port1 BY port2 lg12

Port1 must be connected. Port2 must be disconnected. lg12a

Let port3 be the port to which port1 is connected. Then this operation is equivalent to DISCONNECT port1 followed by CONNECT port2 TO port3, except that the identity of port3 need not be known be the process performing the operation. lg12b

(12) message sending lg13

In the initial implementation, a message will consist of a vector of scalars, like the arguments or the results of a procedure call. Eventually all of these should be generalized to allow arbitrary data structures. lg13a

When process p sends a message over its port k, then lg13b

(a) confirm that port k is connected, lg13b1

(b) stop process p and set its state as pending a message on port k, lg13b2

(c) save the reactivation location for p, lg13b3

(d) let q,m be the process,port to which k is connected. confirm that q is either pending a message on m or else is unstarted and m is an initialization port, lg13b4

(e) and finally, activate q after setting up the segment pointers. lg13b5

The process q will execute code to accept the message in

much the same manner that arguments are passed to a procedure or multiple results are returned from a call. lg13c

Process p will be reactivated when a message arrives back over its port k. Normally this will be a message from q over m, however since connections may be modified while processes are active, the reply may come from some other port in another process. lg13d

The message sending described above causes control to be passed whenever a message is sent. This may be called the "send and wait for reply" operation. Other message operations are possible. lg13e

Four possiblities are lg13f

send and wait for reply, lg13f1

send and continue processing, lg13f2

receive, and lg13f3

receive from any of a set of ports. lg13f4

The detailed effect of these operations depends on whether the specified message path is currently in a synchronous or asynchronous mode (an idea from Bob Balzer's ISPL). lg13g

When a message is sent over a synchronous path, the sending process is blocked and the receiving process is activated. The sending process is reactivated when a reply is sent back over the same message path. Thus the only valid operation with a synchronous message path is send and wait for reply. lg13h

When done with an asynchronous path, a send and wait for reply operation can have a different effect. The message is put in the message path and the sending process is blocked. The receiving process will not automatically be activated however. It will be activated if it has been waiting for a message on this path and it has not been explicitly stopped, otherwise someother process will be choosen by the system for activation. lg13i

In general, if the receiving process has been stopped (by the "Stop" command) then it will NOT be activated as

a result of sending a message to it. It must first be restarted by a "Start process" command. lg13j

The syntax of a send and wait for reply operation is based on the syntax for a procedure call. The message sent corresponds to the arguments while the reply corresponds to the multiple results. The word "PORT" followed by the port name replaces the procedure name of the call. lg13k

Formally, the syntax for a send and wait for reply is lg13k1

[reply1 '←] "PORT" port ['([message] [reply2] ')] lg13k1a

where lg13k2

reply1 = lhs; lg13k2a

the first word of the reply message will be stored here. lg13k2a1

port = name of a port in this process; lg13k2b

the message is to be sent over this port lg13k2b1

message = expression $(', expression); lg13k2c

the expressions are evaluated and sent as the message. lg13k2c1

reply2 = ': lhs $(', lhs); lg13k2d

any words in the reply after the first are stored into these locations. lg13k2d1

lhs = any variable form that can appear on left hand side of an assignment statement; lg13k2e

Just as in calls, messages may be sent and replies accepted which consist of no values. Such null messages serve to transfer control to another process. lg13k3

The other message operations are limited to asynchronous paths. lg13l

The send and continue processing operation simply adds

the message to the queue then allows the sending process to continue. lg13m

The syntax of a send and continue processing operation is lg13n

```
"PUT" port '( message ');
```
lg13n1

The receive operation allows the requesting process to continue if there is a message waiting for it in that path. Otherwise the process is blocked until a message does arrive, and the system chooses some other process to activate. lg13o

The syntax of a receive operation is lg13p

```
"GET" result "FROM" port;
```
lg13p1

```
result = lhs / '( lhs $(', lhs) ');
```
lg13p2

When a message path is created by the "connect" command, it is initially put in the synchronous mode. The mode of a message path may be changed by the command "Make port (Synchronous / Asynchronous)". It is illegal to make a path synchronous if there are messages from either process in the path, however there may be a request in the path from the other process. This restriction is to ensure proper synchronization of the messages and replies during later use of the path. lg13q

If message paths are being used asynchronously a process may wish to output several messages, then wait for a reply over any of a set of ports. It will also be desirable to be able to execute a particular statement depending on which ports actually provided the reply. The syntax for doing this is lg13r

```
"FROM" $pstat "END";
```
lg13r1

```
pstat = source "GET" result ': statement ';;
```
lg13r2

```
source = port $(', port);
```
lg13r3

Any of the listed ports may be the source of the reply. lg13r4

Example: lg13s

```
FROM                                                    lg13s1

   port1 GET c : CALL p;                                lg13s1a

   port3 GET (a, b) : CALL q;                           lg13s1b

   port2, port5 GET c : CALL r;                         lg13s1c

END;                                                    lg13s2
```

When this statement is executed the following takes place. First the listed ports are inspected in the order they appear. If a message is found waiting on one of the ports then it is stored in the result and the associated statement is executed. If none of the listed ports has a message waiting, then the process is blocked and reactivated when a message arrives on one of the ports. lg13t

(13) test for messages lg14

When ports are used asynchronously, it may be valuable to determine whether or not there is a messge waiting on a particular port rather than simply hanging for the arrival of a message. lg14a

This operation returns the number of messages waiting on the specified port. lg14b

Another flavor of the operation might return the number of messages waiting on all ports for the process. lg14c

(14) STOP process lg15

The named process will not be activated, even if messages arrive for it, until a START process command is executed for it. If no process is named then the process executing the command is stopped. lg15a

(15) START process lg16

This operation allows the named process to be activated when a message is sent to it. Reverses the effect of the STOP process command. Note that even a "START"ed process will not be activated until a message is sent to it. lg16a

(16) error recovery lg17

???? how should this be handled ???? 1g17a

MOTIVATION FOR A DATA DEFINITION FACILITY 2

The purpose of a data definition facility is to give the programmer a means to work as directly as possible with his choosen data structures, letting the language system do the mapping into machine acceptable form. 2a

The programmer designs data structures which are (hopefully) appropriate for his problem. A program then involves constructing, testing, and manipulating objects of the choosen structures. However, common programming languages tend to limit the possible structures or make their use very difficult. "Higher level" languages like Algol provide very limited choice of data structures but powerful means for testing and manipulating them. Assembly or machine oriented languages allow the programmer to construct arbitrarily complex structures, but at the cost of mapping those structures into a form acceptable to the machine, i.e. bits in words. This means that a great deal of the programming effort goes into converting the choosen data structures into bit sequences, an occupation often referred to as "bit twiddling". This conversion must be done every time an operation on a structure is performed and therefore can be very time consuming and a great source of errors. 2b

By means of a data definition facility, the programmer can design complex structures and then operate on them in a relatively direct manner with the language taking responsiblity for implementing the "bit twiddling" aspects. 2c

In addition, by letting the language system "know" more about the defintion of the structures, it can provide run time checking and debugging aids to catch attemps to construct objects that do not conform to the definitions. Furthermore, in an interactive system, the data may easily be displayed according to the programmers definition which should prove to be a considerable debugging aid. 2d

The following sections describe a facility for defining data structures and for constructing, testing, and manipulating objects of the defined types. The syntax definitions are given as abbreviated Tree Meta parse rules having the following form. 2e

A right part of a rule consists of one or more alternatives. If there is more than one alternative, they

are separated by slashes (/). Each alternative consists of a sequence of elements. A subsequence enclosed in square brackets, *[* and *]*, is optional. All other elements in the sequence must occur in the specified order. 2e1

The elements may be any of the following: 2e2

the name of a rule; 2e2a

a call on a basic recognizer such as .NUM for a number; 2e2b

a string enclosed in quotes ("); 2e2c

a single character string indicated by an apostrophe (') followed by the character; 2e2d

a list of alternatives enclosed in parentheses; 2e2e

a dollar sign ($) followed by an element, which means a abitrary number of occurences (including zero) of the element. 2e2f

Comments enlosed in percent signs (%) may be embedded anywhere in the rule. The rule is terminated by a semicolon (;). 2e3

DATA DEFINITION SYNTAX 2f

data.definition = data.type.name *[*indexdef*]* '= data.expression; 2f1

If there is an index then objects of the defined type consist of an array of objects of the type given by the data.expression. Otherwise, objects of the defined type consist of a single element of the specified structure. 2f1a

indexdef = 2f2

'*[* '$ '*]* / 2f2a

Meaning indefinite number of elements. The number of elements may be changed at run time. New elements may be added anywhere in the sequence and old elements may be deleted from anywhere in the sequence. 2f2a1

'*[* range.name '*]*; 2f2b

```
        Index may take on values from the specified range.
        Range defintions are discussed below.                    2f2b1

    data.expression = dconcat $("Or" dconcat);                   2f3

      A single data.type may have alternative structures.  If
      there are alternatives, each one may be given a name.      2f3a

    dconcat = [data.type.name ':] dprime $("Concat" dprime);     2f4

      Structures may be concatenated to form new structures.
      A concatenation simply means an element of the first
      type followed by an element of the second type.            2f4a

    dprime = '[ component $( ', component) '] / data.space;      2f5

      Preexisting types may be used in concatenations.           2f5a

        For example a "node" might be defined as a "list"
        concatenated with an integer, where a list would be
        defined elsewhere in the program.                        2f5a1

      Alternatively, a sequence of components may be given.      2f5b

    component = [component.name [indexdef] ':]
    data.expression;                                             2f6

      Each component may be named.  These names may be used in
      ways described below.  If the component is indexed, then
      it is called an array type component, and consists of an
      array of elements of the type given by the
      data.expression.  Otherwise, the component consists of a
      single element of the specified type.  (Note the
      parallel between definition of a data type and the
      defintion of a component within a type).                   2f6a

    data.space =                                                 2f7

      '( data.expression ') /                                    2f7a

      '↑ data.space /                                            2f7b

        A pointer to a structure of the specified type.          2f7b1

      data.type.name /                                           2f7c

        A type defined in the program.                           2f7c1
```

```
        '= exp /                                                    2f7d

           The component is fixed as the value of the expression
           at the time an object is created  (not the time of
           definition of the structure).                            2f7d1

        basic.types;                                                2f7e

      basic.types =                                                 2f8

        "Nil"  %the empty data space% /                             2f8a

        "Any" /                                                     2f8b

           %any type. no restrictions on the type of structure
           that may occur in this position%                         2f8b1

        "Real" /                                                    2f8c

        "Integer" /                                                 2f8d

        "String" /                                                  2f8e

        "Character" /                                               2f8f

        "Bit" ['( exp ')];                                          2f8g

           Value of exp specifies length in number of bits.  Exp
           is evaluated at each instantiation.  If exp is absent
           then component is assumed to be a single bit.            2f8g1

CONSTRUCTORS                                                        2g

   Execution of a constructor makes a new object of the
   specified type (an action referred to as "instantiation")
   according to the given parameter list, and has that object
   as its value.                                                    2g1

       constructor = "New" data.type.name [paramlist];              2g1a

   Unspecified components are initialized to the special value
   "undefined" and may not be referenced until they are
   assigned a value.  Thus if the parameter list is not
   present, all components are simply set to undefined.             2g2

   If present, the parameter list is of the following form:         2g3

       parameter.list = '[ parameter $(', parameter) '] ;           2g3a
```

```
parameter = /component.name ':/ cparam;                          2g3b

  If the component name is not given, then the position
  in the parameter list is assumed to correspond to the
  position of the component in the defined type.  Thus
  may use component names to avoid ambiguity and leave
  them out where they are not necessary.                         2g3b1

  If component names are used then need not specify all
  the parameters (those unspecified will simply be set
  to undefined).                                                 2g3b2

  This works for specifying subcomponents also: the
  programmer is thus free to use component names or not
  in specification of subcomponents indendent of their
  use in component specification.                                2g3b3

cparam =                                                         2g3c

  parameter.list %allowing for nesting% /                        2g3c1

  '? %initial value = undefined% /                               2g3c2

  '( exp $( ', exp ) ') /                                        2g3c3

     %for entering an arbitrary number of elements in
     an array type component%                                    2g3c3a

  exp %value of exp becomes the component%;                      2g3c4
```

If there is more than one alternative for the data.type the alternative constructed will depend on the parameter list. The constructor chooses alternatives from left to right in the definition of the type and tries to build an object representing that alternative. If the arguments are not compatible with that alternative then the next one is tried. If the constructor is unable to build an object of the specified type with the given arguments the construction fails. 2g4

Examples: 2g5

Define 2g5a

```
node =                                                           2g5a1

   /head: /alink: atom, blink: atom/, tail: Integer/             2g5a1a
```

```
      or [seq [$]: Integer];                                    2g5a1b
    atom = ↑node or Nil;                                        2g5a2
  New node [head: [alink: ↑x, blink: ↑y], tail: n]              2g5b
    The equivalent form without component names is              2g5b1
    New node [[↑x,↑y],n]                                        2g5b2
  New node [seq: (w, x, y, z)]                                  2g5c
  New node [seq: ?]                                             2g5d
PREDICATES                                                      2h
  Predicates allow run time testing of objects.  The
  predicates may test both structure and value.                 2h1
  The syntax for a structure.test is as follows:                2h2
    structure.test =                                            2h2a
      data.type.name [test.components] /                        2h2a1
        May test if the object is of the given type
        independently of testing its substructure.              2h2a1a
      '↑ structure.test /                                       2h2a2
        Tests if the object is a pointer to the structure
        specified by the structure.test.                        2h2a2a
      basic.type;                                               2h2a3
    test.components = '[ scomp $( ', scomp) ']';                2h2b
    scomp = [component.name ':] (                               2h2c
      As in the construction of objects, if the component
      name is absent, then it is assumed that the order of
      the component in the list corresponds to its position
      in the definition of the type.                            2h2c1
      test.components %allowing testing of subcomponents% /     2h2c2
      '- %any element% /                                        2h2c3
```

```
    '$ %an arbitrary number of any elements% /                    2h2c4

    binary.relation %an element satisfying this
    relation%;                                                    2h2c5

  binary.relation = [("NOT"|'~)] binrl;                           2h2d

  binrl =                                                         2h2e

    ('= / '#) (structure.test / sum) /                            2h2e1

    (">=" / "<=" / '< / '>) sum /                                 2h2e2

    "IN" ('( / '[) sum ', sum (') / '));                          2h2e3
```

This definition of binary relation is used for all logical tests, Thus wherever can test the value of an element will also be able to simultaneously test its structural composition. 2h2f

In particular, the "Case" statement will allow Tree Meta-like testing of strutures. 2h2g

```
  case = "Case" exp "From" $cases "Endcase" statement;            2h2h
```

The value of the exp, which may be an arbitrarily complex object, is used to chose one of the cases. Each of the cases is headed by a test for either structure, value, or both. The first test which succeeds determines which of the cases will be chosen. For structure tests, the test is true if the object resulting from the evaluation of the exp has the specified structure. If none of the tests succeeds, then the Endcase is executed. 2h2h1

```
  cases = ctest $("Or" ctest) ': statement [';];                  2h2i

  ctest =                                                         2h2j

    structure.test /                                              2h2j1

      %for testing structure as well as value%                    2h2j1a

    binary.relation %for testing value of simple
    variables%;                                                   2h2j2
```

Example: 2h2k

```
Case x From                                                    2h2k1
   node[[↑node,Nil],-]: Begin ... End                          2h2k1a
   node[-, >500]: ....                                         2h2k1b
   atom[=↑z] Or ↑Any Or =0: ....                               2h2k1c
   Endcase ..                                                  2h2k1d
```

SELECTORS 2i

Selectors allow access to components of structures. 2i1

```
selected.var = $selectors pointer.primary;                     2i2
```

The pointer.primary is an object of type pointer. For example, it may be a pointer type variable or the result of a call to a pointer valued procedure. 2i2a

```
selectors = '@ / qualified.name "of";                          2i3
```

A selector may be applied only to a pointer. A '@ refers to the quantity pointed to by the pointer. A qualified.name refers to a specific component, or element of an array type component, of the object pointed to by the pointer. 2i3a

```
qualified.name =                                               2i4
   (data.type.name / component.name) [index] $(':
   subcomponent);                                              2i4a
```

A qualified.name may start with a component name as long as the component name defines a unique component. 2i4b

```
subcomponent = (component.name / .NUM) [index];                2i5
```

These define which of the subcomponents is to be selected, and if the subcomponent was defined to be an array, then which element of the array. 2i5a

If a number is used instead of a component name, then it selects the subcomponent in that position (with the positions numbered from left to right starting with one). For example, 2i5b

if x has n subcomponents, then x:3 selects the third one. 2i5b1

If x is an array of elements, each element having several components, then x/n/:3 selects the third component of the nth element in the array. 2i5b2

Or, if the components of the elements are named a, b, and c, then an equivalent selector is x/n/:c. 2i5b3

index = '/ (exp / '$) '/; 2i6

The value of the expression must be an element from the range for which the component is defined. 2i6a

If the component was defined to be an array, then it may be indexed with a $ in a "Foreach" statement (see below). 2i6b

POINTERS 2j

A pointer to a selected.variable is formed by writting '↑ followed by the selected.variable. 2j1

Iterative construct 2k

Need way to move thru the elements of an array. 2k1

Foreach statement 2k2

"Foreach in" selected.variable "Do" statement; 2k2a

The selected variable should be an array. 2k3

Inside the iterated statement, the current element of the array may be referred to as selected.var/$/. 2k4

If the component was defined to be an array indexed by a certain range, then $ will take on the various values in the range. 2k4a

If the component was defined to have an arbitrary number of elements (with /$/) then $ simply indexes thru the elements. 2k4b

Can test if selected.var/$/ is first or last of sequence. Need to be able to insert and delete from an iterated component. 2k4c

DESCRIPTION BLOCK 3c1

backpointer to name of procedure (hash number). 3c1a

type = procedure / data 3c1b

number of args, results (if procedure) 3c1c

pointers to dependency chain, occurence chain 3c1d

pointer to statement chain in SMAP 3c1e

pointer to local name table (if procedure) 3c1f

starting address, length 3c1g

date/time/initials when last compiled 3c1h

For each procedure also have 3d

LOCAL NAME TABLE 3d1

containing information about the names that are limited in scope to this procedure 3d1a

Mapping from source to object and object to source 3e

When the user places a breakpoint at some (source) statement, the system must be able to determine at which object code location the actual break instruction is to be placed. 3e1

Conversely, when an event, such as the execution of a break instruction or ann illegal instruction, takes place in the object program, the system must be able to show the user the source statement which corresponds to the object instruction in question. 3e2

The system must construct data structures which will allow easy mapping from source statement to object instruction and back. The following describes the choosen structures and how they are built by the system. 3e3

When compiling a procedure, the variable lc specifies the number of instructions which have been produced, the variable oldlc specifies the value of lc when the current source statement was input, and the variable csid is the statement identifier of the current source statement. 3e4

At start of compilation of a procedure, 3e4a

lc ← oldlc ← 0 and csid ← SID of first statement. 3e4a1

When reach the point where are going to begin the parse of a new statement, construct an entry in SMAP describing the previous statement. 3e5

The statement is uniquely specified by its (file #, sid) pair which will be refered to as its SIP (statement identification pair). 3e6

The SIP is hashed to get an entry into the table SMAPHASH. This entry points to a list of all SMAP entries with that particular hash. 3e7

Search the list of entries to make sure that this particular SIP does not occur. (If it does then a single statement is being used in more that one place which is definitely a no-no. Treat as a syntax error.) 3e8

Add a new SMAP entry to the list for this hash number. 3e9

This list is doubly linked since must be able to easily remove entries. 3e10

Now link the entry into the chain of entries for the procedure being compiled. This linkage need not be double since will never be deleting an entry from the middle of the list. 3e11

Now the data fields of the new entry are filled in. These consist of the SIP for the statement, the hash # for the procedure being compiled, and the current values of lc and oldlc. 3e12

The hash # will later provide the means to find the starting address for the code buffer associated with this procedure. The saved values of lc and oldlc give the upper and lower bounds within the code buffer for the instructions produced while this statement was input. 3e12a

The last step after completing the SMAP entry is to set oldlc to the value of lc and csid to the SID of the current input statement. 3e13

To map an SIP to an address: 3e14

Hash the SIP and follow the chain until find the appropriate SMAP entry. 3e14a

Find the starting address for the buffer (using the hash #). 3e14b

Use the saved value of oldlc to find the first location which was produced for that statement. 3e14c

To map an address to an SIP: 3e15

Find the buffer in which the address lies (a processs to be described later). 3e15a

Follow the list of SMAP entries for that buffer until reach one with an oldlc, lc pait wich includes the instruction. 3e15b

Read the SIP from this entry. 3e15c

Dependency/Occurrence List 3f

When a procedure is recompiled it may change in size and need to be relocated in the object program. References to this procedure must then be modified to use this new location. The system thus needs to be able to find all of the references to a particular name. This is done by maintaining a reference list for every name. 3f1

(This list can also be of value to the programmer as a dynamically updated cross reference.) 3f1a

When a procedure is recompiled one of the first steps is to remove all of the references that were previously listed for it. (NOTE: these are references BY the procedure, not TO the procedure.) 3f2

A new list of references by the procedure will be created as the procedure is recompiled. 3f2a

This list of references by a procedure is called its dependency list. The list of references to a name is called its occurrence list, 3f3

As the procedure is compiled the dependency and occurrency lists 3f4

A System for Modular Programming

<JOURNAL>7053.NLS;1, 29-MAY-71 12:14 WHP ;Title: Author(s): William H. Paxton/WHP; Distribution: William H. Paxton/WHP; Keywords: programming module segmentation processes; Clerk: WHP;
Origin: <PAXTON>MOD.NLS;3, 16-FEB-71 17:33 WHP ;

An Introduction to the Augmentation Research Center

This paper will be in the proceedings of the NATO Advanced Study Institute on Display Use for Man-Machine Dialog.

An Introduction to the Augmentation Research Center

William H. Paxton
Stanford Research Institute
Menlo Park, California

To appear in the proceedings of the NATO Advanced Study Institute on Display Use for Man-Machine Dialog, held in Erlangen, Germany, March 30 - April 7, 1971.

Abstract:

An outline of some of the objectives of the Augmentation Research Center and a discussion of the prototype system being developed there.

Introduction:

This paper provides a brief outline of the goals and a discussion of the results of a research group that has been concerned with display use for man-machine dialog since the early 1960's. The Augmentation Research Center (ARC), headed by its founder Dr. Douglas C. Engelbart, includes about thirty members who are actively engaged in both the use and the development of a prototype augmentation system.

Goals:

The research effort at ARC is centered around two interrelated objectives:

1) an exploration of various possibilities of augmenting the performance of intellectual tasks, and

2) the experimental development of a computer-based augmentation system.

In order to augment the performance of the intellectual work of an individual or of a group, increased capabilities in certain domains must be provided. In particular there should

be an increase in the capacity to approach complex situations and to identify the problems therein, to gain comprehension of the nature and context of these problems, and to derive solutions which satisfy given constraints. These improved capabilities should provide both quantitative and qualitative improvements in the accomplishment of intellectual tasks. In addition to faster and more thorough understanding of problems and, therefore, faster and better solutions, there should also be comprehension of problems in situations that previously appeared too complex and, thus, solutions to problems that seemed unsolvable. 5b

Historically such increased capabilities have been achieved through the use of many different aids. Some of the most significant have been language, writing, books, and recently, the computer. These all augment human capacities by the externalization of concepts in symbolic form for both manipulation and communication. 5c

The ARC Augmentation System: 6

By its very nature a computer is well-equipped to aid man in the manipulation and communication of information. A computer-based augmentation system can thus be expected to provide powerful facilities for the continuous creation, study, and revision of symbolic structures. Such a system can be the normal daily working environment for an individual or group. In effect, it can be an on-line office -- complete with powerful means to organize ideas, edit writing, and study information. An on-line system of this sort can provide access to communication systems, clerical services, and special-purpose facilities. 6a

The development and experimental use of such an augmentation system constitute the principal activities of the Augmentation Research Center. The remainder of this paper describes some of the features of this prototype system knowns as NLS, an acronym for "on-line system". 6b

The user's access to NLS is by means of a display console. This display provides views of files of information, as well as other visual indications of the state of the system. The user's input devices at the display console are a keyboard, a

keyset, and a selection device. The keyboard is a standard typewriter keyboard except for the addition of a few keys for special functions. The keyset consists of five keys which may be depressed in combinations to give thirty-one possible chords. Each chord represents a different character with the result that the keyset provides an alternative to the keyboard for many input operations. In particular most commands can be performed with one hand on the keyset for character input and the other hand on a special selection device, called a "mouse", for operand specification. The mouse consists of orthogonally mounted sensors which determine horizontal and vertical positions of a cursor on the display screen. By moving the mouse over a surface, the cursor is made to move in a similar path over the screen. 6c

The files which are viewed and manipulated at this display console are hierarchically structured. The information in the files has a tree structure with a string of text and perhaps other information associated with each node in the tree. There are commands to edit both the structure of the file and the textual component of the node. 6d

There are typically three phases to the execution of such commands: operation specification, operand specification, and confirmation. The man-machine dialog in these commands has been carefully considered. In particular the commands are designed to allow defaults, iteration with minimal respecification of operands, and backup on errors after partial specification. 6d1

The operation specification for an editing command involves naming the type of operation such as "insert", "move", or "replace", and the type of structural or textual entity such as "character", "word", "statement", or "branch". Any operation type may be combined with any entity type to form a valid command. 6d2

The syntax of the editing commands has been described in a Backus-Naur type of meta-language both as a pedagogical aid and as a design aid. The description provides concise documentation for the user and suggests possible extensions and improvements to the system designer. 6d3

In addition to the editing commands, there are a large number of commands for studying the information in a file or a group of files. These studying capabilities include "jumping", "linking", "view control", and "portrayal generation". 6e

Commands for "jumping" to a new view of a file allow many ways of specifying the new location. Just to name a few, it is possible to specify an absolute location in the structure, a location at a position relative to some other given location, a location with a particular user-given name, or a location containing a user-specified content. 6e1

A powerful form of jumping makes use of "links". A link is a textual entity which specifies a file directory, a file in that directory, a location in that file, and a view to be used at that location. By jumping with links, it is possible to move quickly and easily through complex configurations of information contained in many different files. 6e2

As well as being able to jump to new locations, the user may determine what "view" of the information will be presented when he gets there. The simplest form of view control simply limits the depth into the tree of displayed statements. More complex view control can make use of information such as the time of the last editing of the statements, the identity of the last person to edit the statement, or the results of user-programmed tests of the content of the statement. 6e3

In certain instances it is necessary to have more powerful view control which generates different portrayals of a given body of information depending on the interests of the viewer. To accomplish this, the user may write programs which will analyze and reformat the information stored in the file. This facility has been heavily used for management and catalog systems. 6f

For management planning use the data about many tasks, the people involved, the effort levels, and the checkpoints can be integrated and displayed for various analyses. Special views can be constructed with distracting or irrelevant data filtered out by analysis programs. 6f1

for catalog data management and index production, user-created programs are used to extract citation subcollections, construct single or multiple views of the items in the collections, and to format stored collections for display or printing. 6f2

In the reports listed in the bibliography there are descriptions of the computer and display facilities, the results of comparing the mouse to some other display selection devices, descriptions of the software techniques used in NLS, and more thorough discussions of the goals and objectives of ARC. 6g

Conclusion: 7

The potential for augmenting the performance of intellectual tasks by computer aids is obviously great. As a prototype system, NLS serves both as a means for exploring and developing new ideas and as a valuable tool in the work of the Center. It should be emphasized that it is a continually evolving system, changing as the understanding of augmentation systems grows and as such understanding is applied. The system described in this paper will certainly appear primitive when compared to augmentation systems of the future. It is important to remember that NLS is not the only possible approach to developing an augmentation system; it is the hope of the staff of ARC that other groups will become involved in this intersting field and advance alternatives. 7a

Bibliography: 8

---- include the standard ARC reports and articles ---- 8a

An Introduction to the Augmentation Research Center

<JOURNAL>7054.NLS;1, 29-MAY-71 12:40 WHP ; (Expedite) Title: Author(s): William H. Paxton/WHP; Distribution: Douglas C. Engelbart, Charles H. Irby, James C. Norton, Ed K. Van De Riet, Richard W. Watson/DCE CHI JCN EKV RWW; Keywords: augmentation arc introduction; Clerk: WHP;
Origin: <PAXTON>INTRO.NLS;4, 29-MAY-71 12:34 WHP ;

JCN 29-MAY-71 14:51 7055

JCN Notes: NIC DSS Software Planning Session 5/26/71 1:00-3:00

JCN Notes: NIC DSS Software Planning Session 5/26/71 1:00-3:00

Present: 1

RWW CHI WHP BLP JCN WSD 1a

Purpose: 2

To continute discussion of key software coordinating roles. 2a

To review in detail NIC DSS requirements and priorities from the standpoint of ongoing and planned software tasks. 2b

To help get a balanced plan that meets NIC and other needs on a priority-oriented basis. 2b1

Results: 3

We discussed the proposd software coordinating roles and agreed on the following as proposed: 3a

Personnel coordinator -- BLP 3a1

Architecture coordinator -- CHI 3a2

Methodology coordinator -- WHP 3a3

JCN has been gathering and developing notes on the nature of these and other roles as they have been shaping up and will publish them for ARC review soon. Per baseline task (journal,6969,roles:zgwCn). 3b

We went over WSD's task list (NIC related only). 3c

It appears that DSS tasks will need implementation help from other software people if we are meet NIC needs on the schedule proposed for stages. 3c1

The main service of NIC is dialog support so DSS tasks are very important to the success of the whole NIC operation. 3c2

The balancing process over the next two days should make clear what help we can give WSD here. 3c3

We discussed user and DSS document addressee identification problems and the question of having every ARC/NIC online user be assigned a TENEX username, rather than grouping them under site usernames. We agreed that

this requires coordinated design at a different meeting (with JTM, KEV, and others). 3c4

We also discussed the problems we have been experiencing with occasional unstable introduction of new system features into operational use. 3c5

These problems seem to come from several sources. 3c5a

At times we have not done a thorough job of design before implementing features. This has meant that potential internal design system conflicts have not been discovered until new features are put into operation. Their discovery has been at the expense (time and frustration) of ARC users and sometimes of other software people who have had to change "their" parts of the system to restore them to use, the Journal being one of several examples. 3c5a1

It is also possible that we have not tried early enough to discuss potential conflicts with others affected, so that they could plan their work or make design decisions to soften the effect of the incompatability of proposed system changes. 3c5a2

These are just the kinds of problems that the coordinating roles, particularly methodology (designs,..) and architecture (conflicts,..) are expected to act upon. 3c5a3

The design documents and dialogue in the baseline record (entered into the journal) are to provide visibility to aid this coordinating process. 3c5a3a

Next Action: 4

BLP and RWW were to go over the NIC-related software tasks based on the discussions of the past two meetings to tag (for reordering) the tasks in the baseline record according to NIC stages 0, 1, and 2 and relative priorities. 4a

BLP should get the resulting data organized into the Baseline record datafile (msr,baserec,) and prepare useful views of the planned tasks that can be used (by him) in an inital balancing try. 4b

The next software planning meeting was to be 5/27, but was later changed to 5/28 at 1:00, with RWW BLP CHI WHP JCN. 4c

After this meeting, the basic task plans should be ready for discussion with DCE and with individual task pushers of ongoing tasks and negotiation for tasks about to start. hc1

JCN 29-MAY-71 14:51 7055

JCN Notes: NIC DSS Software Planning Session 5/26/71 1:00-3:00

<JOURNAL>7055.NLS;1, 29-MAY-71 14:51 JCN ; (Expedite) Title: Author(s): James C. Norton/JCN; Distribution: Douglas C. Engelbart, Richard W. Watson, Bruce L. Parsley, Charles H. Irby, William H. Paxton, William S. Duvall, John T. Melvin, Ed K. Van De Riet, Kenneth E. Victor/DCE RWW BLP CHI WHP WSD JTM EKV KEV; Keywords: ; Clerk: JCN;

a proposal for handling disk errors

-KEV 1-JUN-71 8:58 7056

-KEV 1-JUN-71 8:58 7056

a proposal for handling disk errors

I propose that we start handling disk errors in the following manner: 1

When a disk error occurs: 1a

A resident page of the monitor will be updated 1a1

This page will contain a list of disk bad spots 1a1a

At the same time, a disk copy of this page will be updated 1a2

This sector will be marked in the system disk bit table as in use 1a3

When the system goes to deallocate a sector from the bit table, the address of the sector to be freed will be checked against the list of bad spots, and if the sector appears in tha badspot table, it will not be deallocated 1b

Please comment on this proposal to me in person, as i would like to implement this, or another scheme, by Wednesday (6/2/71) night. 2

a proposal for handling disk errors

<JOURNAL>7056.NLS;1, 1-JUN-71 8:59 KEV ; (Expedite) Title: Author(s): Kenneth E. Victor/KEV; Distribution: Marilyn F. Auerbach, Walter L. Bass, Roger D. Bates, Mimi S. Church, William S. Duvall, Douglas C. Engelbart, Beauregard A. Hardeman, Martin E. Hardy, Fred P. Hocker, J. D. Hopper, Charles H. Irby, Mil Jernigan, Harvey G. Lehtman, John T. Melvin, Jeanne B. North, James C. Norton, Cindy Page, Bruce L. Parsley, William H. Paxton, Jeffrey C. Peters, Barbara E. Row, Ed K. Van De Riet, Dirk H. Van Nouhuys, Richard W. Watson, Don I. Andrews, Kenneth E. Victor/MFA WLB RDB MSC WSD DCE BAH MEH FPH JDH CHI MEJ HGL JTM JBN JCN CXP BLP WHP JCP BER EKV DVN RWW DIA KEV; Keywords: disk errors; Clerk: KEV;
Origin: <VICTOR>DISKPROPSAL.NLS;1, 1-JUN-71 8:28 KEV ;

MEJ 1-JUN-71 12:06 7057

Phone Log: Call from Wendy Smith, General Research Corp. about ARPA/IPT Library Collection Contract

MEJ 1-JUN-71 12:06 7057

Phone Log: Call from Wendy Smith, General Research Corp. about ARPA/IPT Library Collection Contract

TO: D.C. Engelbart From: Mil Jernigan 1

Phone Log: Call From Wendy Smith, General Research Corporation Aboabout ARPPA/IPT Library Collection Contract. 2

This morning (5-28-71) I received a phone call from Wendy Smith, General Research Corporation, 1501 Wilson Boulevard, Arlington, Virginia 22209, phone (703) 524-7206. Miss Smith Wanted copies of our contract reports covering work done for ARPA. Dr. Chrisler (head of the ARPA contract project at General Research Corporation) asked her to get copies. 2a

General Research Corporation (Dr. Chrisler) has a contract with ARPA to collect copies and compile a bibliography of all contractor reports that have been issued on ARPA/IPT contracts concerning information processing. Dr. Chrisler heard of NIC, without knowing what the Network Information Center was except that it was something to do with the ARPA Network. They (GRC) are not sure what form their final work will take, whether it will be a hard copy bibliography or just the hard copy collection. 2b

I asked her if they had a bibliography they could send someone, to please send us a copy; that we would be interested. I promised to send her a copy of our bibliography and mentioned the film. 2c

MEJ 1-JUN-71 12:06 7057

Phone Log: Call from Wendy Smith, General Research Corp. about ARPA/IPT Library Collection Contract

<JOURNAL>7057.NLS;1, 1-JUN-71 12:06 MEJ ;Title: Author(s): Mil Jernigan/MEJ; Distribution: Douglas C. Engelbart/DCE; Keywords: ; Clerk: MEJ;
Origin: <JERNIGAN>ARPA/IPTLIB.NLS;1, 1-JUN-71 10:44 MEJ ;

DVN 1-JUN-71 15:44 7059

Changes in NLS on or about 6/1/71

n

Changes in NLS on or about 6/1/71

CHANGES IN NLS 6/1 1

A new version of NLS includng the following changes is planned for 6/1. The branches describing the changes were writen by the programmers who made the changes. For clarification I have added some branches that begin, "In other words."

2

SINGLE QUOTES IN MAMES 3

When doing a jump to name and bugging something on the screen, the thing selected can look like: 3a

LD $(LD / '' LD) 3b

i.e. you can have imbedded single quotes in names, e.g., Task'Name'X.

3c

CHANGE NAME DELIMETERS 3d

There are 5 new commands: Name Delimiter Display and Name Delimiter Statement/Branch/Plex/Group. 3e

Name Delimiter Display will display in the name register the name delimiters of the selected statement. The syntax is: 3e1

'N 'D BUG CA %delimiters displayed% CA 3e1a

The other four allow a user to change the name delimiters for all the statements in the specified structural entity. The syntax is: 3e2

'N ('S/'B/ 'P/'G) (BUG/ BUG BUG) %"Left Name Delimiter" GETS DISPLAYED IN THE COMMAND FEEDBACK LINE/ CH CA %"Right Name Delimiter" gets displayed in the command feedback line% CH CA CA 3e2a

Also whenever a statement is inserted the name delimiters are initially set to the delimiters of the statement's successor. 3e3

In other words: Name delimiters are the visibles that set off the name of a statement from the following text. They have always been in parentheses. These comands allow you to use anything you want.

3e3a

YOU CAN BUG ANY VISIBLE FOR A FILE NAME 3e3b

Visibles for file and process name. WHP 12-MAY-71 11:40 3f

When make a bug selection for the OUTPUT FILE, OUTPUT PROCESS, etc. name instead of typing in a literal, a visible string will be used rather than a name as was the case. 3f1

In other words: In the course of an output command you may specify both the file and the directory by bugging the appropriate word on the screen.

3f2

BIG VIEWSPECS 3f3

In DNLS, the viewspecs get large when the two left mouse bottons are held down.

3g

CONTROL O 3h

Control O may be used (like rubout was supposed to work) to stop the sequence generator. This will stop such things as Print in TNLS, searches and Jumps to Name or Word or Content, the Output Processor, and other such things that go thru the file.

3i

CONTROL S 3j

In TNLS, Control S may be used to stop the printing of a particular statement without stopping the entire Print command. 3k

When in the Execute Edit command both ↑O and ↑S have their old meanings.

3k1

CONTROL R 3k2

In TNLS, ↑R now works to have literal retyped. 3l

Whenever a literal is being typed, you may type a control R. This character does not go into the literal but simply causes TNLS to type a carriage return followed by the current literal. It is then possible to continue typing

the literal.

311

ADDRESSES in TNLS WHP 3m

The entire string which defines the address is read before it is interpreted. This means that the string may be edited by BC, BW, etc. retyped by ↑R, just like a literal up to the point that it is accepted. If the address is in error, then the part of the address causing the error followed by a '? will be typed when the address is interpreted. 3m1

It is possible to use the literal escape character (control V) to specify that a character is to be used literally rather than for control. For example, to do a *[TEXT]* search in which the TEXT includes the character *]*, you may type a control V before the *]* in the TEXT and good things happen. 3m2

Indirect links ('↑) are not typed any more since when the ↑ is read the string has not yet been interpreted so the system doesn't know where the TOM is pointing 3m3

Links may now be typed in as part of an ADDRESS 3m4

Before could only use '(name '). 3m4a

In other words: You may load another file by typing space a and then a link, syntax : SP LINK CA. 3m4a1

'@ does a jump file ahead 3m5

In other words: When you move from file to file by links or the load file command in TNLS the system accumlates a record. This record is a ring of fles five files in circumference. @ (syntax: @) will move you to the file ahead of you on the ring, i. e. the file you were in four moves before. 3m5a

both @ and & may be preceded by a number 3m6

3& will do 3 jump file returns, -2@ is the same as 2& 3m6a

'; %Semicolon% string '; is string search limited to one statement 3m7

this search fails if don't find the target in the

current statement rather than going on to look in following statements as /string/ and <word> do. 3m7a

In other words: The command SP ;string; will take you to "string" if and only if "string" is to be found in the statment where your cursor started. If you use ;string; as an address in a command, you may be sure the command will be executed only on an example of "string" in that statment, not in some unknown passsage far down the text. 3m7a1

'' char searches for the character in the current statement 3m8

thus 'x is like :x; 3m8a

'← puts TOM to statement front 3m9

'> puts TOM to statement end 3m10

In other words: the left arrow command (syntax:←) moves your cursor to the first character of the statement, and the right angle bracket command (syntax:>) moves your cursor to the end of the statement. 3m11

altmode now simply echoes as a '$ 3m12

'# is used before markers rather than '$ 3m13

address must be terminated by either CA or C. 3m14

this is because the address is input as a literal 3m14a

this means that it is no longer possible to type SP ADDRESS / 3m14b

and no longer possible to use commas to separate addresses in a group selection 3m14c

In other words: to move to a statment and print it, you now need two commands, syntax:

SP ADDRESS CA
/ 3m14c1

or SP ADDRESS CA
\ (which is equivalent to Print Statement CA
CA) 3m14c2

and in giving the address of groups you must separate the two statement numbers by a command accept. 3m14c3

the + or - word, visible, etc. stuff has been changed to a simpler scheme. Try it and see.In other words: AGHGGGGG. 3m15

TNLS SLASH 3m16

Change to slash command in TNLS WHP 3n

The slash command now types at most 10 characters to either side of the CM 3n1

TNLS SYNTAX 3n2

Changes to command syntax in TNLS 3o

The commands in TNLS have been modified to allow iteration with the use of center dot (C.). 3o1

The following abbreviations are used in this description: 3o2

text1 3o2a

is a text entity that can be selected with one address; Character, Word, etc. 3o2a1

text2 3o2b

is a text entity selected that must be with two addresses; for now only Text. 3o2b1

strc1 3o2c

is a structural entity that can be selected with one address 3o2c1

strc2 3o2d

is a structural entity that can be selected with two addresses 3o2d1

The definition of LEVADJ used in the following is an arbitrary number of U's or D's optionally terminated by a space. Any other character, such as a command accept (CA)

or center dot (C.), serves both to terminate the LEVADJ and provide the next portion of command input. 3o3

Likewise, TEXT is any number of characters up to, but not including, a CA or C. 3o4

A sequence enclosed in parentheses, preceded by a dollar sign, and ending with an option for either a CA or a C., written (CA / C.), may be repeated each time it is terminated with a center dot. The repetitions are stopped whenever the command is terminated by a CA. 3o5

Append 3o6

[to] ADDR CA $(*[from]* ADDR CA TEXT (CA / C.)) 3o6a

Break statement 3o7

$(*[at]* ADDR CA LEVADJ (CA / C.)) 3o7a

Copy and Move 3o8

text1 *[to]* ADDR CA $(*[from]* ADDR (CA / C.)) 3o8a

text2 *[to]* ADDR CA $(*[from]* ADDR CA ADDR (CA / C.)) 3o8b

strc1 *[to]* ADDR CA $(*[from]* ADDR CA LEVADJ (CA / C.)) 3o8c

strc2 *[to]* ADDR CA $(*[from]* ADDR CA ADDR CA LEVADJ (CA / C.)) 3o8d

Delete 3o9

(text1 / strc1) $(*[at]* ADDR (CA / C.)) 3o9a

(text2 / strc2) $(*[at]* ADDR CA ADDR (CA / C.)) 3o9b

Insert 3o10

(text1 / text2) $(*[at]* ADDR CA TEXT (CA / C.)) 3o10a

(strc1 / strc2) *[at]* ADDR CA $(LEVADJ TEXT (CA / C.)) 3o10b

Move (see Copy) 3o11

Replace 3o12

text1 $(*[at]* ADDR CA *[by text?]* 3o12a

File name recognition in TNLS works as in DNLS now. MSC 3s

File names in TNLS should be terminated by a command accept, as in DNLS. 3s1

3s2

File name recognition will not be done; ↑F and ALT do not cause field or name recognition. The OLD FILE / NEW FILE messages are not printed. 3s3

In other wrds: If your file's name is in the most useful form:"Name.NLS;#", to load it you need merely type out all of NAME (without the period) and then give a command accept. The system will then give you the most recent version in time, regarddless of version number. If your file has an extension other than "NLS", you have to type out all three fields, or else type "NAME" followed by Altmode. Note that the full name does not appear as an echo, but altmode does earch out the file. If you want a version other than the most recent you have to type out all three fields.
Likewise inputting out a file, if you type any string up to 29 chracters but excluding periods, NLS will name the file you are putting out to "STRING.NLS;#". If you want to output to a specific extension or to a version other than the most recent, you must type out all the fileds with ocrrect punctuation.
Print files work in the way just described except read "TXT" for "NLS". 3sh

JUMP BACK now works as advertised. MSC 3t

JUMP FILE RETURN ...AHEAD no longer loop. MSC 3u

VIEWSPECS IN FROZEN STATEMENTS 3v

Frozen statements display now observes the viewspecs attached to the statement when it is frozen. MSC 3w

The character set define mode of Execute Viewchange is limited to defining any character as a control character. 3x

Changes in NLS on or about 6/1/71

CONTROL CHARACTEERS IN TNLS 4

Control characters to be entered in literals in TNLS must be prefaced by the literal escape character (↑V). 4a

ERROR MESSAGES 5

Error messages now will appear above the command feedback line instead of in the middle of the screen. Messages that are only displayed for a few seconds will no longer force the user to wait. Instead the user may continue working even while the error message is still being displayed. 5a

Y VIEWSPEC 6

Now it works the way it did in the good old days. 6a

JOURNAL CHANGES WSD 7

Several implementational changes have been made in the JOurnal system. 7a

Functionally, it has not changed. 7b

(1)Use of holding files for JCAT and CNUMBERS 7c

Two new files, TJCAT and TCNUMBERS have been introduced. 7c1

The Journal uses these in the same way wich it proviiously used JCAT and CNUMBERS. 7c2

The JCAT and CNUMBERS files may be updated to reflect the information held in the working files by the command: 7c3

'Execute 'Katalog Klean-up (Password) CA. 7c3a

The password is the same as that for Journal Hard Copy Delivery, JPD. 7c3b

(2) The Hard Copy Distribution has been modified to reflect the new output processor changes. 7d

The flow in printing a document is roughly: 7d1

(a) First Copy 7d1a

Output processor of document to file dpntwrk.nls 7d1a1

Insert halt directives into header following address field. 7d1a2

Output processor to file HPNTWRK.NLS 7d1a3

This file contains only the address portion of the header. 7d1a3a

compute address length (size hpntwrk-1) 7d1a4

goto (c) 7d1a5

(b) successive Copies 7d1b

Insert halt directives into header following address field. 7d1b1

Output processor to file HPNTWRK.NLS 7d1b2

(c) copy contents of hpntwrk and dpntwrk to LPT 7d1c

Up to 10 copies of a document (or documents) are printed without closing the printer. 7d2

(3) Expedite now causes only the addressed copies of expedited documents to be printed. The collection copies are printed long with the normal. 7e

Changes in NLS on or about 6/1/71

<JOURNAL>7059.NLS;1, 1-JUN-71 15:48 DVN ; (Expedite) Title: Author(s): Dirk H. Van Nouhuys/DVN; Distribution: Marilyn F. Auerbach, Walter L. Bass, Roger D. Bates, Mimi S. Church, William S. Duvall, Douglas C. Engelbart, Beauregard A. Hardeman, Martin E. Hardy, Fred P. Hocker, J. D. Hopper, Charles H. Irby, Mil Jernigan, Harvey G. Lehtman, John T. Melvin, Jeanne B. North, James C. Norton, Cindy Page, Bruce L. Parsley, William H. Paxton, Barbara E. Row, Ed K. Van De Riet, Kenneth E. Victor, Richard W. Watson, Don I. Andrews, Dirk H. Van Nouhuys/MFA WLB RDB MSC WSD DCE BAH MEH FPH JDH CHI MEJ HGL JTM JBN JCN CXP BLP WHP BER EKV KEV RWW DIA DVN; Keywords: $NLS syntax TNLS comands ; Clerk: DVN; Origin: <VANNOUHUYS>6/1NLS.NLS;1, 1-JUN-71 15:33 DVN ; Status of NLS

WHP CHI WKE MSC DVN JCN JCN 1-JUN-71 16:51 7060

Transcription of discussion on features in PDP-10 TODAS, 1 February 1971; WHP CHI WKE MSC DVN JCN HAL VDB were present

(whp) Basically, TODAS is made a lot more like NLS from a CRT with the idea that csp pointing to a particular character in a particular statement, and you use that as a means of making selections that are now made with a cursor. 1

(dvn) What's happened to Alter? 2

(whp) Alter has been temporarily left around as an Execute Edit command and works in somewhat the same way. It turns out the control characters are rather nasty on the PDP-10 with the time-sharing system and that was never a very satisfactory approach anyway. 3

(wke) Do you want to back up to the enter portion of it? How do you get into TODAS? 4

(dvn) How do you enter TENEX? 5

(wke) Let's not get into TENEX. 6

(chi) First of all, there is no TODAS. Once you're talking to the EXEC then you just say, NLS, execuse me, it will ask for your device first then your initials. It will assume the user name under which you are entered. 7

(wke) Good, and then you're into it? 8

(chi) (Confirms) And then you're into it. 9

(whp) If the device is a display, you can go into display NLS, you can go into a TI Terminal, or Execuport, 37. 10

(dvn) (TI) is the command 11

(whp) you just put 'T' -- Termi-Net went away, thank goodness. 12

(chi) Once you're in the command structure for structural entities it's sort of the same for the user. 13

(wke) You do 'Load Files'? 14

(chi) Right. Load File, Output File, Update File. 15

(wke) You can go in cold and insert a statement? No. When you first go in you're looking at ... 16

(chi) Okay, back up a second. 17

Transcription of discussion on features in PDP-10 TODAS, 1 February 1971; WHP CHI WKE MSC DVN JCN HAL VDB were present

(whp) You're never in a situation like the 940 where you're there without any file. There's always some file you're looking at. If you haven't specified one, like if you're just going cold into NLS, there's a default file that will either be created for you or if it already exists and will be loaded automatically. 18

(chi) It is currently given the name of your initials like NLS. I might suggest that that be your file directory, for example, because every time you go into NLS that's the file you're going to be presented. If you didn't have one, if you've deleted it or if it's the first time you've even used it, then it will just create another file. And if you don't change it, it will always be identical. 19

(dvn) If you put something in it that will be something you'll get whenever you go in? 20

(jcn) Is that going to be the MAIL system? 21

(chi) It could be used for that. 22

(whp) There is no longer the idea of having a working copy which is a complete copy of your file. Instead it's like, here's the file you're working on, but between you and that file here is a partial copy and you see the files through this partial copy and anything you've changed is represented in the partial copy, so you see the changed version instead of the file itself. 23

(chi) But somebody else could be looking at the same file and his would be the original file without your changes. 24

(dvn) Through his partial copy. 25

(whp) Not through his partial copy because the system does not want to let two people be editing the same file at once because that leads to nasty problems when both try and update. So the system in face when you start to edit the file (tape interrupted)... When someone else tries to make a partial copy for it, they will receive the message this file is currently locked by user such-and-such on console such and such. 26

(dvn) But he can look at it. 27

(whp) He can look at it. 28

(dvn) He can read but not write. 29

(whp) Exactly. 30

(jcn) He can't take a copy of it away, try to do something else, and put it out on a different name? 31

(chi) He could load it and do an output File to another file without changing it and then modify the other one. But then you'd know that you're doing something deceptive. 32

(wke) But any number of people can look at it, and it's not locked until the first guy tries to change it. 33

(chi) That's right. 34

(whp) And, if the first guy changes it and output to a new version, so that that will protect it ... 35

(chi) you cannot modify a file unless it is the most recent version of that file. So if you're looking at an old version, you can't change it. 36

(whp) The point there is that if two people are reading it at the same time and one guy modifies it and outputs to a new version, it's no longer locked for him. So that this guy could then lock it if you didn't have this sort of protection. Instead you want to force this guy to have the most recent version before he starts editing. 37

(wke) That's NLS? 38

(whp) We've got some interlocks like that ... 39

(wke) He can, however, take that old version and create a new file out of it. 40

(whp) Oh, yes. 41

(jcn) Does that mean you can be looking through a file and all of a sudden, if someone else has updated it who had control of it first, all of a sudden you're under pressure ... 42

(whp) No. False. Because there are different versions of files, that's another thing. 43

(jcn) You really have to consciously go to that different version to get it. 44

Transcription of discussion on features in PDP-10 TODAS, 1 February 1971; WHP CHI WKE MSC DVN JCN HAL VDB were present

(chi) In many ways different versions are really different files in the fact that they have the same name. 45

(wke) How do you know that there is a more recent version? 46

(dvn) You can't write a file, and there must be a reason. 47

(chi) It'll tell you why you can't write on it. 48

(whp) There's no way for us at this point to say, give you a message from someone who creates a new version of the file. That's very complicated internally. 49

(wke) But if you try to load it again ... 50

(whp) You would get the newer version. 51

(wke) Like if you decide you want to make some changes in it, if you do a new Load, then you might get the newest version. 52

(chi) If you don't specify the version of it, it will be ... 53

(whp) There are still some rough edges with respect to the interface with the time-sharing system for loading files and outputting files. That's been probably the most difficult area of the entire conversion, and it'll take some time before we get especially familiar with all of that and to get things really smoothed down. 54

(chi) For example, for a while you'll probably have to type in on an Output File the whole name of the file, even if you're outputting to the same file, or enough of the name so the system recognizes it. You can't make the assumption ... 55

(wke) Okay, so we got into it and you got your file. Now what do we do? You start in structuring. 56

(whp) As we were saying, the editing was much more like editing on a display system. Instead of making bug selections, you're moving a pointer around. 57

(dvn) Let me ask this. On Groups, Plex, Branches, Statements, are Move, Copy, Delete, Replace all the same? 58

(whp) I can't answer that because I don't know how they were before. (laughter) I assume they are the same; they're the same as they were on the display system. 59

Transcription of discussion on features in PDP-10 TODAS, 1 February 1971; WHP CHI WKE MSC DVN JCN HAL VDB were present

(chi) They're the same with the exception ... 60

(dvn) Replace is the same. 61

(chi) Replace--I don't know if you could do this on the old system; no, you couldn't. You can now replace a structural entity by selecting another structural entity. I don't think you could do that. 62

(dvn) No, you could not do that. 63

(chi) You could in NLS, and in general most things that you could not do in TODAS and could in NLS you can now do in TODAS. 64

(whp) It might be worth while to talk about how you make selections first. 65

(dvn) Yes 66

(whp) and then go from that end to how that's used. Do you want to run down Get At? Essentially there's one routine. 67

(chi) There's one routine that does all of the selection. 68

(whp) Both for structural editing and text editing. 69

(chi) If you type a command accept, it terminates the address specification. You can type a space simply for clarity and later reading it or for your own things and it does not mean ... 70

(dvn) space in the middle of an address. 71

(chi) and it does not mean, except at the end of the name or something. If you're saying s space d space. 72

(whp) Occasionally you have to use it to separate the fields or something. 73

(chi) The period talks about the current position of the current statement. But does not point strictly to a statement, but points to a character position within the statement. At any time you're where you could give a command you could type a point and it will tell you where your current pointer is in terms of the statement number and a left paren and a character number, right paren. 74

(dvn) It count characters and says, character 50 or something. 75

Transcription of discussion on features in PDP-10 TODAS, 1 February 1971; WHP CHI WKE MSC DVN JCN HAL VDB were present

(chi) Okay, you type an S for Successor, a P for Predecessor, D for Down, U and Up, an H for Head, a T for Tail, E and End, N and Next, and B for Back. 76

(dvn) What's "Back"? 77

(chi) Back is the opposite of Next. 78

(wke) ignoring structure 79

(jcn) It is not Return. 80

(chi) If you were looking at a display ... An ↑ (up arrow) gets you a link, and I wouldn't suggest doing that for a bit. 81

(wke) Gets you a link? 82

(chi) Yeah. 83

(whp) A jump link. 84

(dvn) Space up arrow? 85

(chi) I don't see why you need a space. 86

(wke) What does it do, pick the next link in the statement? 87

(chi) No. At any time you do this you give, for example if you're talking about a link, you could say at the current position, wherever my pointer is, I'm pointing at a link. 88

(dvn) You don't need to say, point ↑ (up arrow) to go to the current Link? 89

(chi) Yeah, if you wanted to use the current position in the pointer, yeah. 90

(dvn) So your pointer is in a link. 91

(chi) you use the pointer here exactly the way you use the cursor in NLS. You point to a character in a statement and you say that's somewhere in a statement... You can type an & (ampersand) for File Return; I would also suggest that you not do that for a bit. And a : (colon) for a name. There are two new things. If you type a [(left square bracket) and some string, it finds that content and positions the pointer right after the content, from wherever it happens to be at that time. 92

Transcription of discussion on features in PDP-10 TODAS, 1 February 1971: WHP CHI WKE MSC DVN JCN HAL VDB were present

(wke) points to the character following the content. 93

(dvn) When it gets to the end of the file it will go back to the beginning and search through? 94

(chi) No. 95

(dvn) If you start in the middle of the file. 96

(whp) Right. You are searching for the next one. If you say this brackets thing directly following by an "F", then it starts at the top of the file. 97

(chi) If you type a > (left angle bracket) followed by a string followed by a < (right angle bracket), then it will search for a "word" that is that string. If that string is preceded by or followed by a character, number or anything, then it won't work. 98

(wke) It has to be a word. 99

(dvn) You mean preceded and followed by spaces. 100

(chi) Right. 101

(whp) Not necessarily spaces, non-letter digits. 102

(chi) If you type $ (a dollar sign) and a string, then you're talking about a marker, the name of the marker. 103

(dvn) Typed inside the square brackets. 104

(chi) No, this is a separate command. 105

(chi) you can type a + (plus) or a - (minus) and then a number and then a W for Word or C for Character or a V for Visible or an I for Invisible. And if you don't type any character at all, then it will assume character. And that moves you relating to wherever you are. 106

(wke) +3W will move you forward three words. 107

(chi) That's right. 108

(jcn) Good. 109

(wke) Yeah. 110

Transcription of discussion on features in PDP-10 TODAS, 1 February 1971; WHP CHI WKE MSC DVN JCN HAL VDB were present

(dvn) Groovy. 111

(wke) Wow. 112

(chi) And that's basically it. 113

(whp) It doesn't go off the end of the statement obviously. If you say something that's bigger than your statement you just go to the far end of the statement in whichever direction you've said and just sit there. 114

(chi) That's with the plus and minus the scan will go across. 115

(wke) You mentioned markers, did you say how you set markers? 116

(chi) Yeah. There's a command called Fix, Fix Marker Name, and you specify the name and the position where you want it to move. 117

(wke) How do you do that? 118

(chi) Well, in specifying the position, then you use any combination of these things. 119

(wke) Position first. What's the syntax of that? 120

(whp) We've tried to make the commands give ... 121

(chi) Yeah, the commands tell you what they're after. 122

(whp) tell you want to do; they say 'At', or 'By' ... 123

(wke) So you would say, Fix. F. 124

(whp) You type F, and it will say "Fix Marker Name space" and wait for you to type the literal, you type the literal 125

(chi) and it says "At" and you type some expression that says where you want it to be, which can be a statement name plus three words plus this content. 126

(wke) Very nice. 127

(dvn) What about Substitute? Is it there? Gone away? What? 128

(chi) Substitute is there and to the user will be basically the same as it was. You still substitute over a structural entity or string or ... I haven't even looked at that. 129

Transcription of discussion on features in PDP-10 TODAS, 1 February 1971; WHP CHI WKE MSC DVN JCN HAL VDB were present

(whp) Alter you still type statements numbers as a final thing. 130

(chi) and we're also going to put in some representation for SID's which are a permanent identifier for a statement. If you take the statement out of the file and put it in another file, you lose that SID. 131

(dvn) What's the difference between that and the statement number? 132

(whp) The SID is something that is assigned to statements whether they have names or not. 133

(dvn) All statements have one. 134

(whp) All statements have one. 135

(chi) But you could move the name of a statement to another statement. You can't move the SID. 136

(whp) One thing we've left out, on the address thing, is that the name is first on next ... After you type you say : (colon), the name (door slams) ... f to get the first (door slams) ... At the highest command level we talked about the "point" command. You just type a period and it types out where you are. You type out a slash and that types out the statement where you are, and when it reaches the character position, it types an angle bracket, line feed, angle bracket to show you what character you're actually on. The character you're selecting is the first character of the line after the line feed. 137

(chi) That turns out to be extremely useful. 138

(wke) Oh, yeah, I can see that. 139

(whp) There's also an up arrow command that jumps you to the Back and a line feed command that takes you to Next and prints those. 140

(wke) Statements? 141

(whp) Right. 142

(dvn) Up arrow takes you to where? 143

(whp) It's like a Jump Back command in NLS; it takes you to the statement that you'd see in front of that one if you had a list. 144

Transcription of discussion on features in PDP-10 TODAS, 1 February 1971; WHP CHI WKE MSC DVN JCN HAL VDB were present

(wke) You can move your pointer back also? 145

(whp) And it moves your pointer back. We tried to be careful that we moved the pointer around in a reasonable way. 146

(wke) In that case you can, would move it to the beginning of the statement you just ... 147

(whp) Whenever you move it to the new statement, it moves to the first character of the statement. 148

(wke) The first character of the statement is the first character not counting the statement number. 149

(chi) Right. The statement number is not part of the statement. 150

(whp) There's also a command that simply lets you position the pointer some place else. You type 'space' and then some string that positions the pointer just like this addressing service. 151

(wke) Sounds like a very nice system. 152

(whp) Yeah, it's nice. 153

(whp) In addition to that, the idea is that you're going to be using these word searches or content searches to move around through the file to a large extent. In order to avoid having to type that in again, if you want to repeat a search, the system remembers the last search you did. If you are at the at the highest command level in alt mode, it simply repeats that last search again on jump to link. 154

(wke) How about that. 155

(dvn) Groovy. 156

(whp) Say you've searched for the word "something" and you hit an alt mode, it actually echoes the entire command just like you typed it in so you see what you've done. So you can do an alt mode, slash; alt mode, slash; and jump around and see it. 157

(wke) What's the slash do? 158

(whp) The slash is printing. The Alt Mode to jump to it, the slash to print it. 159

(wke) I see. 160

Transcription of discussion on features in PDP-10 TODAS, 1 February 1971; WHP CHI WKE MSC DVN JCN HAL VDB were present

(chi) The slash is a Print Statement with indication of where the pointer is. The other commands are basically like NLS, with a few exceptions. 161

(wke) If I specify an address of a place I want to go to I can search in brackets. If I can print that ... with a slash. 162

(chi) Right. 163

(dvn) Or it will go to it silently and wait for your command. 164

(wke) Then the only way I can go to find things or find the occurrences of the word in the "computer". Do I always have to get that line feed in the middle of it? 165

(dvn) No, you can go to it and still do Print Statement, just to get the statement; but you won't know where you are in the statement, except you will know, in your mind. 166

(wke) So you could do an alt mode, print statement, and where does the print statement take you? 167

(chi) Exactly where it was; print statement and some expression which can be the pointer, command accept assumes point. 168

(wke) So the alt mode, ps, point, command accept will print the next statement without the slash. 169

(whp) It would have to be ps, command accept, command accept; where the first command accept terminates the string, saying where you wanted to be. 170

(wke) You mean if you hit a command accept, the point is a default. 171

(chi) If you haven't done anything, if your address specification ... 172

(whp) No, Bill is right. You just hit a command accept. Your address specification starts from the current point, so if you just hit a command accept then that's it. 173

(chi) The commands that were in TODAS are basically the same now with the exception of this added addressing capability. Mostly they dealt with structural entities; now we also have the same commands for textual entitites. So that instead of just moving a

statement or branch you also move words, or text, or visible or whatever you want. 174

(dvn) And they are called W or V or I or T? How do you specify the words that are transposed? 175

(chi) You use the same expression. You could say, for example, move .-2word .-3 to +3. 176

(whp) There are also markers; it's easier to fix markers. 177

(wke) The marker can be any type string? 178

(chi) Up to four characters. 179

(wke) Four characters only? 180

(whp) Five. 181

(dvn) When you say move .+3 to .-3 is .+3 a word or character? 182

(whp) you'd have to say "W" if you wanted it to be word; if you don't say anything, then it assumes it to be character. 183

(jcn) Content Analyser--did you get to talk about that yet? 184

(chi) No, that will be basically the same. You specify where you want the bug selection to be with the same expression we were talking about. The Execute Content Analyser. You can do any of these things in two different ways depending on how you think. You can set up the pointer first and then give the command, or you can give the command and give the expression except for the pointer. 185

(chi) This system is in some ways biased toward 15- and 30-character per second devices because it does things like give you a carriage return when you're starting something new. For example, if you do an Insert Statement and you hit a Control B and do some level adjusts, the first thing it does when you level adjust is give you a carriage return so you start fresh, and on a ten-character device that might be kind of sloppy. People are not going to want to wait for that carriage return. Also Level Adjust is not terminated only by a command accept and a space now. If you do a U D DD or something like that, and start typing as soon as you hit a non U/D it will assume you're starting to type text. 186

Transcription of discussion on features in PDP-10 TODAS, 1 February 1971; WHP CHI WKE MSC DVN JCN HAL VDB were present

(whp) The idea there is to avoid a situation in which you're merely typing away and are doing nothing but having the system look for U's and D's. 187

(dvn) I've had that happen. 188

(chi) Are we going to put the statement number stuff in for paragraphs? I think there's going to be a global switch that you can set using Viewchange and that will determine whether or not when you're doing level adjusts if you get statement numbers generated. That is a very, very costly thing for the system to be doing, and if you want your version of NLS to go fast, take the numbers out; and if you want it to go slow, and want the statement numbers in, do it. 189

(dvn) But you haven't implemented it? 190

(chi) The way it is right now it does not do statement numbers. 191

(wke) Does it do it when you first do the ... 192

(chi) It doesn't do it at all. 193

(wke) Leave it that way for a while. People will work with it that way for a while before you spend much time on anything else. 194

(chi) Well, Bill Duvall has made a request that we change it around. 195

(jcn) That's the default? 196

(chi) For him, yes. 197

(dvn) I'm thinking of the naive user. The old way would be better. 198

(wke) It's easy to get your statement number printed, though. At any time when you want to know where you are as far as a statement number, just a period and it types it. A period, insert statement, and you'll know exactly where you are. 199

(chi) The trouble with that, of course, is if you're doing center dots you don't have the option of saying where you are. We could put that in. 200

(whp) But the point is being moved around with the last inserted entity... You say point and that tells you the last one you did,

then you say insert statement, command accept and you're back right where you were again.(chi) Do you want to go through and I'll tell you which commands have been added to the other TODAS commands? 201

(chi) First of all, Join has been changed to Append.

There's a Break Statement which the old TODAS didn't have.

Copy and Delete are still the same except your commands are textual entities.

Execute

Content Analysis

Declare File Ownership

Execute Edit 202

(whp) That's still Alter. The control characters get changes around. Control C in TENEX is what the Rub Out was to the 940. Control T to TENEX gets gobbled up at the very lowest level of input returns which causes it to type out something that says what your process is doing right now. Let 'it's running at location such and such.' 203

(chi) The Control T turns out to be a handy thing. If you don't know if your program's waiting for you to type a character, then you just type a Control T, and it would say it's an I/O wait and you would type a character. Or if you don't know how many command accepts to do ...

File Verify 204

(whp) The Execute Verify replaces the file clean up. It's a much faster version and a lot cheaper to use and does not modify the file at all; it simply is a read only sort of thing. 205

(dvn) you read your file, and it has 45 bad characters; then what? 206

(whp) No, it doesn't look at bad characters. There's no way in the system right now to make the system go through automatically and read out bad characters for you. That could be added as a process of some sort. 207

Transcription of discussion on features in PDP-10 TODAS, 1 February 1971; WHP CHI WKE MSC DVN JCN HAL VDB were present

(wke) If Verify finds anything wrong, what does it do? 208

(whp) If Verify finds anything wrong it deletes the file; from your ... you're not looking at it, it sort of throws it out; rejects it. The reason that is done is because, from our experience on the 940, people tend to use file clean-up and if there's anything wrong at all they don't try to clean up that version of the file. Instead they throw that version of the file away and go look for another. 209

(wke) Is there any way to save the file? 210

(dvn) I use file cleanup often. 211

(wke) To save the file? 212

(dvn) Yes. 213

(chi) The theory behind this is with the automatic check pointing making a copy of whatever you're working on every couple of minutes that if you do a file verify and something is wrong it's very cheap to go back. 214

(wke) The thing I've found if you get something bad on the disc and do a clean-up on it and there's something wrong, a bad check sum, a lost stb, the whole file isn't gone by any means. If that's the only copy you have around, I sure wouldn't want to throw it away. 215

(chi) It does not throw it away. You're just not looking at it now; it will give you another file. 216

(wke) Is there any process that will let you do anything to save as much as possible on it? 217

(chi) You can do an Output File on it. 218

(wke) You can do an Output File? 219

(chi) Sure, that would regenerate the whole file. 220

(wke) O.K. 221

(chi) It's just that the file clean-up that we have now is not really necessary. 222

(Wke) So you do a Verify, and if it throws it out, you Load it again and do an Output File and save as much as you can of it. 223

(chi) An Output File is much more thorough than File Clean-Up. 224

(hal) What does the Output File do? What takes up the slack? 225

(chi) There are two kinds of Output File now. First of all, there is the automatic checkpoint, and you can retrieve your checkpoint. Secondly, there is what we call an Update File and at any point when you're working along if you've decided that you've made enough changes to your partial copy and now you want to put that back on the real file and free that file, then you just say, Up Date File, Command Accept and it just maps what you've changed onto the old file, and you can't do that if somebody else is reading the file. You've got to be the only one looking at it. 226

(Wke) Oh, you can't. Then that's going back to the old version. 227

(chi) That's going back to the old version, and you can't write a new version if somebody else is reading it. 228

(Wke) So Update will not make a new version; Output File will. 229

(chi) Output File will. Output File assumes very little about the old file and picks up the statement from it and generates a new file taking this statement... Generally when you do an Output File you do an Output File to a new version, and the old one will exist and as one number less forever. TENEX has a very serious problem there in proliferation of versions. They will eventually get around to implementing a feature that says for this file I want to keep three versions, and it will cycle around with three versions or something like that. Currently they don't have anything implemented like that, and you end up after a while going back and deleting all the old versions. 230

(Wke) The File Update can be a problem if somebody else is looking at it, you can't update it, you're going to be very frustrated. 231

(hal) How do you delete particular versions? 232

(Whp) You couldn't do that now anyway. 233

(Wke) You have to do that now, you have to go around and say,

who's got my file? You say, 'I don't want a new version, what the hell am I going to do now?' 234

(chi) A file with a version number is really a file; that's a whole file of specifications. There's a delete command at the executive level. In generaly TENEX uses an alt mode to name recognition and uses a carriage return to do command terminations. so in general, if you hit an alt mode when you've said, delete file, then it will present you with the oldest version of the file. That's the default. 235

(wke) Really eventually, it would be nice to have a way if you wanted to update a file and somebody else has it, to give him a message. 236

(chi) That turns out to be pretty tough. 237

(wke) I'm sure it is. 238

(chi) But that's what we ought to do. It would also be nice if when you tried to do the update file, you were told who was reading the file that was preventing you from doing that, but there's no way of getting answers through TENEX, until we have time to change... 239

(wke) Got to have something to do in the future. Can't solve it all today. 240

(jcn) Every time you Output File, a new file is created? 241

(chi) No, you don't have to do it, that's by default. You can specify the old version. But the way you specify the old version is somewhat cumbersome in TENEX. To do name recognition the way we used to hit command accepts when you say Output File and you type part of the name, another command accept and it would give you the rest of the name. Well, in TENEX if you do that it will give you a new version. Okay, so the default is not very nice then. What you have to do is hit a control F. There are different fields in the TENEX file name. There's the actual file name, there's the extension, and there's the version. 242

(dvn) What is extension? 243

(chi) The extension turns out to be a very nice thing. For example, if you have a file in different stages like we have code files that contain the text and that would have NLS files maybe and we have partial copy files that are some NLS files and we

have relocatable binary files. Well, they can all be the same filename but different extensions. You can say File X.nls, file x.pc for partial copy, etc. 244

(wke) So in Output File for the old one you have to type the whole thing. 245

(chi) Either type the whole thing or you type enough of the name so that the system will recognize the name and type a control F and it will finish up to that point, and you type enough of the extension so it will recognize that and type a control F, and it will recognize up to that point and then you type the version that you want. 246

(jcn) Are we going to try to redesign that part of it? 247

(chi) We'll eventually have to do something nicer, but for now--that's taken up almost all of the last week trying to get that as good as it is. It's really a drag. 248

(wke) In the meantime the only reason to do Output File is to clean up your file, or to make a new one; otherwise the update will do it. 249

(hal) But Update doesn't clean up your file. 250

(chi) Update does not clean up. Update does not really work. You'll probably find that you do Execute Verify quite often because it's fairly inexpensive. 251

(hal) What happens if something is wrong with the file? 252

(chi) Then you go back to your last checkpoint. 253

(dvn) Then you find out that the something was wrong in the last checkpoint? 254

(chi) Then there's another checkpoint. 255

(jcn) You don't make checkpoints; the system does it. 256

(chi) You can make checkpoints or the system will. Currently we haven't really set the time interval, but it will probably be something like four or five minutes. 257

(wke) How? 258

Transcription of discussion on features in PDP-10 TODAS, 1 February 1971; WHP CHI WKE MSC DVN JCN HAL VDB were present

(chi) you just do output Checkpoint, and you can say Load Checkpoint. If you say Load Checkpoint, command accept, it will load the most recent checkpoint you have. Otherwise you can say Load Checkpoint older or newer; and it will get the older or newer version. 259

(jcn) For that file? 260

(chi) For that file. There are two checkpoints kept for every file that you are currently modifying. 261

(jcn) What if I'm modifying ten files? 262

(chi) Then there's a current checkpoint and partial copy and old checkpoint for every single file. 263

(jcn) Change drum assignment to 2 million? 264

(chi) There's no drum assignment kind of feature here. You get as much as you need. 265

(msc) There's a limit on the number of files. 266

(jcn) How many? 267

(wke) 120? 268

(chi) TENEX as it exists right now will only allow you to have 13 open at once. Every one that you're actually modifying counts as four because there are two checkpoints and a partial copy and an original. 269

(dvn) So that's three files. 270

(chi) So that's three files. That's actively working on them. If you've been working on one and now you're not working on it, you could be working on three others because then we can close some of those files. 271

(wke) Do you close it? When you try to work on another one, do you automatically close it? 272

(chi) The way it is in TODAS, you really only work on one at a time because it's very difficult to do that from a teletype. On the screen you'll be able to work on as many, if you want to set up the windows, within the limits of TENEX. 273

Transcription of discussion on features in PDP-10 TODAS, 1 February 1971; WHP CHI WKE MSC DVN JCN HAL VDB were present

(wke) If I've got three on the screen, 274

(chi) Then you can be actively working on three. 275

(wke) and then go to a fourth one, it's going to close one of those three. 276

(chi) If you do Load File in one of those windows, then it will close the one that was there and open the new one. 277

(dvn) Then you can't have more than three windows? 278

(chi) You can have eight windows. 279

(dvn) But you can't have a file in each of the eight. 280

(whp) That's right. 281

(chi) You could if you were reading out of most of them. 282

(wke) Does closing include a file update? 283

(chi) It closes the whole state of the world exactly the way it was, when you were working on your partial copy and everything. If you do a Load File and you have at some point in time been working on it but haven't done an Update or anything to get rid of the partial copy, then when you load it you get this exactly the same thing. 284

(wke) So it opens all four of those for you. 285

(chi) Well, it doesn't open the checkpoint. 286

(wke) But it opens the partial and the file? 287

(chi) Right. 288

(dvn) Two minutes later. 289

(wke) So you could have been working on one, it closed, and then you could load it and do a file update. 290

(chi) Sure. 291

(wke) We've got a lot to learn about that... How do you get rid of that partial copy? 292

Transcription of discussion on features in PDP-10 TODAS, 1 February 1971; WHP CHI WKE MSC DVN JCN HAL VDB were present

(chi) When you do an Output File or an Update File it eliminates your checkpoint and partial copies. 293

(wke) On an Update File? 294

(chi) Because you've restored the old one to the state that you want. 295

(wke) Eliminates your checkpoints? 296

(chi) That's right. Because the checkpoints are no longer valid. So does Output File. 297

(dvn) It has to be that way. 298

(chi) It has to be that way because a checkpoint is a copy of your partial copy at some point in time, and if you update the original copy the original file no longer ... 299

(wke) What about things if you do this Update File and it hits a bad spot on the disc and the whole thing is gone and you've lost your checkpoint and there you are. 300

(chi) We don't throw the rest of it away until we're done. But the problem is you're changing the original file. The partial copy is just a filter through which you're seeing the original file. If you start changing the original file, then the filter's no good any more. 301

(dvn) It's a map for a place that no longer exists. 302

(wke) I realize that. 303

(jcn) You can set up your own checkpoint by hand, called checkpoint. 304

(wke) It's still a partial copy. You mean name it. Well, that doesn't work either. 305

(chi) We can clean it up if that turns out to be a problem. 306

(dvn) You can always put out the file under a new name. 307

(whp) If that turns out to be a problem we can solve it. 308

(wke) That would depend on the reliability of the devices. 309

(chi) Yeah. So far it's been very realiable. 310

(wke) It's a whole lot clearer this way. 311

(jcn) Are files open whether you've changed them or not? 312

(chi) No, a file is only open if you're actually working on it. 313

(jcn) so a lot of link jumping doesn't have the effect of creating new files. 314

(chi) No. The way it's going to work on the display version is that any file that you have current displayed, some part of it's currently displayed, that will will be open, and you have write access to it, if you have a partial copy for it, then you can modify it. As soon as you take it down, then it's closed for you. 315

(hal) How much space is there? 316

(whp) About an order of magnitude bigger than the 940? 317

(hal) I was wondering about the proliferation. 318

(chi) The disc space is four times. 319

(wke) Yes, four times the disc space. 320

(chi) Each NLS file that you have can be ten times a large. 321

(jcn) In terms of number of statements, or characters, or combination? 322

(chi) Well, the whole thing. 323

(whp) Essentially a combination. I don't expect each statement to be ten times as big as the current statement; but given that they're roughly the same as the ones now, you can have ten times more. Probably you won't want to do that. Probably it will be easy enough to do cross files and to keep your files broken up into more manageable sizes. 324

(dvn) For something like the report when you're doing final put togethers of the report and putting together a book so you can sort through it. 325

(hal) Couldn't you use the extensions? 326

(whp) you could. You still have handles on the system that would allow you to link different files together. 327

(chi) For example, we're planning to do our (I don't know if this will work or not), but we have a cross reference facility on the 940 which we're bringing across to the 10. If you notice the second blue binder, that's our cross reference and we intend to make that one file. 328

(wke) Nice. 329

(chi) That will be pushing it a bit. 330

(wke) Any loose ends anyone wants to know about? 331

(jcn) Syntax for Content Analyzer is the same then all throughout? 332

(chi) I don't think there are any changes? 333

(wke) And Executable Text will work just like before. 334

(chi) There's no Executable Text at this point. 335

(wke) I see. 336

(jcn) Analyzer Formatter is now L10. There is no Executable Text yet; there will be? 337

(whp) We haven't decided. 338

(wke) It's on my schedule. You saw my schedule; I talked to Bill Duvall about it. 339

(jcn) There's no point in talking about Quick Print, or PASS4 or things like that? 340

(wke) Oh, yes, let's talk about that. 341

(chi) Quick Print's basically the same as it was except it prints out an elaborate heading for you now telling you the person who did the Quick Print and the time that he did it, level, clipping, line truncation if set to something other than "all", it tell you what that is. 342

(jcn) You don't have any control over what it does print up there, like suppress all that junk. 343

Transcription of discussion on features in PDP-10 TODAS, 1 February 1971; WHP CHI WKE MSC DVN JCN HAL VDB were present

(dvn) And PASS4 is not up yet? 344

(chi) PASS4 you've got to talk to Bruce Parsley about. 345

(jcn) That's three or four weeks off. By that time we'll probably have NLS. 346

(chi) There's an Insert Sequential which replaces Insert QED. 347

(jcn) And it will do all the things Insert QED did when we have the console. 348

(chi) Yeah. Load 940 Files. There's a process which Dave Hopper is writing which retrieves a 940 random file from a KDF tape or archive tape and puts it into 10 file space. Then you can go into NLS and do a Load File on it just like you would do on a normal 10 NLS file except you say, Load 940 File, and it'll do the translation into the 940 format. You can also do an Insert Sequential. That doesn't make any difference. Isn't there also some kind of an Insert 940 also? 349

(whp) You can do the same thing alright. 350

(wke) Is there a merge command? 351

(chi) Merge has not been put into NLS yet; it should be very trivial. 352

(jcn) When it is it will be like what we have. 353

(chi) It will probably be more like Copy. I'd like to make it more like a Copy Branch or something like that than a Merge. 354

(whp) You'd say Merge Branch or something like that? 355

(chi) something like that and it just does a filtered copy. 356

(jcn) Where do we find out what the Collector/Sorter is going to be like? Is Duvall writing something? 357

(chi) He wanted to put that in as two commands in NLS, and I don't know if that's going to happen or not. He didn't want it to be a separate pattern. 358

(chi) What wonderful things have we left out? You now say, to do the Analyzer/Compiler sorts of things you say, 'go to L10', and

to execute your program you say, 'EXecute Program', which is sort of the opposite of the way it used to be. 359

(Wke) Go to L10, that compiles a program that the pointer's pointing to. to? 360

(Whp) We're getting a little esoteric. 361

(chi) Right. 362

(wke) Yeah. 363

(jcn) Does this mean we're going to be able to build a calculator in TODAS eventually? 364

(whp) There's no reason why not. 365

(chi) There's also a Set Command. 366

(wke) What's a Set Command? 367

(chi) Upper and Lower and ... 368

(wke) Oh, that. 369

(jbn) Upper case? 370

(chi-whp) Ummm. Don't know about that. 371

(jcn) We'll go along with the same 'Set Modes'. 372

THE END 373

WHP CHI WKE MSC DVN JCN JCN 1-JUN-71 16:51 7060

Transcription of discussion on features in PDP-10 TODAS, 1 February 1971; WHP CHI WKE MSC DVN JCN HAL VDB were present

<JOURNAL>7060.NLS;1, 1-JUN-71 17:14 BER ;Title: Author(s): William H. Paxton, Charles H. Irby, William K. English, Mimi S. Church, Dirk H. Van Nouhuys, James C. Norton, James C. Norton/WHP CHI WKE MSC DVN JCN JCN; Keywords: ; Clerk: BER;
Origin: <ROW>TODAS.NLS;3, 1-JUN-71 16:22 BER ;

WHP 1-JUN-71 18:07 7061

Statement Property Lists

This outlines an approach to provide a property list data structure with each statement in an NLS file.

Statement Property Lists

Syntax of data structure for statement property list 1

(spl) statement property list = data list 1a

(dl) data list = 1b

A data branch which may optionally have a data list linked to its right 1b1

(db) data branch = 1c

Statement data block (called the top of the branch) which may optionally have a data list linked below it 1c1

(sdb) statement data block = 1d

A header, containing links and other fixed format information, contiguous with a variable sized block of data 1d1

(stid) statement identifier = 1e

specifies a particular ring element in a particular file 1e1

(stdb) statement data block identifier = 1f

specifies a particular statement data block in a particular file 1f1

Each branch of the statement property list is called a property 2

One of the fields in the data block header specifies the "type" of the block 3

The type of the top of each property is assumed to be different and is used as the indicator (or name) for the property 4

The actual structure will be a doubly linked list of sdb's. 5

Each sdb will have a "right" pointer, a "down" pointer, and a "back" pointer. 5a

If there are no branches to the right, then the right pointer is zeroed. 5b

If there are no branches below, then the down pointer is zeroed. 5c

If the sdb is the head of the spl, then its back pointer

points to the ring element and a flag (sxflag) is turned off. This flag is turned on in all other sdb's. 5d

If the sdb is the head of any other list, then its back pointer points to the sdb above the list. 5e

If the sdb is not the head of a list, then its back pointer points to the sdb to its left. 5f

These conventions are compatible with the current file structure. In other words, it will not be necessary to convert files. 5g

There is room available in the sdb header for the two new pointers (the back pointer is already there), the flag, and the type field. 5g1

These fields are all zeroed in existing sdb's. 5g2

The following procedures will be provided 6

Statement property list procedures 6a

Get statement property list -- getspl(stid) 6a1

This function returns the stdb of the start of the list or a 0 if the statement has no list 6a1a

Store statement property list -- stospl(stid, stdb) 6a2

Changes the ring element (stid) to point at list (stdb) 6a2a

Copy statement property list -- copspl(source, dest) 6a3

Source and dest are stid's 6a3a

If dest has a statement property list it is deleted 6a3b

Dest gets a copy of source's spl 6a3c

Store property -- stoprop(stid, stdb) 6a4

Make the branch (stdb) the current property for statement (stid) of type given in the header (of stdb) 6a4a

If there was already a property of that type it is deleted 6a4b

Get property -- getprop(stid, type) 6a5

Returns the stdb for the property of that type or 0 if the statement has no property of that type 6a5a

Delete property -- delprop(stid, type) 6a6

If the statement (stid) has of property of that type, the property is deleted 6a6a

Data list procedures 6b

Copy data list -- copdl(stdb, fileno) 6b1

Makes a copy of the list starting at stdb in the file specified by fileno and returns the stdb for the copy 6b1a

Delete data list -- deldl(stdb) 6b2

Deletes the data list starting at stdb and fixes up the structure if the links from the list are nonzero 6b2a

Insert data list to right -- indlrt(old, new) 6b3

Old and new are stdbs 6b3a

The list starting at new is inserted into the structure to the right of old 6b3b

Insert data list down -- indldn(old, new) 6b4

Old and new are stdbs 6b4a

The list starting at new is inserted into the structure at the head of the list below old 6b4b

Data branch procedures 6c

Copy data branch -- copdb(stdb, fileno) 6c1

Makes a copy of the branch starting at stdb in the file specified by fileno and returns the stdb for the copy 6c1a

Delete data branch -- deldb(stdb) 6c2

Deletes the data branch starting at stdb and fixes up the structure if the links from stdb are nonzero 6c2a

Remove data branch -- remdb(stdb) 6c3

Removes the branch at stdb from the structure and zeroes its links but does not delete the branch. The branch may then be inserted at a new location. 6c3a

Statement data block procedures 6d

New statement data block -- newsdb(size, fileno) 6d1

Returns stdb for a new block in the specified file. Size must include room (sdbhdl words) for the header 6d1a

Copy statement data block -- copsdb(stdb, fileno) 6d2

Makes a copy of the block in the specified file and returns stdb for the copy 6d2a

Load statement data block -- lodsdb(stdb) 6d3

Loads the block from the file if it is not already loaded and returns the page index for the page into which the block is loaded (this index may then be used to "freeze" that block at that address while it is accessed) and the core address of the start of the block. 6d3a

Statement Property Lists

<JOURNAL>7061.NLS;1, 1-JUN-71 18:08 WHP ; (Expedite) Title: Author(s): William H. Paxton/WHP; Distribution: Walter L. Bass, Mimi S. Church, William S. Duvall, Douglas C. Engelbart, J. D. Hopper, Charles H. Irby, Harvey G. Lehtman, John T. Melvin, Bruce L. Parsley, Kenneth E. Victor, Richard W. Watson, Don I. Andrews/WLB MSC WSD DCE JDH CHI HGL JTM BLP KEV RWW DIA; Keywords: propertylist statement data structure file list; Clerk: WHP;
Origin: <PAXTON>SPLDOC.NLS;8, 1-JUN-71 15:03 WHP ;

Memo, D. C. Engelbart To Cordell Green, 2 JUN 71 5:16PM:
On Large, Discipline-Oriented Data Bases for On-Line Network Use

Sent to Cordell, 2 June 71, with the four enclosres cited:
(5220,) (5773,) (5774,) (7014,).

Memo, D. C. Engelbart To Cordell Green, 2 JUN 71 5:16PM:
On Large, Discipline-Oriented Data Bases for On-Line Network Use

Cordell: 1

You have asked me to record some of my thoughts and plans that bear relevance to some suggestions and possibilities (that have increased in number and seriousness since ARPA's Contractor's meeting at San Diego), pertaining to such as: 1a

A Computer Science Library (e.g., see McCarthy (5773,)) 1a1

An Encyclopedia of Computer Science (e.g. Mc Carthy (5774,)) 1a2

An On-Line Professional Journal -- e.g. for Artificial Intelligence; a journal that is composed, reviewed, and published with computer aids, by parties distributed about the world (but particully, by active parties on the Network). 1a3

I use the terms, "discipline-oriented data base for Network use." to refer to these types of possibilities. 1b

Special consideratons, relevant to the On-Line Computer-Science Proposal: 2

There are a number of questions and techniques that I think need R&D work before it would be feasible to design, assess, or implement the system to support the conversion of library material and its significant subsequent usage: 2a

(1) What form of digital encoding should be used for describing special graphical constructs that assumedly ought to be digitally encoded? E.g. formula, graphs, line diagrams, tables, etc. 2a1

(2) What form of digital encoding should be used for describing such typograhical parameters as type fonts, page-composition geometry, etc. that would be of likely value? 2a2

(3) What provisions can (should) be made to accommodate users whose terminals fall at various points along the spectrum of sophistication and speed? 2a3

(4) What provisions can (should) be made for a remote user to produce a hard copy replication of material retrieved

from this digital store? Speed, resolution, cost, and accessibility all need to be appropriate. 2a4

It would be an unusual situation for which every deserving user would be able to do all of his serious studying at a console. 2a4a

(5) What sort of cross-reference conventions to establish, and how to record and implement their usage? 2a5

For instance, although cross references within the journal articles are usually relatively unspecific (e.g., just to the article, occassionally to a page), subsequent computer usage would like to provide referencing exactly and explicitly to any entity in the symbol structure -- to any text passage (a character, word, string, paragraph, or section, ...), or to an expressioon within an equation, or to a part of a diagram or photograph, or etc. 2a5a

(6) How might be handled such as photographs or other graphic components of the text that are deemed impractical to encode digitally but yet important to carry along. 2a6

Video-like scan signals can be digitized and stored, of course. If the full range of dot resolution, grey scale, color, were encoded, there would be a huge bit load for some publication items. 2a6a

Would this approach be feasible? 2a6a1

If not, what other means of digital encoding could be considered? 2a6a2

Is the equipment available to store photographs etc. in photoform, with associated means (probably parallel to the corressponding digital means) for access, scanning, video transmission and switching, viewing remotely, etc.? 2a6b

(7) What sort of production system would be required for the transcription? 2a7

How much checking, verifying, backup storing, etc.? 2a7a

Memo, D. C. Engelbart To Cordell Green, 2 JUN 71 5:16PM:
On Large, Discipline-Oriented Data Bases for On-Line Network Use

What kind of organization to handle the bulk, keep it under control? 2a7b

A computer-aided system for management and control? 2a7b1

Could it feasibly be partitioned into activities that separate organizations could handle? 2a7b2

Might it be of any significant value to connect between such organizaions via a computer network? 2a7b3

What kinds of equipment, techniques, special training, etc. would be involved? 2a7c

E.g., could OCR equipment be of significant help? 2a7c1

Can automatic page-turning devices be used? 2a7c2

Who could set up and manage such a system? 2a7d

(8) Whose subject-cataloging system to adopt, for bibliographic control and retrieval? 2a8

Who will do the classification (What the librarians call "cataloguing"? 2a8a

(9) What structure and organization to give to the bibliographic accessing and retrieval data? 2a9

Until questions such as these are resolved, it isn't appropriate to ask whether or not IPT should commit itself to converting a discipline's library and making it available on line to a significant community. 2b

A more appropriate sort of question might be, "Should IPT support a conditionally successive set of specific goal-oriented tasks aimed at: 2c

(1) Studying the possibilities of pursuing such a program; and developing a set of feasibility criteria toward which specific set of R&D projects could be directed. 2c1

This project would have good intrinsic value, even if further action were aborted/deferred. But to be worthwhile, it would have to be driven by a full-time, dedicated guy who is a good manager and who is or can

become fully aware of the whole-system bag of considerations. 2c1a

(2) Formulating and evaluating a best-design plan for the whole venture. 2c2

(3) Executing the plan, under a qualified, experienced management team. 2c3

In the meantime, I think that it would be important to pursue answers to some of the component problems as listed in Branch 2a above. From my experience over the past ten years, where the problems of making an exotic system really workable aren't really met until the system is put to work in an evironment of "doing real work that way," I'd strongly recommend that these techniques be pursued within relevant activities designed so that steady evolution of technique accompanies a serious application (serious, but not over-loading). 3

I shall communicate further about two activities in particular that my group proposes to pursue: A Center for Supporting the Development and Operation of other groups' Documentation Development and Management Systems; and a Network-accessible Technical Intelligence System (already known hereabouts as RINS, for Research Intelligence System), for the domains of concern to computer-based systems development. 3a

The first five items under 2a would be logical developments for a Documentaion Development and Management System -- where useful productivity could be achieved at intermediate stages of knowhow, and good, solid perspective provided for each new stage of development, up to the point where full graphical content could be handled well. A Technical Intelligence System could begin with compromise, photo-form storage of relevant mterial, with useful means of physical access and replication, and evolve as rapidly as possible towards solving the digital-store and digital-conversion techniques. 3b

If limited resources were to be allocated to ingesting a complete-text literature collection for an on-line community, then 4

Rather than considering all of the literature related to Computer Science, I'd recommend beginning with a small subset -- a carefully determined collection deemed to be of highest

Memo, D. C. Engelbart To Cordell Green, 2 JUN 71 5:16PM:
On Large, Discipline-Oriented Data Bases for On-Line Network Use

Value to the community that is bootstrapping the system-development industry. 4a

AN "Encyclopedia" venture would come the nearest to having significant payoff, to my mind. It would need to start modestly, in order to settle upon techniques, formats, etc., and in its earlier modes would have some of the flavor of a "professional journal" perhaps, but I'd think that it ould want to be pushed not with miscellaneous contributions, but rather with some people or groups specifically contracting to produce particular sections, willing to go through successive drafts over a period o many months as the content, form, publication mode, etc. were being worked out. 4b

I have planned for a long time to pursue something close to this end, with what collaborative efforts I would be able to enlist; I have been calling my thing a "Handbook" rather than an Encyclopaedia. I actually think that a Handbook would be the more useful thing, making it the object of continuing effort to shape and build, towards having a really effective System-Builders' Handbook. 5

I feel that it should be built so as to be very effectively used, studied, queried, argued over, updated (under comprehensive and effetive editorial management) by an on-line community; but I am also convinced that it should be publisheable for hard-copy consumption, with a lot of effort applied to a transformation/mapping process tht would produce really effective hard-copy reference materials from the computer-held form. The hard-copy form would want to have computer-query techniques to support its user. 5a

One important hard-copy form would be micro-fiche, and an on-line micro-fiche reader (one that can retrieve a specified fiche from a cartridge, and position it to a specified frame) is an important possibility here. 5b

See the "Relevancy Memo" (5220,) for the related notions as communicated to Larry on 7 Dec 69. Note the term and description of "Super Document." The first real super document was to be "The Handbook." I assumed at that time that we would learn first with a Handbook covering our own "augmentation systems," and then propose extending it in content to the larger domain of commputer systems (with distribued-dialogue collaborative help). This type of push

Memo, D. C. Engelbart To Cordell Green, 2 JUN 71 5:16PM:
On Large, Discipline-Oriented Data Bases for On-Line Network Use

Was among the development efforts given up in trade for transferring to the TENEX. 5c

I still feel strongly that an on-line Handbook prject is very worthwhile. It happens that I see some other pursuits that are more basic to get launched, insofar as judging the longer-term payoff in return for the energies that my group could apply, and associated activities about the Network. 5d

I've mentioned these other things in brief to you, and will soon be formulating them and communicating them to IPT as a thinkpiece communique in negotiation toward our next contract. They involve much of the same basic-technique development, but first in direct association with supportive efforts on real system documentation activity, where immediately useful development of graphic representations, manipulaton tools, hard-copy publication means, control and management aids for the documents would be pursued (and applied). I'd propose a Handbook to evolve as the sort of "Meta Documentation" for these first relly applicable Documentation System techniques, and to grow from there along with what other Network and system tools need the kind of evolutionary-design documentation that a Handbook provides (i.e., that includes glossaries, design principles, etc.. 5e

Relevant considerations from the INFOSYS Panel's study: 6

I have been participating in a study on "Libary Automation" -- as a member of the Information Sytems Panel, which was established under the Computer Scienes and Engineering Board of the National Academy of Sciences. The Panel is just finishing an eighteen-month study sponsored by the Council on Library Resources. 6a

From that study I have developed some definite beliefs, some of which are shared by fellow panelists to the extent that they are being included in our final report. Refer to the "Appenndix A" paper (Jounal, 7014,) for extensive notes thereto that I contributed toward our Panel's final report. 6b

When I have further opportunity, perhaps I could elaborate usefully upon these library problems. It seems apparent that there would be ways to conduct some of our experiments on the ARPA Network that would increase their usefulness as a guide the Library world in its desparate effort to automate. Any sizeable venture toward putting a library on line should

certainly be done in such a way as to make it something relevant to the libraries' future. 6c

Memo, D. C. Engelbart To Cordell Green, 2 JUN 71 5:16PM:
On Large, Discipline-Oriented Data Bases for On-Line Network Use

<JOURNAL>7062.NLS;1, 2-JUN-71 8:01 DCE ; Title: Author(s): Douglas C. Engelbart/DCE; Clerk: DCE;
Origin: <ENGELBART>AGREEN.NLS;5, 2-JUN-71 7:43 DCE ;
.DIR=1;.SCR=2;) .HED=".MCH=72; .GCR;.HJH=2;DCE 2-JUN-71 8:00
7062.MCH=65;
.HJH=1;
Memo, D. C. Engelbart To Cordell Green, .GDATM;:
On Large, Discipline-Oriented Data Bases for On-Line Network Use
"; .MCH=65; .SNF=72; .DLS=0; .PGN=0; .PES;
Links: (AScratch,) (AORGNP,) (AARCXP,) (AARCNP,) (ABCXP,)
(ABCNP,)
Message to Cordell re. the CS-Lib possibility, etc.
From:(AScratch, 2:gebbt) ['(];

RECEIVED at ARC Week Ending 28 May 1971

for Storage and Retrieval
Gordon K. Springer Dec 1970
Describes computer system developed to accept digitized or textual information as input and to produce the necessary index information needed to enter an information retrieval system to store, retrieve, or update a file of pictorial data. 4a

University of Michigan. MHRI. Dept of Computer and
7041 Computation Sciences
Structural Communication in a Personal Information Storage and Retrieval System

Richard Warren Sauvain March 1970
Describes development and use of AUTONOTE system. 4b

Executive Office of the President
7038
Delphi Conferencing (i.e., Computer Based Conferencing with Anonymity)
Murray Turoff March 1971 TM-125
Report on a 13 week conference, Spring 1970, of 20 individuals throughout the U.S. Conference procedures, statistics, hardware, software, and cost considerations. 4c

Pennsylvania State University. Computer Science Dept.
7043
The SOLID System. Vol. I Design Philosophy, Basic Frame, and Compressor
Paul A. D. deMaine, James T. Perry, and Gordon K. Springer 1 May 1971

Appendix
7044 SOLID system can be used to organize any collection of information items, e.g., documents or business files, which have been assigned unique descriptor sets.
Automatic Organization of Files. I. Overview of the
7045 SOLID System.
P. A. D. deMane, N. F. Chaffee, G. K. Springer.
Description of a high-speed self-organizing, fully

automatic, Information Management/Retrieval System.
FRONT provides user-machine interface, TRANSLATOR converts
requests to information independent forms, and RETRIEVAL is
capable of processing any type of question. hd

RECEIVED at ARC Week Ending 28 May 1971

<JOURNAL>7063.NLS;1, 2-JUN-71 9:52 BER ;Title: Author(s): Jeanne B. North/JBN; Distribution: Douglas C. Engelbart, ARC Black Board, Little Black Book/DCE ABB LBB; Clerk: BER;
Origin: <ROW>RECEIVED.NLS;1, 2-JUN-71 9:51 BER ;