# VISTA

## Programmers Reference Manual

Information system

First edition
August 1984

PN 99110445

# NOTICE!

The content of this publication can be changed without warning

Norsoft makes no representation or warranties with respect to the contents hereof and specially disclaims any implied warranties or merchantability or fitness for any particular purpose.

First Edition - August 1984 - is based on VISTA 1.04

# TABLE OF CONTENTS

- o O o -

# Chapter 1 OVERVIEW OF VISTA

The version of VISTA described in this manual is a single user system for 16 bit computers with a minimum of 256 K memory (or more powerful computers).

As supplied VISTA PL consists of 4 executable programs

- VISUP
- SKJEMA
- VIPS
- SKJDOK

SKJEMA Is the schematic generating and editing module.
SKJEMA is also the tool to generate and edit the VPL-code assosiated with a given schematic.

VIPS Is the run-time module. It is both a programme and an application generator.
As a programme it provides a document handling system.
Documents can be stored ,retreived ,displayed ,changed ,erased , sorted in many different ways and printed.
As an application generator it provides a development test-bed which includes debugging aids. With use of VPL, taylor made applications can be designed ,developed and tested. VPL also allows the designer to develop a user interface so that the document handling system is hidden from the user.

VISUP Is a module to edit userdefined procedures. VISUP can also be used to select terminaltypes and languages (usually only one language and one terminaltype is supported)

SKJDOK Is a programme to print the defined schematics and VPL code for documentation and debugging purposes

# Chapter 2 Description of VPL language

## 2.1 Introduction

VPL is an acronym for VISTA Programming Language. VPL is a special purpose language for use with the VISTA Applications Generator. It is implemented as an interpreter in VIPS which is the name of the run-time module in the VISTA product.

VPL is optimised for ease of database interaction, string manipulations, and system control. The VPL interpreter supports procedures written in VPL .

The fundamental terms used in this description are "operator" and "argument". An operator is a "verb" in natural language, i.e. it indicates action. Plus ("+") is an example of an operator. Operators usually perform some action on arguments ("nouns" in natural language) to produce a result. The result of an operator can be viewed as an argument to the following operator.

Examples of operators:

| | |
|---|---|
| + | Plus |
| = | Assignment |
| PICK | Named operator for fetching substrings |

Examples of arguments:

| | |
|---|---|
| #13 | Screen Field |
| #201 | Hidden field (off screen storage) |
| #509 | System Variable |
| #901 | Status line field |
| '...' | Quote String |
| 513 | Positive number |

Constants can be delimited by quotes or double quotes. As a special case positive numeric constants can be written without quotes. Thus the two constants: '234' and 234 are equivalent.

In VPL the order of evaluation is left-to-right. There is no hierachy between operators. Several special characters are neither operators nor arguments. The most obvious example of these are parenthesis.

Examples of special characters:

| | |
|---|---|
| ( ) | Parenthesis |
| < > | Indicate a right argument list |
| , | Statement separator, or right argument list separator |
| ; | Comment follows |

A VPL process consists of statements. Several related VPL operators following one another within a statement form an expression. A VPL statement can contain several unrelated expressions. Each line of VPL may contain several statements. Comma is used as statement delimiter. If the last operator in a statement does not return a value the delimiter is not required.

## 2.2 Syntax of VPL

Operators in VPL perform some action on the arguments which surround them. To formalize this description arguments are described relative to their position from the operator in question, hence the terms "left argument" and "right argument".

Example: 1 + 2

| | |
|---|---|
| 1 | is the left argument |
| + | is the operator |
| 2 | is the right argument |
| 3 | would result from this expression |

In the above operation the left and right arguments are symmetrical (i.e. 2 + 1 would yield the same result). If the operator was minus ("-") then the right argument would be subtracted from the left argument. Thus 5 - 3 would yield 2 .

Arithmetic operators require two arguments and the above notation is sufficient. In the more general case it may be necessary to have more or less than two arguments. If no arguments are required then the operator can be by itself. If one argument is required it is usually given as the left argument (but in some situations it can be given as the right argument).

If more than two arguments are needed for an operator (often the case in string handling) then a construct called a "right argument list" is used. A right argument list appears to the right of the operator it refers to and is surrounded by "<" and ">". There can be up to 10 arguments in a right argument list. Each argument in the list can be an expression. Arguments in a right argument list are separated by commas.

Following is a list of valid operator/argument sequences:

| | | | |
|---|---|---|---|
| i) | No arguments | e.g. | BLANK |
| ii) | Left argument only | e.g. | 1 INPUT |
| iii) | Right argument only | e.g. | BLANK 1 |
| iv) | Right argument list only | e.g. | BLANK < 1 > |
| v) | Left and right argument | e.g. | 1 + 2 |
| vi) | Left argument and right argument list | e.g. | 1 PICK < 2 , 4 ><br>1 PICK <   , 4 > |

Example of a right argument list:

```
'abcdefgh' PICK < 3 , 5 >
```

Meaning: From the left argument (string 'abcdefgh') pick 5 characters starting from position 3 yielding the string 'cdefg' as the result.

Most operators yield a result. This result can be used as the left argument to the following operator (i.e. the operators further to the right in the current expression).

## 2.3 Order of evaluation

An expression is scanned left to right. The first argument seen in an expression is taken as a left argument. Assuming an operator is recognized next the scan will continue looking for a right argument to this operator. When another argument is recognized it is taken as the right argument to the current operator and then the current operator is executed. The result of the operator becomes the left argument as the scan continues along the expression.

The left-to-right scan attempts to maximise the number of arguments to an operator.

Example:

```
12  +  13  +  3  + '-6' = #3 = #4
```

The first addition has a left argument and a right argument and after its execution the line can be envisaged as:

```
          25  +  3  + '-6' = #3 = #4
```

And so on...

```
                   28  + '-6' = #3 = #4

                            22  = #3 = #4
```

Here the operator is assignment with the right argument being screen field three. Assignment yields its left argument as its result. The string '22' would appear in screen field 3.

```
                                  22  = #4
```

Now the string '22' is put in screen field 4, and the original expression results in:

```
                                        22
```

N.B. In the above example positive numeric constants where written without quotes while the negative constant '-6' was surrounded by quotes.

### 2.3.1 Parenthesis

Parenthesis are special characters as indicated above. They can be used to change the order of evaluation described in the above paragraph. As soon as a left parenthesis is detected the expression contained in the set of parenthesis (there must be a matching right parenthesis) is evaluated before anything else is done.

Example:

```
3  +  2  + (  3  *  4 ) = #4
```

This will go through the following steps in evaluation:

```
5  + (  3  *  4 ) = #4
```

The left parenthesis is now detected when a right argument is sought for the addition, therefore:

```
5  +  12          = #4

      17          = #4
```

So '17' would appear in screen field 4 and the overall expression would finish with

```
                    17
```

N.B. If the parenthesis had not been present in the above example then '32' would have been placed in screen field 4.

Parenthesis can be nested to sexteen levels in a single expression.

### 2.3.2 Right arguments

If there is only one right argument it can be represented in two ways:

```
a)       1    + 3
b)       1    + < 3 >
```

Both expressions will result in 4. The above representations are equivalent ( so the first is favoured because it is simpler).

Some operators require three or more arguments. The "PICK" string operator for taking a selected number of characters out of a given string is such an example. The three arguments are:

| | |
|---|---|
| i) input string | - left argument |
| ii) position to take characters from | - 1st right argument |
| iii) number of characters to take | - 2nd right argument |

Example:

```
                  'Paul-Brennan' PICK < 6 , 3 >  = #2
```

Evaluates to:

```
                                          'Bre'  = #2
```

The multiple right arguments are represented as a list of expressions separated by commas, and the list delimited by "<" and ">". The interpreter supports up to 10 arguments in a right argument list.

If, for example, the first right argument is to adopt its default value ( position 1 in the case of PICK) then the following expression is possible:

```
                  'Paul-Brennan' PICK <, 3 >     = #2
```

Evaluates to:

```
                                          'Pau' = #2
```

It is important to realize that the right arguments in such a list can themselves be VPL expressions. Thus the first example in this paragraph could appear as:

```
                  'Paul-Brennan' PICK < 4 + 2 , 3 > = #2
```

Evaluates to:

```
                  'Paul-Brennan' PICK < 6     , 3 > = #2
```

And then:

```
                                              'Bre' = #2
```

## 2.4 Fields and variables

### 2.4.1 Screen fields

The screen fields are those areas in the schematic on the screen into which the user is allowed to enter data. Each field can be viewed as an entity. Screen fields can be no longer than the width of the screen and must always be wholly within one line. The number of screen fields in any one schematic is limited to 200. It is possible to have schematics with no screen fields at all.

The screen field numbers are a sequence running from 1 up to a maximum of 200. The "natural" order of screen field numbering is left-to-right and then down the screen. This order is assumed in the module which creates schematics called "SKJEMA". This module allows re-ordering of the screen field sequence. The screen field numbering sequence determines the order in which the cursor will pass between the screen fields. Screen fields can always be accessed by number. VPL accesses the screen fields by stating their field numbers prefixed by "#".

Note that constants are contained in quotes. It is permissable to write positive constants (i.e. numeric strings) without quotes.

Example:

```
'fred'= #33
 #33  = #2
```

The first statement would put the string 'fred' into screen field 33. It would appear left justified in that field. If the field was longer than 4 characters then spaces would be added to the right. If the field was less than 4 characters then only the leftmost characters of 'fred' would appear.

The second statement would pick up the contents of of screen field 33 and then place it in screen field 2, thereby replacing the previous contents of screen field 2.

When screen fields are read trailing spaces are ignored. Thus in the above example if the screen field 33 was 80 characters long then reading it (left argument of second statement) would yield only 4 characters. When the contents of one field is being assigned to another this is not important but if the contents of two fields are being joined together (an operator called JOIN) then this is significant.

Example:

```
  'abc'   = #1      ;Assume field 1 is 8 characters long
#1 JOIN #1 = #2     ;Assume field 2 is 8 characters long
```

Then screen field 2 would finally contain:
abcabc (left justified)

VPL code can be executed in various contexts but in all cases one of the screen fields is assumed to be "current". As a shorthand notation the "current" screen field can be addressed as #0.

Example:

```
'Hello' = #0     ;Put 'Hello' into current screen
                 ; field
```

To summarize: The fields within the schematic on the screen are called screen fields. These screen fields can always be addressed by a sequence of numbers. Optionally these fields may also be named.

### 2.4.2 Status line fields

There can be from 1 to 10 statusline fields refered to as field 901,902, --- 910.
See chapter 4, the description of the operators SA, SL, SP, SR, SV and SW.

### 2.4.3 Hidden fields

Hidden fields are thus named because they have most of the properties of screen fields, but lie off the screen "hidden" from the users sight. There are three main differences between hidden fields and screen fields:

Both leading and trailing blanks are removed when a value is assigned to a hidden field.

If a numeric type datum (the result of an arithmetic operator) is assigned to a hidden field, the numeric type is retained, and rounding has no effect. A reference to such a value by an operator which requires string type data will give the same effect as if the reference was an arithmetic expression (see the description in chapter 4 of the assignment operator (=)).

All hidden fields are global, and are initialized as empty when VIPS is started. It should be noted that nothing can be assumed about the "volatile" hidden fields, and that the "long" hidden fields may be reinitiated to empty if a value is assigned to system variable 519 (See chapter 5)

There are three groups of hidden fields:

Short hidden fields 201-230. These fields each have a length of 16 characters, and can be used by applications for global storage of values.

"Volatile" short hidden fields 291-299. These fields have the same properties as the user short hidden fields, but are primarily meant as temporary work locations for VISTA. They may be freely used by user applications, but must be regarded as undefined on entry into a schematic and after the use of a procedure.

Long hidden fields 301-310. These fields each have a default length of 80 characters, and otherwise have the same properties as the user short hidden fields by default. A total of 800 characters are reserved for the long hidden fields, thus making 10 such fields available as the default value. These fields are numbered 301-310, The length of these fields may be changed by the application by writing a value to system variable 519 (see the description in chapter 5). The length may be in the range 40-255 characters, thus the number of long hidden fields may vary from 20 to 3. When a new value is assigned to system variable 519 the new length is calculated and stored in system variable 518. At the same time all the user long hidden fields are reinitiated to empty.

### 2.4.4 System variables

System variable are "hidden fields" numbered from 401 to 599. These variables are divided into 2 classes, the informative variables which can only be read, and the variables that affects the behavior of the system and which can be both read and written. These variables are described in detail in chapter 5.

## 2.5 Procedures

VISTA allows the use of user defined procedures. VPL is processed interpretatively. When the interpreter encounters an operator name it does not recognize, it assumes that it is a procedure call, and attempts to execute it as such after first compiling the arguments in the same manner as for an ordinary operator. A procedure can as the operators have zero or one left arguments, and zero, one or more right arguments (a list of maximum ten right arguments).

A procedure returns a result, which is a string of zero through 255 characters long.

The procedures reside in the file VISETUP.VSF, and are defined using the program VISUP.

A procedure may call another procedure. Procedures are recursive in nature. A procedure may therefore directly or indirectly call itself.

Return from a procedure is done either when a RETURN operator is encountered, or after the last line of the procedure has been processed.

Inside the procedure an argumen is indicated by % (percent) followed by a number ,and are referenced as:

| | |
|---|---|
| %1 | Left argument |
| %2 | Right argument |
| %11 | First argument of right argument list |
| %12 | Second argument of right argument list |
| -- | |
| . | |
| . | |
| -- | |
| %20 | Tenth argument of right argument list |

The return value from a procedure must be assigning to %0.

A procedure may be invoked with a variable number of arguments. In this case the operator EXIST (see chapter 4) may be useful.

Inside a procedure #0 is a reference to the contents of current field, while #448 contains the field number of current field .

The operators GOTO, EXIT and SCHEMA are illegal inside procedures.

## 2.6 Special action

These are facilities to aid in debugging or error correction.

### 2.6.1 System error action

If, during processing of VPL, an error condition is discovered, VIPS displays an error message on the status line and waits for a keypress from the user:

| | |
|---|---|
| F8 | The current process is terminated. |
| Down arrow | The line of VPL where the error condition was detected is displayed on the status line, and ?? marks the position in the line. The system waits for a further keypress. |
| Any other | The current process is resumed at the beginning of the next line of VPL. |

### 2.6.2 Keyboard interrupt

Keyboard interrupt is a facility to interrupt the execution of VPL, and is incurred by pressing F8 twice. The processing is halted, the message "Keyboard interrupt" is displayed on the status line, and the system waits for a keypress (see the previous paragraph).

### 2.6.3 Debugging single step

The system has a facility to execute VPL on a line by line basis, while each line is displayed on the status line before execution. This is controlled by system variable 535, see chapter 5.

## 2.7 Help structure

The system provides a default help structure which can be evoked at any time by the user pressing F2. This help structure can be replaced by a user supplied help structure (wholly or partly). This is described in chapter 4, the SHELP operator.

# CHAPTER 3 VISTA/VERBAL - The underlying database system

The VISTA product grew out of an attempt to add a menu-based interface to a database called VERBAL developed by Sturla Sandlie in 1978.

VERBAL is a freestuctured database management system with a query language based on a grammar of sentences, subsentences and words. The query language is extremely flexible, and had to be flattened somewhat to fit the constraints of presenting information through fixed formats (schematics). To avoid confusion between original VERBAL and the database system used by VISTA the latter is called VISTA/VERBAL.

The first thing to point out about VISTA/VERBAL (and VERBAL) is that the database system is not told in advance about the format of information (documents/records) to be stored in the database. This means that the database file initiation is very simple, and can be done at a very early stage, after which data entry may be started as soon as the first data entry schematic has been made. This feature also means that schematics may be changed, or added after a lot of information has been stored without necessitating a reorganization of the database.

VIPS, the run-time module, can only access one database file at a time but this is no problem in practice as this database file may contain any number of registers(groups of documents with the same schematic name). The term "file" refers mostly to a single operating system file throughout this documentation, but may refer to a group of such files if the operating system on the host machine is to restrictive re. filesize. This will be transparent to the VISTA user except in the context of file by file security backup.

## DOCUMENTS

VISTA/VERBAL stores documents. Documents are made up of fields. In Data entry mode there is a one by one relationship between the fields in the schematic and the fields in the stored document. It is also possible to store documents from VPL, in which case there need be no relationship between the stored documents an any schematic. Only the contents of non-empty fields are stored, and all leading and trailing blanks are removed (to save storage).

Documents stored in Data entry mode can contain up to 200 fields, while there is no limitation on the number of fields if a document is stored from VPL.

Each field can vary between empty and 255 characters. All fields fall into one of two classes. These are "key fields" an "non-key fields". Non-key fields are stored as part of the document, but are not searchable, while the contents of key fields may be used to retrieve the document through a Search. Key fields may contain no keys, one key or multiple keys. Multiple keys are separated by semicolon(s).

## KEYS

Keys are the "hooks" the user has to information stored in the database. It is important to distinguish between "keyfields" and "keys". Keyfields are defined in the schematics, and if a schematic is used for Data entry or Edit then the corresponding key/ non-key is invoked. As noted above a keyfield may contain zero, one or many keys. It is also possible, through the use of VPL, to associate a key value with a different field from where the value is stored. This is especially useful in documents containing data on a tabular form, with repeated groups of identical information.

## SEARCH PROFILE

A search profile is a collection of search criteria which will be applied to the database to find all documents containing the specified combination of keys. Any combination of keys may be given as a search criterion.

The fastest search is for exact match on one or more keys. It is also possible to specify one or more "wild card" select criteria which includes such terms as "all except", "greater than" and "less than".

The collection of documents resulting from a search are all those that satisfy all the given criteria concurrently.

## OCCURRENCE LIST

The result of a search is an "occurrence list". This is a list pointing to the documents which met the search criteria. VISTA/ VERBAL can maintain approx. 125000 such lists concurrently, while VISTA limits the number available to the user to 101. These are referenced by list numbers 1 through 101. List number 101 is spesial purpose, and is commonly called the current occurrence list. This is used as the default occurrence list where such is needed.

At any time a document can be pointed to by zero, one or several occurrence lists. Occurrence lists are global in action but they are all removed when exit is made from the VIPS program. When the VIPS program is started there are no occurrence lists.

Note: Occurrence lists generated by Sort will cause spurious results if used with any of the logical list operations.

The Delete of a document will make that document unavail- for a later search, but the document itself is not removed immediately, it is still available for read-access in all the lists pointing to it. Attempts to modify or delete an already deleted document will cause an error message.

## LOGICAL LIST OPERATIONS

This is best shown by example:

Let us assume that there is a register of Norwegians whose names and other information are held in a group of documents. The names and towns from where they come are:

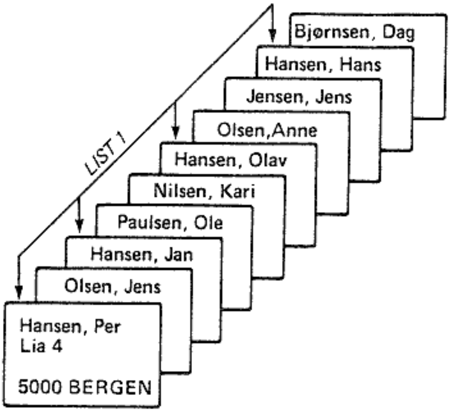
| Christian name | Surname | Town |
|---|---|---|
| Per | Hansen | Bergen |
| Jens | Olsen | Bergen |
| Jan | Hansen | Oslo |
| Ole | Paulsen | Trondheim |
| Kari | Nilsen | Alta |
| Olav | Hansen | Larvik |
| Anne | Olsen | Bergen |
| Jens | Jensen | Bergen |
| Hans | Hansen | Bergen |
| Dag | Bjornsen | Kirkenes |

Now let us apply two search profiles against these documents.

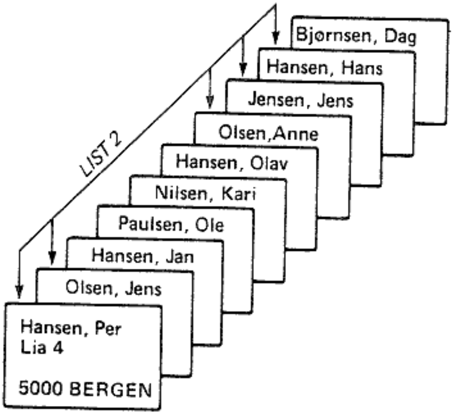
List 1 will be those with a surname of "Hansen".

List 2 will be those who live in "Bergen".

These lists are visualized on the next page.

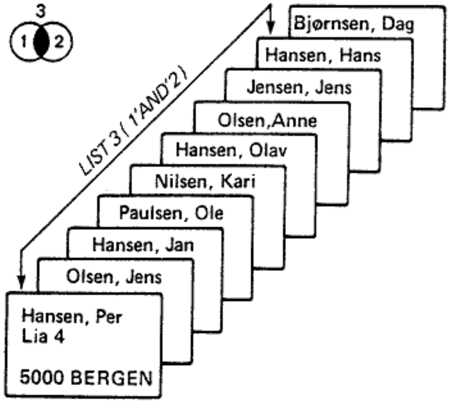


1. List of those with a surname of "Hansen".



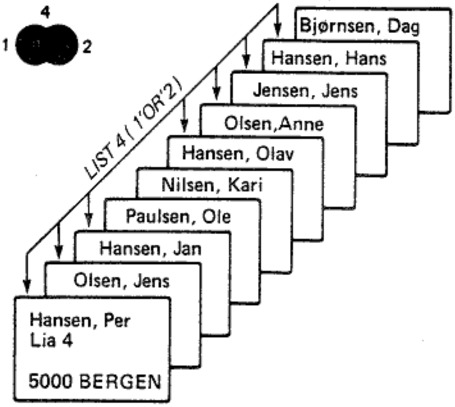
2. List of those who live in "Bergen".

With the logical list operation "AND" a new list can be formed of ALL those who have the surname "Hansen" and live in "Bergen". This new list is called list 3 and is shown below.



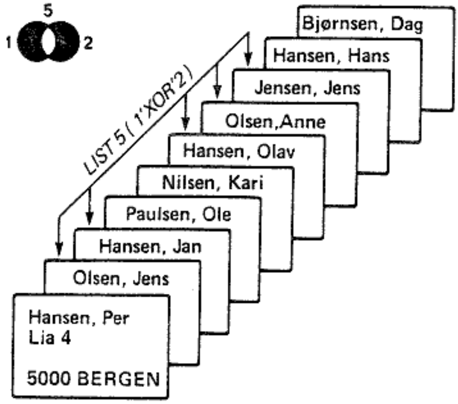
3. List of all those who have the surname "Hansen" AND live in "Bergen".

With the logical list operation "OR" a new list can be formed of those who live in "BERGEN" OR have the surname "Hansen". This new list is called list 4. N.B. Those who both live in "Bergen" and are called "Hansen" are not duplicated.

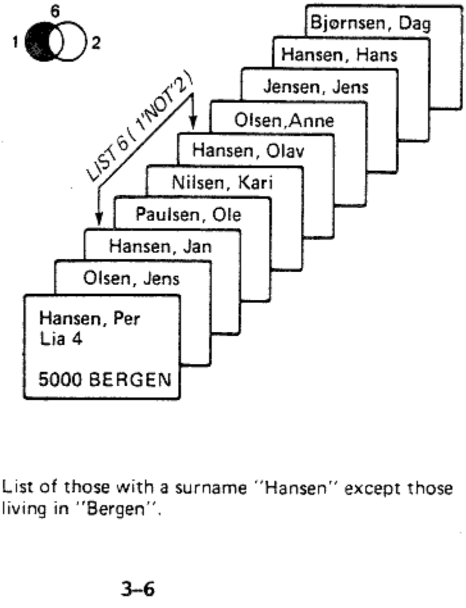


4. List of those who live in "Bergen" OR have the surname "Hansen".

With the logical list operation "XOR" a new list can be formed of those who live in "Bergen" or are called "Hansen", but excluding those who both live in "Bergen" and are called "Hansen". The new list is list 5.

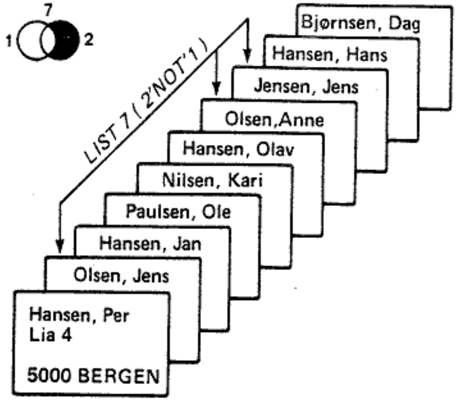


With the logical list operation "NOT" it is possible to form a new list of those in the first list excluding those in common in the second list. Since the order of the list is important two examples are given. List 6 are those whose surname is "Hansen" who do not live in "Bergen".



6. List of those with a surname "Hansen" except those living in "Bergen".

List 7 contains those who live in "Bergen" but whose name is not "Hansen".



7. List of those living in Bergen except those with a surname "Hansen".

## TERSE DATABASE DESCRIPTION

Keywords: Document, field, key, non-key, keyfield, dictionary

The database system underlying VISTA stores documents. The format of a document is not defined in advance to the database system. Each document can contain any number of fields. Fields can vary between empty and 255 characters long. Documents are not of fixed length. The amount of storage set aside for each document is the sum of its fields after removal of leading and trailing spaces.

Fields can be divided into two types, key and non-key fields. The key/non-key make-up of documents can vary dynamically from one document to the next.

The database system maintains dictionaries. These dictionaries are not predefined, being maintained by the system dynamically without user interaction. They are introduced here to give an insight into the operation of the database. Documents in the database may be considered as divided into groups. Each group has an associated schematic name from which that group of documents were generated.

A separate dictionary holds the schematic names associated with the documents. Every document in the database has one, and only one schematic name associated with it.

Keys are maintained in a set of different dictionaries. In the simplest case a key is the contents of a keyfield in a document. A keyfield will be considered to hold no key if it is blank. A keyfield can hold more than one key with semicolon ";" being considered as the delimiter. A transformation takes place from the contents of a keyfield to the value stored in the dictionary. The key value stored in the dictionary will be no more than 31 characters after all blanks are removed, and lower case letters are folded to upper case. It is important to realize that the field name (without extention) from which the key came form an integral part of the key. Keys will be randomly distibuted throughout the set of dictionaries.

Inserting, editing and deleting of documents is done at the time of request, and the dictionaries are suitably adjusted, thus the presence of dictionaries is transparent to the user. The only visible effect of having a set of dictionaries with random distribution of keys is in the speed of retrieval.

The database need not be reorganized in order to reuse space that is released by deleted documents. This is done dynamically.

# Chapter 4 DESCRIPTION OF VPL OPERATORS

## FORMAT OF OPERATOR DESCRIPTION

```
Operator Name:            1)                   !       2)
                                               !
                                               --------------------

Class:          3)

Arguments:      4)

Result:         5)

Summary:        6)


Description:
------------
                7)
Examples:
                8)
Extensions:
-----------
                9)
```

## EXPLANATION OF FORMAT:

1) This is the operator name or alternatively the symbolic representation of that operator (e.g. "+" ).
2) This box is for redirections. Not all operators have a separate page each.
   For example, in the case of the arithmetic conditional operators (EQ NE GE GT LE LT) only EQ is described at length while the others are redirected to EQs description. Thus in the case of LE which should appear before LLENG then the box in LLENG is used to note that LE is described under EQ.
3) For ease of description elsewhere the operators are classified functionally. The current classes are:
   - Database
   - String
   - Control
   - Arithmetic
   - Status line
   - Printing and sequential file handling
   - Specials

   In addition operators generating condition codes have "conditional" appended to the class.

4) The arguments that an operator takes are encoded as follows:

| | |
|---|---|
| L | Left argument |
| R | Right argument |
| R1 | First argument in a right argument list |
| R2 | Second argument in a right argument list |
| .. | |
| R10 | Tenth argument in a right argument list |

If an argument code is followed with a "*" then it is compulsory. In many cases a simple right argument (R) and a right argument list (R1) are identical as seen by the operator. If this is not the case then it will be noted in the operator description.

5) If an operator yields a result then this is indicated by a "Yes". If the result can be very simply described then it is. (e.g. error code).
6) Summary of the usages of this operator. This is restricted to a few lines. The symbol "->" is used to mean "yields the result" and should not be confused with assignment (=).
7) The description of the action of the operator. This description attempts to be as definitive as possible and does not concentrate on the usages of the operator. If the operator effects or is effected by system variables or other operators then this is noted.
8) These are examples drawn from the general usage of the operator. An attempt is made to give examples of all the normal usages of the operator. Again the symbol "->" is used to mean "yields the result" and should not be confused with assignment (=). What lies to the left of "->" will usually be a VPL expression. To the right of the "->" will be a number (expressed in the simplest form) or a string (which will be enclosed in quotes).
9) If there are some extensions to the operator that would complicate the description then they may be described in this section. These extensions would be for advanced use or not meant to be used at all by the application designer but included for completeness.

This operator description format is meant only as a guide and where an operator needs special attention then this format will be "bent". In particular points that need to be stressed may be set off with the heading "N.B." .

```
Operator Name:              +                    !
                                                 !
                                                 --------------------

Class:          Arithmetic

Arguments:      L*  R*

Result:         Yes

Summary:        num     +       num     ->      num
```

Description:
------------
This operator will add two numbers together.

The left and right arguments must be given. They both must be numbers (i.e. strings that can be interpreted as numbers).

Examples:

```
         1   +   1   ->  2       ; as expected 1 plus 1 gives 2
        '1'  +  '1'  ->  2       ; both strings can be decoded as
                                 ; numbers
        22   + '-1'  ->  21      ; negative numbers need to be
                                 ; expressed as strings
        1.2  +  3.7  ->  4.9
         1   +  <3>  ->  4       ; right argument can be in a list
```

Notes about numeric accuracy
----------------------------
Three major data types exist: character, integer, and floating point. To make the situation more complicated each of these major data types can be sub-divided into more data types. For example, one can have 16 bit and 32 integers (and lots more).

In VISTA everything is treated as a string of characters. A number is a string which can be interpreted as a number. For example, the string '123.4' can be interpreted as a number while '12a.4' cannot. Strings need to surrounded by quotes when written explicitly in VPL code. An exception to this rule is a positive number which may be written without surrounding quotes. This is meant as a notational "short-hand" to save keystrokes during VPL coding.

Within the database everything is stored as a character string. During VPL interpretation deferred arguments and hidden fields are held in the most convenient internal form. If the result of arithmetic is to be placed in a hidden field and if this result can be represented as an integer then it is. If the result of arithmetic is to be placed in a hidden field and if the result cannot be represented as an integer then it is held internally as a double precision real.

All type translations are carried out transparently and this information is provided to assure the application designer that maximum numeric accuracy is being maintained.

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| Operator Name: | - |  |  |  | !<br>! |
| Class: | Arithmetic |  |  |  |  |
| Arguments: | L R* |  |  |  |  |
| Result: | Yes |  |  |  |  |
| Summary: | num | - | num | -> | num |
|  |  | - | num | -> | num |

### Description:

This operator will subtract two numbers or negate a number.
If two arguments are given then the right is subtracted from the left.
If only a right argument is given (i.e. no left argument) then it is negated (i.e. subtracted away from zero).

The right argument must be given. It must be a number (e.g. a string that can be interpreted as a number). If a left argument is given then it must also be a number.

Examples:

```
 3   -   1   ->  2       ; as expected 3 minus 1 gives 2
'3'  -  '1'  ->  2       ; both strings can be decoded as
                         ; numbers
22   - '-1'  -> 23       ; negative numbers need to be
                         ; expressed as strings
1.2  -  3.7  -> -2.5
     -  3    -> -3       ; negation of 3 gives -3
     - '-1'  ->  1       ; negation of -1 gives 1
```

| | | | | | |
|---|---|---|---|---|---|
| Operator Name: | * | | | | |
| Class: | Arithmetic | | | | |
| Arguments: | L* R* | | | | |
| Result: | Yes | | | | |
| Summary: | num | * | num | -> | num |

Description:

This operator will multiply two numbers together.

The left and right arguments must be given. They both must be numbers.

Examples:

```
 1    *  2   ->  2      ; as expected 1 times 2 gives 2
'1'   * '2'  ->  2      ; both strings can be decoded as
                        ; numbers
22    * '-3' ->  -66    ; negative numbers need to be
                        ; expressed as strings
1.2   * 3.7  ->  4.44
 2    * <3>  ->  6      ; right argument can be in a list
```

```
Operator Name:          /                      !
                                               !
                                               --------------------
```

Class: Arithmetic

Arguments: L R*

Result: Yes

```
Summary:       num     /       num     ->      num
                       /       num     ->      num
```

Description:
------------

If two arguments are given then the left is divided by the right. If only a right argument is given then it is inverted (i.e 1 / num ).

The right argument must be given. It must be a number. If a left argument is given then it must be a number.

Division is performed to a precision of 16 digits(equipment dependent).

The result of the division can be controlled by system variable 406. If it is 1 (default) then the result is returned as calculated. If it is 0 then the result is truncated towards zero to an integer.

If the divisor is zero then the error message: "** VPL ** Attempt to divide by zero" will be placed on the status line.

Examples:

```
              6   /   3   ->  2        ; 4 divided by 2 gives 2
             '6'  /  '3'  ->  2        ; both strings can be decoded as
                                       ; numbers
             22   /  '-2' ->  -11      ; negative numbers need to be
                                       ; expressed as strings

              1   /   3   ->  0.3333333333333333

              6   /  <3>  ->  2        ; right argument can be in a list

                  /   2   ->  0.5      ; no left argument so invert 2


              0   =   #406             ; want result of division as
                                       ; integer
              8   /   3   ->  2        ; result truncated to integer
              1   =   #406             ; back to normal division
              8   /   3   = [2] #7     ; round result placed in field 7
                                       ; to 2 decimals:

                              ====================
After:        Screen field 7: !              2.67!
                              ====================
```

Operator Name: =

!
!

---------------------

Also treated here: =[ Dec. Just.]

Class: string

Arguments: L* R*

Result: Yes

Summary: num1 = num2 -> num1

Description:

------------

This operator assigns the left argument into the field indicated by the right argument. By "field" is meant any screen field, status line field, hidden field, or any system variable which can be written to. Also "field" could be the indirection of an expression.

The result of this operator is its left argument.

It is important to note that this operator ( = ) is viewed syntactically in exactly the same way as the operator + (for example). That is to say they both have a left argument, a right argument, and a result.

The format of the data assigned to a field depends both on the left and right arguments.

The contents of screen fields, status line fields, and fields returned from the database are always "string type". This means they are stored internally with an exact ASCII representation. The result of the arithmetic operators yields a more concise internal form called "numeric type".

As a general rule when "string type" is assigned to a screen or a status line field it will be left justified. When "numeric type" is assigned to a screen or a status line field it will be right justified. This default justification can be overridden by system variable 522 or by the letter "L" or "R" within a[] structure.

When "string type" or "numeric type" is assigned to a hidden field or a system variable then it can be viewed as being left justified. Furthermore "numeric type" will be put in a hidden field in some convenient internal type (integer or double precision real).

When "string type" is placed in a screen or status line field then spaces will be added to, or characters truncated from it so that it completely replaces the previous contents of that field.

When "numeric type" is placed in a screen or status line field then spaces will be added to it to the left so that it completely replaces the previous contents of that field. If the "numeric type" is too long to be placed in a screen or status line field then digits to the right of the decimal point (and the decimal point) will be truncated in an attempt to fit the "numeric type" into the field. If

Operator name: ≐ (continued)

the "numeric type" will still not fit in the field then the field is filled with exclamation marks "!!!!!!!!".
If rounding has been specified (either by system variable 451 or by a number between [] ) then it only affects the placement of "numeric type" in screen and status line fields.

If justification has been specified (either by system variable 522 or by "R" or "L" between [] ) then it affects both "string type" and "numeric type" in screen and status line fields.

If "string type" is known to represent a number (e.g. passed a NUMERIC operator test) then it can be turned into numeric type by adding zero to it.

Example 1:

```
                                 ====================
Before:        Screen field 3:   !SOMETHING         !
                                 ====================

Expression:     'abcdef'    =    #3    ->     'abcdef'

                                 ====================
After:         Screen field 3:   !abcdef            !
                                 ====================
```

Example 2:

```
                                 ====================
Before:        Screen field 3:   !abcdef            !
                                 ====================

Expression:     1.2  *  3.7 =    #3    ->     4.44

                                 ====================
After:         Screen field 3:   !              4.44!
                                 ====================
```

Example 3:

```
                                 ====================
Before:        Screen field 3:   !              4.44!
                                 ====================

Expression:     1.2  *  3.8 =[1L] #3 ->       4.56

                                 ====================
After:         Screen field 3:   !4.6               !
                                 ====================
Operator Name:          =>              ! See BRANCH operator
                                        !
                                        ---------------------


Operator Name:          =>>             ! See GOTO   operator
                                        !
                                        ---------------------


Operator Name:          ?               ! See INPUT  operator
                                        !
                                        ---------------------


Operator Name:          ==              ! See EXECUTE
                                        ! operator
                                        ---------------------
```

Operator Name: **AND**

Class: Arithmetic, conditional

Arguments: L* R*

Result: Yes, condition code

Summary: cc AND cc -> cc

Description:

This operator performs a logical AND operation between its arguments and produces the appropriate result. The truth table for AND is:

| LEFT | RIGHT | RESULT |
|---|---|---|
| false | false | false |
| true | false | false |
| false | true | false |
| true | true | true |

```
where: false  <->  0 [ -0.5 < x < 0.5]
       true   <->  not false
```

Note: Using the AND operator with arguments that are not the result of conditional expressions may cause unexpected results, as the AND operator is simply a multiplication of values.

IF-THEN-ELSE-ENDIF and DO-WHILE-ENDDO strtures can both be controlled by condition codes. Sometimes a combination of conditions is required to be true for some action to be taken. This operator can be placed between two other conditions so that the net result is only true when both component conditions are true.

Examples:

```
      1   AND    1   ->  1            ; from above table
      0   AND    1   ->  0            ; from above table

  NB! 0.7 AND  0.7   ->  0.49         ; unexpected result

      if  #3 eq 33 AND (#201 lt 0)  then ....
                                      ; if field 3 is equal to
                                      ; 33 AND hidden field 201
                                      ; is less than 0 then....

      do  while ( #901 empty AND (#1 numeric) AND (#2 gt 0) )
          ...                         ; many ANDs can be used
      enddo
```

```
Operator Name:               ASCII                    !
                             -----                    !
                                                      --------------------
Class:          string

Arguments:      L*   R

Result:         Yes

Summary:        num     ASCII               ->    str
                str     ASCII   '-1'        ->    num
```

Description:
------------
This operator will return the ASCII equivalent of the given number in the simplest case. Numbers in the range 0 to 255 are mapped to their ASCII equivalents. All other numbers are mapped to 32 (space).

Non-printable ASCII characters (e.g CR = 13) will not effect the internal workings of the VPL editor. When a string containing non-printable characters (or un-mapped characters for that screen) is put in a screen field then a special character will be substituted on the screen. This special character is defined in the screen handler.

If a right argument is given and it is '-1' then the operator will return the numerical equivalent of the first character of the string given as the left argument. If the left argument is a null string then zero is returned.

Examples:

```
 32       ASCII              ->  ' '
'32'      ASCII              ->  ' '
 65       ASCII              ->  'A'
 97       ASCII              ->  'a'

 11       ASCII              ->   ?     ; screen representation
                                        ; depend on handler
 256      ASCII              ->  ' '    ; out of range

' HELLO'  ASCII  '-1'        ->   32
'HELLO'   ASCII  '-1'        ->   72
''        ASCII  '-1'        ->   0
1         ASCII  <-1>        ->   49
'1'       ASCII  <-1>        ->   49
```

```
Operator Name:          ATTR                 !
                        ----                 !
                                             --------------------
Class:         special

Arguments:     L   R

Result:        No

Summary:           ATTR         ; clear attribute in all fields
                   ATTR   fld   ; clear attribute in given field
               num ATTR         ; set attribute in all fields
               num ATTR   fld   ; set attribute in given field
```

Description:
------------

VISTA supports up to 64 programmable attributes. These are numbered 0-63. Attribute 0 is usually referred to as the "clear" attribute. This operator allows screen fields and status line fields to have their attributes set (1-63) or cleared (0).

In a given terminal only some of these attributes may be defined. It is difficult to squeeze more than 10 different attributes out of most monochrome terminals. With colour screens the whole 64 can be utilized. Attribute numbers are associated with physical screen attributes when the terminal handler is defined in VISETUP.

If this operator is used without any arguments then all the screen fields (not the status line fields) are cleared to the zero attribute. If a left argument is given then it should be a number in the range 0 to 63. The left argument is taken as an attribute number. If a right argument is given it should be in the range 1-200 or 901-910. Field numbers which do not have corresponding fields are ignored.

Examples:

```
        ATTR              ; clear all screen fields to
                          ; attribute 0
        ATTR   2          ; clear screen field 2 to
                          ; attribute 0
    7   ATTR              ; set all screen fields to
                          ; attribute 7
    4   ATTR   15         ; set screen field 15 to
                          ; attribute 4
    4   ATTR   902        ; set status line field 902 to
                          ; attribute 4
```

```
Operator Name:            BELL             !
                          ----             !
                                           ---------------------
```

Class: special

Arguments:

Result: No

Summary: BELL ; ring the bell

Description:

The action of this operator is to ring the bell.

This operator takes no arguments and does not return a result.

Examples:

```
        BELL                   ; ring the bell
        BELL BELL              ; ring the bell twice
```

Extension:

As a special option a number can be given as the first element of a right argument list. Note that a normal right argument is ignored.

The following table gives the action associated with a number in the first element of the right argument list.

| | | | |
|---|---|---|---|
| 1 | clear screen and home | 2 | home |
| 3 | cursor up | 4 | cursor down |
| 5 | cursor left | 6 | cursor right |
| 7 | bell | 8 | delete character |
| 9 | insert character | 10 | delete line |
| 11 | insert line | 12 | erase line |
| 13 | cursor return | 14 | erase character |

N.B. People use the above codes at their own risk! Normal schematic and keyboard handling should be sufficient without the user resorting to these explicit controls.
The software maintains an internal map in memory of what is on the screen. This map is updated to reflect changes.

Example:

```
        BELL  <1>              ; clear screen and home cursor!
        BELL  <12>             ; erase the line the cursor is
                               ; currently in.
```

```
Operator Name:          BLANK                !
                        -----                !
                                             --------------------

Class:          string

Arguments:      L   R

Result:         Yes

Summary:        BLANK        ->  num     ; blank all screen fields
           fld  BLANK        ->  num     ; blank fields after fld
           fld  BLANK  type  ->  num
```

Description:
------------

In the simplest case (with no arguments) all fields on the screen will be blanked. The result will be the number of fields in the schematic.

If a left argument is given it is assumed to be a field number. In this case all fields with a number greater than this number (not equal) will be cleared. The result will be the number of fields in the schematic.

If a left argument is given and a 'TYPE' of '1' is given then all fields less than or equal to the number given in the left argument are cleared. The result is the left argument.

If a left argument is given and a 'TYPE' of '-1' then the result will be the number of the first non-blank field greater than the field indicated by the left argument. If there are no more non-blank fields in the schematic then '0' is returned as the result.

Examples:

```
        BLANK                           ; Blank all fields on the
                                        ; screen.  Result is the
                                        ; number of screen fields.
        3   BLANK                       ; Blank all fields from
                                        ; field 4 onwards. Result is
                                        ; number of screen fields.
        3   BLANK   1                   ; Blank fields 1, 2 and 3
                                        ; Result is '3'

        BLANK '-1'     ->               ; Result is the field number
                                        ; of the first non-blank
                                        ; field.
        3   BLANK '-1' ->               ; Result is the field number
                                        ; of the first non-blank
                                        ; field from 4 onwards.
```

Operator Name: **BRANCH** !

Symbolic representation: =>

Class: control

Arguments: L R (one or the other, right takes precedence)

Result: No

Summary:

```
BRANCH  :L1:        ; branch to line with label "L1"
:L1: =>             ; branch to line with label "L1"
```

N.B. A label is converted into a number by the VPL pre-processor.

## Description:

This operator will transfer control locally within the current process. It is normally used in conjunction with labels. These labels are evaluated by the VPL pre-processor into the numbers (offsets) referred to below.

The IF-THEN-ELSE-ENDIF and the DO-WHILE-ENDDO structures should be sufficient for most programming needs and the use of BRANCH can be viewed as the last resort. It is not recommended to branch into DO loops (but it is well defined). If this operator is to be used for the infamous "computed gotos" then labels are not sufficient.

Either a left or right argument must exist. If both left and right arguments exists then the right argument is taken. The argument must be a number. If necessary it will be rounded to an integer. This integer is referred to below as the offset.

The VPL interpreter will allow any line within the process to be accessed by this operator. The current line is taken to have offset zero. Negative offsets refer to lines before the current line while positive offsets refer to lines after the current line.

Thus the new current line after this operator will be the old current line plus the offset (which is the argument to this operator). System variable 506 reflects the current line number within the current process and will be changed by this operator to indicate the new line number.

If the offset is too greatly negative then the new current line will be line 1 of the process. If the offset is too greatly positive the process will finish and control will be transferred to the next process.

Examples:
First an example showing the normal use of BRANCH and labels:

```
 1>    #1= 201                              ;initialize loop variable
 2> :L1:if #201 gt #448 then :L2: => endif  ;if passed last field out
 3>         if ##201 empty then             ;check if this field empty
 4>             '***' = ##201               ; if so put '***' in it
 5>         endif                           ;
 6>         #201 + 1 = #201                 ;increment loop variable
 7>     BRANCH :L1:                         ;branch to start of loop
 8> :L2:putdoc
 9>      .....
    ^

    ^ assumed to be first position in line
```

This is a loop to replace all blank fields in a schematic with '***' before it is stored. First the loop variable (#201) is initialized on line 1. The loop begins on line 2 with the loop condition which is keep looping until the loop variable exceeds the number of fields on the screen. Line 3 uses the loop variable indirectly to find out if the corresponding screen field is empty. If so the field has '***' put in it. Line 6 increments the loop variable. Line 7 branches back to line 2 (i.e. label :L1:). When the loop is finished the fields on the screen are stored as a document by line 8. It is worth noting that the above example could be done by a DO-WHILE-ENDDO loop (more easily).

Other examples:

```
'-1' =>        ;branch back one line
0    =>        ;branch to the start of the current line
''   =>        ;branch to the start of the current line
999  =>        ;branch to next process
BRANCH '-1'    ;branch back one line
```

## Extension:

The BRANCH operator does have an extended form which is meant only for internal use (i.e. by the pre-processor). When debugging VPL code and looking at lines of code as they are executed then the user may notice that some DO-WHILE-ENDDO structures are replaced by an extended BRANCH. This takes the form:

```
BRANCH <offset,position,condition-code>
```

Again the user is warned not to use this, especially the "position" which is the position within the new line that execution will commence from. The reason for this is that the position of something in a line relative to the beginning of that line is modified by pre-processing (i.e. the line is packed). The default for position is the first and the default for the condition code is true.

```
Operator Name:            CREATE            !
                          ------            !
                                            --------------------

Class:          database

Arguments:      L   R

Result:         Yes, error code (0 if no error)

Summary:        CREATE      ->  err
           ln   CREATE      ->  err
                CREATE  reg ->  err
           ln   CREATE  reg ->  err
```

Description:
------------

This operator will create a new document in the database. The document will be empty and belong to the register given by the right argument. If no right argument is given then the current schematic name is used as the register name. The current schematic name can be read in system variable 403.

When a new document is "created" in the database then it will become the only document in an occurrence list. This is a handle to the newly created document which allows following operators such as PUT and PUTDOC to put data into that document.

The occurrence list number is given as the left argument to this operator. It should be in the range 1-101. List 101 is the current occurrence list and is assumed if no left argument is given. The previous contents of the given occurrence list will be replaced.

When a document is "created" it has one key field placed in it. The field name is "0" and its contents is the register name it belongs to.

If no database is open when this command is used then this is indicated by error code 47.

Example:

```
        CREATE             -> 0    ; Create a new document in the
                                   ; database with register name
                                   ; the same as s.v. 403.  The
                                   ; current occurrence list is
                                   ; used.  No errors results.
     3  CREATE             -> 0    ; Create a new document in the
                                   ; database with register name
                                   ; the same as s.v. 403.
                                   ; Occurrence list 3 is used.
                                   ; No error results.
    52  CREATE 'customers' -> 0    ; Create a new document with
                                   ; register name 'customers'.
                                   ; Occurrence list 52 is used.
                                   ; No error results.
        CREATE             -> 47   ; Attempt to create a document
                                   ; is unsuccessful because no
                                   ; database is open.
```

```
Operator Name:           DATETIME            !
                         --------            !
                                             ---------------------
```

Class: Special

Arguments:

Result: Yes

Summary: DATETIME -> yyyymmddhhmmssxxx

Description:
------------

This operator picks up a date time stamp from the host operating system if it is available and provides its result in the form of a 17 character string.

This operator takes no arguments.

The result is a 17 character string arranged in such a way as to make it suitable for sorting. Hence the year is first with 4 digits (A.D.) followed by 2 digits for the month (01 -> January, 12 -> December). The next is the day (2 digits) followed by the hour of the day. The hour of the day will be given in the 24 hour clock system. The next two digits are the minute followed by the second (2 digits) followed by 3 digits for milliseconds.

If, for example, the host operating only gives time resolution down to one hundreth of a second then the last digit in the string will always be zero.

The string only contains the numeric digits 0 1 2 3 4 5 6 7 8 9 .

Example:

```
        DATETIME    ->  19840514102941350
                        ====--==--==--===
                         !  ! ! ! ! !  !
                        year!day!min! millisecond
                            !   !   !
                          month !  sec
                                !
                               hour
```

Operator Name: **DBCLOSE**

Class: Database

Arguments:

Result: Yes , error code (0 -> no error)

Summary: DBCLOSE -> err

Description:

This operator will close the currently open database. All buffers held in memory associated with the database will be sent to secondary storage and the file will be closed.

When this operator is used all occurrence lists associated with the current database file are lost.

This operator has no arguments and returns an error code as the result.

If no database is open when this operator is used then error code 47 is returned indicating database not open.

System variable 401 contains the name of the last opened database while system variable 528 holds the current database status:

| Value in 528 | Meaning |
|---|---|
| 0 | No database currently open |
| 1 | A database without checkpoint is open |
| 2 | A database with checkpoint is open |

Orderly exit of VIPS (e.g. mode 8 and via the keyboard interrupt) will close the currently open database.

If a database with a checkpoint is open then this operator will perform a checkpoint as part of the database close (i.e. there is no need to have DBSAVE immediately before DBCLOSE).

If a disorderly exit is made (e.g. power fluctation, resetting the CPU) and a database is open without a checkpoint then it is potentially damaged.

Example:

```
#401      ->   'STOCK' ; database called 'STOCK' is
#528      ->   2       ; open with the checkpoint on
DBCLOSE   ->   0       ; it is closed successfully
#528      ->   0       ; now there is no database open
DBCLOSE   ->   47      ; so a further close causes an
                       ; error code to be returned
```

```
Operator Name:              DBOPEN                 !
                            ------                 !
                                                   --------------------
Class:          Database

Arguments:      L*  R

Result:         Yes , error code  (0 -> no error)

Summary:        str     DBOPEN   type    ->      err

                where    type  1 = open for read/write (default)
                              -1 = create new db - checkp. off
                              -2 = create new db - checkp. on
```

Description:

This operator will open a database file, creating it if requested.

This operator requires a left argument and can optionally have a right argument. The left argument should be a non-blank string obeying the host operating system's conventions for file names. If an extension is not given then the extension ".VDB" will be assumed.

The right argument to this operator is optional. If it is not given it is assumed to mean that the database should exist and be opened for read/write (equivalent to type=1 ). By opening a database for "read/write" is meant that the user can both SEARCH and GET data as well as PUT and DELETE data. If the database file does not exist or a file of that name does exist and is not a database then error code 48 is returned ("Not a Vista/Verbal database").

If the database previously existed and was created without a checkpoint then a disorderly exit (e.g. power fluctation, CPU reset) will leave a flag set within the database such that later attempts to open that database will result in error code 49 being returned ("Database left open?"). Such a database cannot be used by VIPS.

It is possible to create a new database with or without a checkpoint. To create a new database with checkpoint off then the right argument should be '-1'. To create a new database with checkpoint on then the right argument should be '-2'. In both cases of creating a new database it is then available for "read/write" interaction. Once a database has been created with the checkpoint on then the checkpoint will stay in force whenever that database is used. Once a database is created with checkpoint off then the checkpoint will not be available thereafter.

System variable 401 contains the name of the last opened database while system variable 528 holds the current database status:

| Value in 528 | Meaning |
|---|---|
| 0 | No database currently open |
| 1 | A database without checkpoint is open |
| 2 | A database with checkpoint is open |

If a database was open at the time this operator is executed then before an attempt is made to open the database given by the left argument:

a) If the previous database was without a checkpoint (#528=1) then it is simply closed.
b) if the previous database had a checkpoint (#528=2) then it is closed in such a way that updates since it was opened or since the last DBSAVE are ignored.

Examples:

```
'TEST'  DBOPEN       -> 0      ; a database file called
                               ; 'TEST.VDB' exists and
                               ; has been opened.
#528              ->  2        ; 'TEST.VDB' was created
                               ; with checkpoint on
#401              ->  'TEST'   ; as expected

'VISTA' DBOPEN  '-1' -> 0      ; create a new database
                               ; file called 'VISTA.VDB'
                               ; without checkpoint.
                               ; N.B. The previously
                               ; opened database file
                               ; 'TEST.VDB' would be
                               ; closed.
#528              ->  1
#401              ->  'VISTA'
```

Operator Name: DBSAVE

Class: Database

Arguments:

Result: Yes , error code (0 -> no error)

Summary: DBSAVE -> err

Description:

This operator will perform a checkpoint on the currently open database file if that file was originally created with the checkpoint option on. If the currently open database file was created with the checkpoint option off then all buffers associated with it are "washed" to disc.

The use of this operator in no way effects occurrence lists and the related document pointers within those occurrence lists.

A checkpoint is a mechanism for maintaining database integrity at a given point (i.e. when this operator is executed). A checkpointed file keeps all updates against the database in a special area until a checkpoint is performed or the file is closed. At this point the updates are consolidated in the database file. If the database is "crashed" in the interim period then all updates since the last checkpoint or database close are ignored.

Note that if a checkpointed database file is open and a new database file is opened then the updates against the original checkpointed database file are ignored (i.e. updates since its last checkpoint or close).

Examples:

```
'VISTA'  DBOPEN     -> 0   ; open database file
#528                -> 2   ; it is checkpointed
....
....                       ; perform a series of updates
....
        DBSAVE      -> 0   ; perform a checkpoint
```

```
Operator Name:          DBSIZE              !
                        ------              !
                                            --------------------
Class:        Database

Arguments:

Result:       Yes

Summary:               DBSIZE     ->     num
```

Description:
------------

This operator will return the size of the currently open database file measured in kilobytes as its result.

This operator requires no arguments. This operator returns a result.

The action of this operator is to return the size of the main database file as its result.

A temporary file is formed by the VIPS module which is used both by the database system and the VPL interpreter. The size of this file is not included in the result of this operator.

The figure returned by this operator is a measure in kilobytes of the amount of contiguous room being used by the database system in the main database file. Depending on the history of that file the figure returned by this operator can be less than the actual file size given by the host operating system, the same, and in some rare cases, slightly larger.

If no database is open when this operator is used then zero is returned as the result.

Examples:

```
'VISTA'  DBOPEN     ->  0   ; open database file
         DBSIZE     ->  28  ; 28 kilobytes of the file
                            ; 'VISTA.VDB' are being used
         DBCLOSE    ->  0   ; close database file
         DBSIZE     ->  0   ; no database currently open
```

```
Operator Name:              DELETE          !
                            ------          !
                                            ---------------------
Class:          Database

Arguments:      L

Result:         Yes , error code  (0 -> no error)

Summary:        ln      DELETE      ->      err
```

Description:
------------

This operator will delete a document in the database. The document must be in an occurrence list.

This operator can have a left argument. If it is given then this left argument will be the number (1-101) of the occurrence list from which the document will be deleted. If no left argument is given then the current occurrence list is assumed (101).

Within each occurrence list is a document pointer. This document pointer can be moved by the STEP operator. The deleted document will be the document addressed by the document pointer in the given occurrence list. The given occurrence list is not modified by the delete operation (i.e.the list length is the same). The document pointer is automatically STEPped to the next document (last steps to the first).

A deleted document remains in all the occurrence lists that it was in at the time of deletion. A deleted document can be read ( GET ) but attempts to modify it will return an error code. If an attempt is made to delete an already deleted document then nothing happens and error code 44 is returned ( Trying to delete a non-existent document ). Trying to write ( PUT ) to a deleted document will cause the same error.

After a document is deleted the SEARCH operator will no longer find it.

Examples:

```
    'smith'    SEARCH  'namreg:sname'  -> 4    ; Search for documents
                                               ; with 'smith' in field
                                               ; 'sname' in register
                                               ; 'namreg'.  4 found.
                                               ; Put in current list
               STEP                            ; Step to 2nd document
               DELETE                  -> 0    ; Delete 2nd document
                                               ; Step to 3rd document
```

An error situation:

```
               STEP         '-1'               ; Step back to 2nd doc.
               DELETE                  -> 44   ; Attempting to delete
                                               ; the second document
                                               ; again gives an error
```

```
Operator Name:          DO              !
                        --              !
                                        --------------------
```

Other operators
described here :     WHILE   ENDDO

Class: Control

Arguments: None for DO and ENDDO,  L  R  for WHILE

Result: No

Summary:

```
DO
    WHILE  ( condition-code )
ENDDO
```

Description:
------------

The three operators DO-WHILE-ENDDO form the basic looping structure in VPL. The DO and the ENDDO operators mark the beginning and end of the loop respectively. The WHILE operator controls the loop and can be found anywhere between the DO and the ENDDO. The WHILE operator need not be used or may be used one or more times within a loop.

DO-WHILE-ENDDO structures can be nested to any level. They must be nested wholly within one another. A DO-WHILE-ENDDO process must lie wholly within a process. If an ENDDO is missing then the loop will execute once if the WHILE condition is true, or the process will terminate if the WHILE condition is false.

The DO and the ENDDO operators do not take left arguments, and right arguments are not scanned for. These two operators do not return results.

The WHILE operator can have a left argument, a right argument , or both. The WHILE operator does not produce a result. If the WHILE operator does not have any arguments then an "Argument expected" error is generated.

The WHILE operator looks for the condition code in its right argument. If it has no right argument then its left argument is used as the condition code. For readability it is suggested that the condition code be given as an expression surrounded by parenthesis as the right argument to the WHILE operator.

If the WHILE condition is true then execution continues immediately after the WHILE operator (and its right argument). If the WHILE condition is false then execution continues following the corresponding ENDDO in the DO-WHILE-ENDDO structure.

The VPL interpreter currently supports the following conditional operators:

| | | | | |
|---|---|---|---|---|
| EQ | NE | GT | GE | LT |
| LE | EMPTY | NEMPTY | NUMERIC | NNUMERIC |
| SEQ | SNE | | | |

and the following operators for combining the above conditional operators:

| | |
|---|---|
| AND | OR |

All these conditional operators yield 1 or 0. 1 implies true. 0 implies false. An arithmetic expression can be used to generate a condition code. In this case all numbers between '-0.5' and '0.5' are taken as false while all other numbers are taken as true.

Care should be taken with the use of BRANCH together with DO, WHILE, ENDDO structures. BRANCHing into such loops from outside is especially dangerous and not a recommended programming practice. (N.B. Such an action is still well-defined from the point of view of the VPL interpreter).

The interpreter places a limit on the number of DO-WHILE-ENDDO structures which can be found on one line. The limit is 5 sets. Hopefully no-one would put more than one DO-WHILE-ENDDO structure on one line.

Examples:

```
5  = #201                 ;initialize loop counter
DO                        ;start loop
WHILE (#201 GT  0 )       ;exit loop if loop counter zero or
                          ; less
    #3 * #201= #3         ;perform five times
    #201 - 1 = #201       ;decrement loop counter
ENDDO                     ;end loop
....                      ;continue here when WHILE condition
                          ;fails
```

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| Operator Name: | | DOCDECOD | | | |
| Class: | Database | | | | |
| Arguments: | L* R | | | | |
| Result: | Yes | | | | |
| Summary: | num | DOCDECOD | ln | -> | str |

Description:

This operator is designed for dismantling a document about which there is very little known. It can be used for debugging purposes.

Documents are stored with a field name associated with a string of data. If the string of data is null or only contains spaces then nothing is stored. The data in a document is usually fetched out on a field by field basis using the same field names under which it was stored. A problem may arise if the user does not know what field names were used when the document was stored. The operators which support the normal transfers of data to and from documents are: PUT, PUTDOC, GET, and GETDOC.

Documents are stored within the database system as a series of "lines". Each line contains a field name, a string of data, and an indicator whether the field is key or non-key. These lines within a document are sorted by field name. If the field name is numeric then it has leading zeros put on it so it is always at least three characters long (e.g. field name '3' becomes '003'). The 0th line in a document does not obey this rule and has field name '0' with a string which is the register name this document belongs to. The 0th line is a key.

The DOCDECOD operator can decode documents line by line. The right argument is the occurrence list number (1-101). If there is no right argument the current occurrence list is assumed (101). The document addressed by the document pointer of the given list is decoded. The result is the string of data the document line contained. A left argument must be given. It must be a number. The meanings of this number are listed below:

| Left argument to DECDECOD | Meaning |
|---|---|
| -1 | get the 0th line (register name) |
| 0 | get the next line (1st to start with) |
| 1 | get 1st line |
| 2 | get 2nd line |
| 3 | get 3nd line |
| etc. | |

The result of this operator is the string of data contained in a line. Several system variables give more information.
System variable 512 gives the field name (and field extension if there is one) associated with the last use of DOCDECOD.
System variable 513 gives the line number of the last line fetched by DOCDECOD. This line number is negated if the line is non-key.
System variable 523 returns '1' if the last line fetched by DOCDECOD was the last line of that document, otherwise it returns 0.

If a left argument of 0 is used by DOCDECOD then all the lines of the document can be viewed. After the last line is fetched then the first line is fetched again so that any loop based on DOCDECOD must look at system variable 523 for its termination condition.

Examples:

```
        SEARCH  'NAMREG:'    ->  34  ;get all the documents in the
                                     ;'NAMREG' register into the
                                     ;current list. 34 found.
        STEP                         ;step to the second document

  '-1'  DOCDECOD      -> 'NAMREG'    ;contents of the 0th field is
                                     ; the register name: 'NAMREG'
        #512          ->  '0'        ;field name '0'
        #513          ->   0         ;0th field is key
        #523          ->   0         ;it is not the last line

    0   DOCDECOD      -> 'The Grange';contents of 1st line
        #512          ->  'ADDRESS'  ;field name
        #513          ->   1         ;1st field is key
        #523          ->   0         ;it is not the last line

    0   DOCDECOD      -> 'John'      ;contents of 2nd line
        #512          -> 'FNAME'     ;field name
        #513          ->   -2        ;2nd field is non-key
        #523          ->   0         ;it is not the last line

    0   DOCDECOD      -> 'Smith'     ;contents of 3rd line
        #512          ->  'SNAME'    ;field name
        #513          ->   3         ;3rd field is key
        #523          ->   1         ;it is the last line
```

```
Operator Name:              EMPTY                 ! for ELSE see IF
                            -----                 !
                                                  ---------------------
Other operators
described here:             NEMPTY

Class:          String, conditional

Arguments:      L   R   (at least one, right takes precedence)

Result:         Yes, condition code

Summary:        str  EMPTY      ->     cc
```

Description:
------------

The EMPTY and the NEMPTY (read "not empty") operators return condition codes depending on whether their arguments are space filled or not.

Both operators need either a left argument or a right argument. If they have both then the right argument is taken. For readability it is recommended that only a left argument is used.

The EMPTY operator returns the true condition code (i.e. 1 ) if its argument is empty. That is to say it is full of spaces or it is of length zero. If its argument is non-empty (i.e. contains some other character apart from space) then the false condition code (i.e. 0 ) is returned.

The NEMPTY operator returns the false condition code if its argument is empty. That is to say it is full of spaces or it is of length zero. If its argument is non-empty (i.e. contains some other character apart from space) then the true condition code is returned.

Examples:

```
IF  #3 EMPTY  THEN            ; if field 3 empty then
    '****'=#3                 ; put 4 "*" in field 3
ENDIF                         ; end of IF structure

DO                            ; DO loop
WHILE (#201 NEMPTY )          ; while field 201 is not empty
    ....                      ; ....do something
ENDDO                         ; end of DO structure
```

```
Operator Name:              EQ                      ! for ENDDO see DO
                            --                      ! for ENDIF see IF
                                                    ------------------

Other operators
described here:     NE      GE      GT      LE      LT

Class:          Arithmetic, conditional

Arguments:      L*  R*  R2

Result:         Yes, condition code

Summary:        num  EQ  num          ->  cc
                num  NE  num          ->  cc
                num  GE  num          ->  cc
                num  GT  num          ->  cc
                num  LE  num          ->  cc
                num  LT  num          ->  cc

                num  EQ <num,fuzz>    ->  cc
```

Description:

These six operators will compare numbers and return a condition code based on the ordering of these numbers.

All six operators require both a left and right argument. Both arguments must be decodable as numbers. A second right argument can be given. If so it must be a number. The result is a condition code, 1 for true, 0 for false.

The comparison is quite straight forward if the two number involved can be represented internally as integers because this is an exact form. In the case of two integers being compared the 2nd right argument will be ignored.

If either of the numbers being compared cannot be represented as integers then the following information should be taken into account.

Firstly, if a 2nd right argument is given then it will be taken as the value for fuzz. If a 2nd right argument is not given then the value in system variable 517 will be taken. The value for fuzz should be in the range 0 up to the maximum number of digits precision given by double precision reals on the host machine (e.g. IEEE gives 16 digits).

In the following:

| | |
|---|---|
| num1 | is the first number being compared |
| num2 | is the second number being compared |
| num$ | is the larger of the two in magnitude |
| fuzz | comparison tolerance index |
| ! ! | indicates the absolute value of |
| * | multiplication |
| < | is less then |
| ** | exponentiation |

```
1)  if  ! num1 - num2 !   <  ! num$ ! * 4.9 * (10 ** (-1-fuzz))
    then  num1  and num2  are taken to be equal.

2)  if  ! num$ !  <  10 ** (-1-fuzz)
    then  num1  and num2  are taken to be equal.
```

If either condition 1 or 2 is met then then num1 and num2 are taken to be equal. If neither condition 1 nor 2 is met then num1 and num2 are taken to be unequal.

In practical terms this means that if fuzz (s.v. 517) is set at 10 (which is the current default) then there is an uncertainity in a comparison of 1 cent in 200,000,000 dollars. This type of accuracy should be sufficient for most applications. Taking the fuzz too close to the number of digits precision claimed by the manufacturer runs the risk of comparisons such as: 2/3 EQ ( 1/3*2 ) failing.

Examples:

```
 1     EQ    1    ->  1        ; always true
 1     GT    1    ->  0        ; always false
'1'    LE    1    ->  1        ; always true

 1     EQ  <1,1>  ->  1        ; true, fuzz makes no
                               ; difference to integers

 0.6   EQ  <1,0>  ->  1        ; true since 0.6 and 1 are
                               ; within 0.5 of one another
```

```
Operator Name:           ERROR                  !
                         -----                  !
                                                --------------------
Class:          Special

Arguments:      L*

Result:         Yes, same as left argument

Summary:        num  ERROR     ->     num
```

Description:

This operator will accept an error code given as its left argument and display the appropriate system error message. The result of this operator is its left argument (e.g. if 3 is the left argument then 3 will be the result).

The left argument must be given and it must be a number. The result will be the same number.

Errors in VPL can be divided into two classes:

1) The simplest group are those associated with incorrect or missing punctuation (e.g. unmatched parenthesis, unmatched quotes) and missing or incorrect arguments to an operator. This class of error will usually cause a VPL error to appear on the status line when the VPL interpreter recognizes the mistake. Most VPL errors are prefixed by the string "** VPL **" followed by a brief explanation of the error.

2) The other class of error are those returned as error codes by various operators. These errors tend to be higher level and indicate the action associated with the operator was not performed. The error code gives some reason for the failure (or indicates that the operator worked). For example, if a sequential file is to be opened to read access then the SOPEN operator will return error code 52 if the file is not found.

The VPL interpreter currently supports over 50 error messages. Each one of these error messages has a corresponding error code.

The error code zero (0) is reserved to indicate no error. If the ERROR operator is given zero as its left argument then it does nothing apart from providing zero as its result.

Positive error code numbers indicate an error has been detected by the VPL interpreter. If the ERROR operator is given a positive number as its left argument it will place the corresponding error message on the status line. For a list of positive error codes see Appendix B.

A negative error code indicates that something is wrong with the database file, or the system file, or the user schematic file. Negative error codes are serious and if the VPL interpreter cannot continue then VIPS may be aborted. When negative error codes are given to the ERROR operator the error message "** VPL ** Verbal Filing System error number:" followed by a negative number is placed on the status line. The meaning of this negative error number (so-called VFS error) is listed in Appendix B.

In the case of a non-zero number being given to the ERROR operator then the corresponding error message is placed on the status line after which the cursor waits at the end of the status line for user input. Three different things can happen depending on the next keypress:

1) If the down arrow key is pressed then the offending line is displayed with the symbol "??" a little to the right of the position the error was recognized. Because of pre-processing the line may not look exactly the same as the original. This is due to redundant spaces being removed and comments being stripped off (to speed execution). The cursor again waits at the end of the status line for user input. The next keypress will invoke either action 2) or 3) below:

2) If F8 is pressed then the user is asked the following question: "Exiting current process: continue? (Y/n). If the user responds with a "Y" or a carriage return then the current process is aborted as if F8 was hit (equivalent to having the expression " EXIT 8" inserted in the code). If the answer of "N" (or "n") is given then VIPS returns to the host operating system after closing all open files.

3) If any other key is pressed then execution continues at the point immediately following the ERROR operator.

Examples:

```
          0       ERROR   ->  0       ; no error so this operator
                                      ; does nothing

    'FRED'  DBOPEN  ERROR = #201 -> 49  ; attempt to open the database
                                      ; 'FRED.VDB' is unsuccessful
                                      ; as indicated by the non-zero
                                      ; error code returned
```

The ERROR operator would cause the following to appear on the status line:

"Database left open? "

The cursor would then wait at the right hand end of the status line. If down arrow is pressed then the following would appear on the status line:

" 'FRED' DBOPEN ERROR ?? = #201 "

The cursor would wait again at the end of the status line awaiting input. If the space bar was pressed then execution of VPL would re-commence with the assignment of 49 into hidden field 201

P.S. This particular error message would indicate that the database file 'FRED.VDB' was not created with a checkpoint and in its last usage was not closed properly. Maybe the user pressed the CPU reset button?

Operator Name: EXECUTE

Class: Special

Arguments: L*

Result: If type 0 or 1 execute and the executed string yielded a result then it yields a result, else no result

Summary: str EXECUTE -> ??

Description:

This operator treats its left argument as a VPL expression and executes it.

This operator must have a left argument.

The left argument can be anything recognizable by the VPL interpreter as a VPL statement (optionally including a comment). The string must not contain a label. The string should not contain an operator causing a control transfer (EXIT, GOTO, or BRANCH although BRANCH 0 is allowed). The string can contain IF-THEN-ELSE-ENDIF and DO-WHILE-ENDDO structures as long as they are wholly contained in the string.

There are three types of actions allowed with the EXECUTE operator. These types of actions are reflected by the numbers 0,1, and 2 in system variable 418. The system is initialized to 1 in s.v. 418. System variable 418 can be written to.

Type 0) The EXECUTE operator will return the result from the string it is executing as this operators result. The executed string can itself contain EXECUTE operators. A structure similar to subroutines can be envisaged, each EXECUTE wholly nested within the other. If necessary this can be done to many levels but there is a slight time and space penalty (on the temporary file) associated with every extra level.

Type 1) The EXECUTE operator will return the result from the string it is executing as this operators result. If the executed string had a pending operator (i.e. one looking for a right argument) when the interpreter reached the end of the string then this operator is carried out into the context of the original EXECUTE operator. In a sense the operator is returned as the result. The executed string can itself contain EXECUTE operators. Such imbedded EXECUTEs do NOT form a subroutine structure but rather chain to one another. When any of these imbedded EXECUTEs (no matter where in the chain) reaches the end of its string then the VPL interpreter continues after the original EXECUTE statement. The first usage of the EXECUTE causes a slight time and space penalty (in the temporary file) but imbedded EXECUTEs add virtually no overhead.

Type 2) The EXECUTE operator will not return a result. The rest of the line the EXECUTE operator was found on will be ignored and the VPL interpreter will continue at the beginning of the following line. The executed string can itself contain EXECUTE operators. Such imbedded EXECUTEs do NOT form a subroutine structure but rather chain to one another. When any of these imbedded EXECUTEs (no matter where in the chain) reaches the end of its string then the VPL interpreter continues at the beginning of the line following the original EXECUTE statement. This method of EXECUTE is the fastest and entails virtually no overhead.

If system variable 418 is changed within an executed string then great care should be taken!

Examples:

```
          "  1  +  3  "  EXECUTE    =  #2  ->  4
```

Assuming type 0 or 1 EXECUTE then the left argument to the EXECUTE operator would be evaluated by the interpreter and the number 4 would be placed in screen field 2. The result of the EXECUTE operator (and the whole expression) would be 4.

Extension:
---------

In types 0 and 1 of the EXECUTE operator then the passed parameter result %0 can be used as a local variable. It can be viewed as a variable length hidden field with a length between 0 characters and 255 characters.

Operator Name: EXIST

-----

| | |
|---|---|
| Class: | Special |
| Arguments: | L |
| Result: | Yes ( 1, 0 or -1 ) |
| Summary: | str EXIST -> 1, 0, or -1 |

Description:

This operator returns an indicator of the kind of left argument it has. In particular whether it exists or not and if so whether it can be written to.

This operator may have a left argument. The left argument may be a passed parameter (e.g. %11) which has not been defined. The result is either 1, 0, or -1.

This operator will determine the existence and "writability" of its left argument. The results of this operator have the following meaning:

| Result | Meaning |
|---|---|
| 1 | Left argument exists and is writeable (and readable).<br>e.g. existing screen fields, defined status line fields, defined hidden fields, passed parameters which represent writeable variables, and writeable system variables. |
| 0 | Left argument does not exist.<br>Either there is no left argument or the indicated field does not exist or passed parameter was not defined in the invocation of the procedure. |
| -1 | Left argument exists and is readable (not writeable) This could either be an explicit constant, the result from an operator, a system variable which is read only, or a passed parameter representing one of these |

Examples:

```
12     EXIST   ->    -1  ; an explicit constant is
                         ; only readable

#201   EXIST   ->     1  ; a hidden field is
                         ; readable and writeable

#845   EXIST   ->     0  ; no such field
```

Assuming a procedure call TEST is invoked as follows:

```
#1  TEST   <'name: ',33> = #2
```

then in the procedure definition of TEST the following would be observed:

```
%0      EXIST   ->      0   ; result doesn't exist yet

%1      EXIST   ->      1   ; if the current schematic
                            ; has 1 or more fields

%2      EXIST   ->      0   ; this procedure invocation
                            ; does not have a right
                            ; argument, it has a right
                            ; argument list.

%11     EXIST   ->     -1   ; first element of right
                            ; argument list is constant
                            ; it is:  'name: '

%12     EXIST   ->     -1   ; second element of right
                            ; argument list is constant
                            ; it is:  33

%13     EXIST   ->      0   ; third element of right
                            ; argument list not given


222 +   36  =   %0          ; the result passed
                            ; parameter is unique in
                            ; VIPS.  Even though it
                            ; didn't exist previously
                            ; it can be written to.
%0      EXIST   ->      1   ; and now it exists and is
                            ; writeable!
```

```
Operator Name:              EXIT              !
                            ----              !
                                              --------------------
Class:          Control

Arguments:      L  R  (if both right is taken)

Result:         No

Summary:        num     EXIT
                        EXIT    num     ; where num is from 1 to 8

N.B.            Illegal inside a procedure
```

Description:
------------

This operator unconditionally leaves the current process. If it has an argument then it is a number between 1 and 8 which simulates the situation in which the corresponding function key (i.e. F1 to F8) was pressed and in which the process has finished. If no argument is given then the previously pressed function key (reflected by system variable 453) is assumed.

Either a left argument or a right argument can be given. If both are given then the right argument is used. The argument must be a number in the range 1 to 8. This operator has no result.

The process to which control is passed will depend on the process in which the EXIT operator is executed. The possibilities are listed below:

1) A screen field process will exit to the first line of the END process.
2) The END process will exit to the first line of the SUPER END process.
3) The SUPER END process will exit to the first line of the SUPER BEGIN process.
4) The SUPER BEGIN process will exit to the first line of the SUPER end process.
5) The BEGIN process will exit to the first line of the END process.

System variable 453 contains the function key exit code. It will contain the values 1 to 8 or 255. The values 1 to 8 indicate that F1 to F8 have been pressed during input, and 255 indicates no function key has been pressed. If this operator has an argument which is a number in the range 1 to 8 then this number is placed in s.v. 453. The SUPER BEGIN process always places the value 255 in s.v. 453 before it is executed. System variable 453 is writeable. Only the values 1 to 8 and 255 are meaningful.

Examples:

```
          EXIT    8                   ; leave the current process now
                                      ; and place 8 in s.v 453
```

Assume this is a screen field process:

```
          INPUT                       ; get input from current field
          IF #509 LT 0 THEN           ; if last key pressed was function
              EXIT                    ; key then exit to END process
          ENDIF                       ; otherwise continue processing
```

If a function key was pressed during the INPUT operator this latter method would convey the number of the function key via system variable 453 to the END process.
Note that whenever a function key is pressed during input that it is recorded in both #509 (negated) and #453. It is always safer to test #509 first because some key must be pressed (not necessarily a function key) in order to return from keyboard input.

```
Operator Name:        EXP                    !
                      ---                    !
                                             ---------------------

Class:          Arithmetic

Arguments:      L  R  (one or the other, right takes precedence)

Result:         Yes

Summary:        num    EXP              ->  num
                       EXP   num        ->  num
                       EXP  (num)       ->  num
                ;these three are equivalent
```

Description:
------------

This operator performs a natural exponentiation. The result is the given argument after it has been used to raise "e" to that power.

This operator requires an argument. It can be either a left argument or a right argument. If both a left argument and a right argument are given then the right argument is used. The argument must be a number. This operator returns a result.

The number "e" is approximately 2.71828182845904 . The result is the number obtained by raising the constant "e" to the given argument. The given argument can be any number less then 300. This upper limit is chosen because the result would have approximately 250 significant digits and larger numbers would overflow the 255 characters limit on strings. In practice this should not be a significant limitation.

If an argument 300 or greater is given then a "** VPL ** Attempt to divide by zero" error message will be placed on the status line.

Examples:

```
         1   EXP              ->  2.71828182845904
             EXP  (1)         ->  2.71828182845904
        10   EXP  (1)         ->  2.71828182845904

        10   EXP              ->  22026.4657948067
```

To get the quadratic root of a number this operator could be used together with the LOG operator (takes natural logarithms).

```
        16  LOG  /  4  EXP  ->  2
```

Operator Name: **FIND**

|  |  |
|---|---|
| Class: | String |
| Arguments: | L* R1 R2 R3 |
| Result: | Yes |

```
Summary:        str  FIND                    ->  pos
                str  FIND   pos              ->  pos
                str  FIND  <pos,char>        ->  pos
                str  FIND  <pos,char,type>   ->  pos
```

Description:

This operator will find the position of the first occurrence of a character after a given position within a string.

This operator must have a left argument. This operator may have three right arguments. If the first or third right arguments are given they must be numbers. This operator returns a result.

The left argument is the string which will be examined. If it is a null string then zero will be returned as the result.

The first right argument is the position in the string after which the character will be looked for. If the position is not given it is assumed to be zero so that the search starts at position 1 in the string.

The second right argument is the character which is being looked for in the string. If not given or a null string then it is assumed to be space. If the second right argument contains more than one character then the first is taken.

The third right argument is the type. It should be zero or 1. If not given it is assumed to be zero.

When the type is zero this operator will return the position of the first occurrence of the given character. If the character does not occur then zero is returned as the result.

When the type is 1 this operator will return the position of the first non-occurrence of the given character. If there are no non-occurrences (i.e. the string only contains the given character) then zero is returned as the result.

Examples:

```
    'Paul Landa'   FIND            ->     5   ;space in 5th position

    'Paul Landa'   FIND  5         ->     0   ;no space after 5th pos

    'Paul Landa'   FIND  <,'a'>    ->     2
```

```
'Paul Landa'    FIND  <2,'a'>     ->     7

'Paul Landa'    FIND  <7,'a'>     ->     10

'Paul Landa'    FIND  <10,'a'>    ->     0    ;no more 'a's left

'Paul Landa'    FIND  <,'a',1>    ->     1    ;first non-occurrence
                                              ;of 'a'

'Paul Landa'    FIND  <1,'a',1>   ->     3

'Paul Landa'    FIND  <3,'a',1>   ->     4

'    '          FIND              ->     1    ;first space

'    '          FIND  <,,1>       ->     0    ;first non-space
```

Operator Name: FLENG !
----- !

---------------------

Class: Special

Arguments: L

Result: Yes

Summary: num FLENG -> num

Description:

This operator returns the length in character positions of the variable corresponding to the given number.

This operator should have a left argument. If it is given, it must be a number. Non-integers are rounded to integers if necessary. This operator returns a result.

The left argument of this operator should be a non-negative. If the left argument is not given then zero (i.e. current screen field) is assumed. Numbers of screen and status line fields will return the length in characters of the indicated field as the result. If the screen or status line field does not exist then zero is returned as the result. Short hidden fields return the number 16 while long hidden fields return the number in system variable 519 (length of long hidden fields). Numbers associated with system variables return -1. Other numbers return -1.
Summary of left arguments to this operator:

| Left argument | meaning |
|---|---|
| 0 | return length of current screen field |
| 1-200 | return length of field if it exists else zero |
| 201-300 | return 16 for short hidden fields |
| 301-400 | return value currently in s.v. 519 which contains length of long hidden fields |
| 401-599 | return -1 |
| else | return -1 |

Examples:

| | | | | |
|---|---|---|---|---|
| 33 | FLENG | -> | 12 | ;screen field 33 is 12<br>;character positions long |
| 159 | FLENG | -> | 0 | ;current schematic does not<br>;have such a field number |
| 901 | FLENG | -> | 79 | ;first status line field<br>;is 79 characters long |
| 902 | FLENG | -> | 0 | ;second status line field<br>;is not defined |
| 201 | FLENG | -> | 16 | ;short hidden fields give<br>;16 |

```
#519            ->  80      ;length of long hidden
                            ;fields
401    FLENG    ->  80      ;long hidden fields would
                            ;therefore give 80
519    FLENG    ->  -1      ;system variables give -1

1234   FLENG    ->  -1      ;unknown
```

|  |  |  |  |  |
|---|---|---|---|---|
| Operator Name: |  | FOLD |  |  |
| Class: | String |  |  |  |
| Arguments: | L* R |  |  |  |
| Result: | Yes |  |  |  |
| Summary: | str FOLD |  | -> | str |
|  | str FOLD | type | -> | str |

Description:

This operator will change strings from lower case to upper case, from upper case to lower case, or the first letter of each word from lower case to upper case.

The left argument must be given. If a right argument is given it should be either 1 or '-1'. The result will have the same number of characters in it as the left argument.

In the simplest case (with no right argument) the result will be the left argument after all characters have been made upper case (fold to upper case).

If the right argument is '-1' then the result will be the left argument after all characters have been made lower case (fold to lower case).

If the right argument is '1' then the result will be the left argument after the first character of each word has been made upper case. The default delimiter between words is space. If the first position in the left argument is non-blank then it is taken to be the start of a word.

When the first letter of each word is being folded to upper case (when 'type' is '1') it is possible to define word delimiters other than space. Two system variables are available for this purpose. These are #524 and #525. Thus two different characters can be taken as delimiters. Both #524 and #525 are initialized to space.

Examples:

```
'abcdef'      FOLD              ->  'ABCDEF'

'ABCDEF'      FOLD    '-1'      ->  'abcdef'

'this is a test'  FOLD          ->  'THIS IS A TEST'

'this is a test'  FOLD  '1'     ->  'This Is A Test'

'this is a test'  FOLD  <'1'>   ->  'This Is A Test'
```

```
':' =   #524
':' =   #525
'this is a:test'  FOLD  '1'      -> 'This is a:Test'
                                 ;N.B. the first character of the
                                 ;      result is upper case


' ' =   #524
':' =   #525
'this is a:test'  FOLD  '1'      -> 'This Is A:Test'
```

Operator Name: FSTAT

Class: Special

Arguments: L

Result: Yes

Summary: num FSTAT -> num

Description:

This operator will return a value indicating whether the screen field corresponding to the given number is key or non-key and what its verification is.

This operator should have a left argument. If it is given it must be a number. Non-integers are rounded to integers if necessary. This operator returns a result.

If the left argument is not given then the current screen field is assumed. If the left argument is given it must be a number in the range 1 through 200 or 291 through 299.

The value of the result is a numerical encoding of the verification associated with the screen field when it was defined by the SKJEMA program.

| Absolute value of result | meaning | As coded in SKJEMA |
|---|---|---|
| 0 | screen field not defined | |
| 32 | key field, no verification | (" ") |
| 65 | key field, no verification | ("A") |
| 66 | key field, alphabetic, digits, space | ("B") |
| 68 | key field, digits, /, comma, +, space | ("D") |
| 69 | key field, everything, push left at right | ("E") |
| 70 | key field, as 69 but digits, comma,+, space | ("F") |
| 77 | key field, space, digits, comma, + - | ("M") |
| 78 | key field, space, digits, comma, + | ("N") |
| 80 | key field, space, digits, comma, stop, + | ("P") |
| 81 | key field, space, digits, comma, stop, + - | ("Q") |
| 90 | key field, space, alphabetic | ("Z") |
| -97 | non-key ", no verification | ("a") |
| -98 | non-key ", alphabetic, digits, space | ("b") |
| -100 | non-key ", digits, /, comma, +, space | ("d") |
| -101 | non-key ", everything, push left at right | ("e") |
| -102 | non-key ", as 69 but digits, comma,+, space | ("f") |
| -109 | non-key ", space, digits, comma, + - | ("m") |
| -110 | non-key ", space, digits, comma, + | ("n") |
| -112 | non-key ", space, digits, comma, stop, + | ("p") |
| -113 | non-key ", space, digits, comma, stop, + - | ("q") |
| -122 | non-key ", space, alphabetic | ("z") |

The result is a positive number if the indicated screen field was defined as a key field when it was defined in the SKJEMA program. The result is a negative number if the indicated screen field was defined as a non-key field when it was defined in the SKJEMA program. The result is zero if there is no such screen field in the current schematic.

Examples:

```
        1   FSTAT   ->      66        ;screen field 1 was defined as
                                      ;a key field with type "B" ver.
        2   FSTAT   ->      -97       ;field 2 is non-key with type
                                      ;"a" verification
        99  FSTAT   ->      0         ;there is no screen field 99
```

```
Operator Name:          GET                        ! for GE see EQ
                        ---                        !
                                                   ---------------------
Class:          Database

Arguments:      L*  R

Result:         Yes

Summary:        fds     GET                 ->  str
                fds     GET     list        ->  str
```

Description:
------------

This operator will fetch the contents of a field from the current document of the given list. The required field is addressed by its field descriptor. The result is the contents of the addressed field. If the field descriptor is not valid for that document or the given list is empty then a null string (length 0) is returned.

This operator requires a left argument. This left argument must fit the format given below for a field descriptor (e.g. it cannot have 2 ":"s in it). If this operator has a right argument then it must be a number and in the range 1 to 101. List number 101 refers to the current list. If no right argument is given then the current list is assumed. This operator will always have a result. If nothing is found then this result will be a null string.

The left argument should either be a blank (null) string or contain a field descriptor. If it is blank (or null) then a null string is returned as the result of this operator. Otherwise it will be interpreted as a field decriptor.

The format of the field descriptor is as follows:

```
        reg:nam.ext

            where:
                    reg  is the register name (ignored by GET)
                    nam  is the searchable part of name
                    ext  is the non-searchable part of name
```

The register name is not required by the GET operator and will be ignored. It may be useful to have the register name present from the point of view of checking that the register name is the same as that which the referenced occurrence list was generated by. In the future the interpreter may check this.

The searchable part of the name must be given and be non-blank. The field name "0" (zero) is reserved for a field containing the register name of the document (put in there by the CREATE operator). Two methods of field naming are supported. The first method is by number, in which the field name can contain up to three digits. The second method is by a string which can be up to 31 characters long and must not start with a digit (or contain ":", ".", or blank).

The extension is optional and can be up to 3 alphanumeric characters long. If the field was defined with an extension (i.e. by a PUT operator) then the same extension must be given to the GET operator which fetches it.

System variable 513 is modified by the execution of a GET operator. It is set to a positive number if the fetched field was stored as a key and is set to a negative number if the fetched field was stored as a non-key. #513 is set to zero if the field is not found or the register name field (field 0) is fetched.
The magnitude of the value placed in #513 is the line number within the document the field descriptor and its contents are stored in. This information about the internal line number is useful for debugging. See opertor DOCDECOD if more information is required about this.

If a database is not open when the GET operator is used then a null string is returned.

Examples:

```
        'Parramatta'    SEARCH  'owners:town'    ->  3

                                            ; find all documents in the
                                            ; register called 'owners'
                                            ; which have 'parramatta' in
                                            ; a field called 'town'.
                                            ; 3 documents found and placed
                                            ; in current list.

       ; N.B. after a search the document pointer points to the first
       ; document in the resultant occurrence list.

        'town'  GET     -> 'Parramatta'  ; get contents of field called
                                            ; 'town'. Result as expected!

        'owners:town'   GET      ->  'Parramatta'
                                            ; the register name is
                                            ; currently ignored.
```

Assuming that there are fields in the document called: '3', 'address', 'note.1' and 'note.2'
then:

```
        3        GET     ->  'suburb'     ; the field called '3' contains
                                            ; 'suburb'

        'note.1'  GET    ->  'West, Cumberland'
        'note.2'  GET    ->  ' 922-2222 '
        'note'    GET    ->  ''            ; if there is no such field
                                            ; then a null string is
                                            ; returned
```

Operator Name: GETDOC

Class: **Database**

Arguments: **L R**

Result: Yes, error code

Summary: list GETDOC type -> err

Description:

This operator will fetch the current document in the given occurrence list and place it in the screen fields.

This operator may have a left argument. If so, it must be a number. This operator may have a right argument. If so, it must be a number. Non-integers are rounded to integers if necessary. This operator returns a result which is an error code.

The left argument is a list number. It should be in the range 1 to 101. Occurrence list 101 is referred to as the current occurrence list and is assumed if no left argument is given.

The right argument is the type. If given it should either be 0 or 1. If the right argument is not given then a type of zero is assumed. If the type is zero then all screen fields are cleared before an attempt is made to read the document. If the type is 1 then the screen fields are not cleared before an attempt is made to read the document. Therefore when the type is 1 screen fields which do not have a counterpart in the document are left unaltered by this operator.

The result of this operator is an error code. If the operation is successful then zero is returned. If there is no database open then error code 47 is returned.

The action of this operator is to get the fields out of the current document in the given occurrence list and put the contents of these fields into the corresponding screen fields.

Currently the fields in a document are named. Each field in a document can have up to a 31 character field name and optionally a three letter extension. The field name in a document can start with either an alphabetic character or a numeric character (i.e. 0 to 9). Screen fields, however, are numbered in sequence by the SKJEMA program which is used to create schematics. In the future it will be possible to optionally associate a field name and an extension to a screen field. To distinguish the compulsory screen field number from the optional screen field name, the latter must not commence with a numeric character (i.e. 0 to 9).

This operator needs to map field names in a document to screen fields which are currently only numbered but in the future will be optionally named as well. This operator will decide whether a field name in a document corresponds to a screen field number or a screen field name on the basis of the first letter of the field name in a document. If it is a numeric character (i.e. 0 to 9) then it maps to a screen field number. If it is not a numeric character then it maps to a screen field name (optional extension). As currently implemented the GETDOC operator will ignore fields in a document whose names do not begin with a numeric character.

Example:

GETDOC is a convenience operator for getting the contents of a document in a list onto the screen with the minimum of fuss. To demonstrate its action the following example shows the definition of the procedure GETDOCC which is written in terms of more primitive operators and functionally the same as GETDOC.

The procedure is listed on the next page. Note that some effort is put into checking the validity of the parameters passed to this procedure.

```
;  list_number  GETDOCC   type      ->  error_code
;
; Purpose:
;     VPL procedure to perform same action as GETDOC operator
; Input:
;     list_number      should be 1 to 101, 101 assumed if not given
;     type             not given     ->  blank all screen fields first
;                      =0            ->  blank all screen fields first
;                      =1            ->  don't blank screen fields
; Output:
;     error_code       =0      successful
;                      =47     database not open
; Fields used:
;     #291, #292, #293, #527
;
; Programmed by:
;     Douglas Gilbert, NORSOFT A/S,  840623
;
     0   =   %0               ;assume successful result
     101 =   #291             ;default list number
     0   =   #292             ;default type
     IF  %1 EXIST  THEN            ;if the left argument exists
         IF  %1 NUMERIC  THEN      ;if the left argument is numeric
             %1  =   #291          ;then overwrite default type
         ENDIF
     ENDIF
     IF  %2 EXIST  THEN            ;if the right argument exists
         IF  %2 NUMERIC  THEN      ;if the right argument is numeric
             %2  =   #292          ;then overwrite default type
         ENDIF
     ENDIF
     IF #528 EQ 0 THEN
         47  =   %0                ;if no DB open then return error
         RETURN                    ;code 47
     ENDIF
     IF #292 NE 1 THEN
         BLANK                     ;if type isn't 1 blank all fields
     ENDIF
     IF #291 LLENG EQ 0 THEN
         RETURN                    ;if the nominated list is empty
     ENDIF
     1   =   #293                  ;initialize loop variable
     DO                            ;start loop
         #293 DOCDECOD #291 = #527     ;decode line of document
                                       ;s.v. 512 get field name
         IF #512 NUMERIC THEN          ;if field name numeric
             IF #512 LE #447 THEN      ;if field name less than
                                       ; highest screen fld number
                 #527 = ##512          ;then place contents of line
                                       ;on screen (N.B. indirection)
             ENDIF
         ENDIF
         #293 + 1 = #293               ;increment loop variable
     WHILE (#523 NE 1)                 ;set to 1 by DOCDECOD when last
                                       ; line of document
     ENDDO                             ;loop back if "while" true
     RETURN
```

Operator Name: GOTO

Symbolic representation: =>>

Class: Control

Arguments: L R (one or other, if both right is taken)

Result: No

Summary:

```
num     GOTO
        GOTO    num
```

N.B. Illegal inside a procedure

Description:

This control operator will unconditionally transfer control to either a screen field process or the BEGIN, END, SUPER BEGIN, or SUPER END process. Control is always passed to the first line of the new process.

This operator should have a left argument or a right argument. If it has both the right argument is taken. The argument must a number. No result is returned.

Two separate cases exist depending on the argument.

Case 1) It resolves to zero or a positive integer.

If the left argument resoves to a positive integer then it is interpreted as a screen field process number. Control will be passed to the first line of the nominated screen field process. If the number is greater than the number of available screen fields the error "No such field" appears on the status line.

If the left argument is zero then control is passed to the screen field process indicated by system variable 448 (current field number).

Case 2) It resolves to a negative integer.

Only the negative integers "-1", '-2', '-3', and '-4' are allowed (else "No such process" error). Encryption of the negative numbers:

| Number | Meaning |
|---|---|
| -1 | GOTO BEGIN PROCESS |
| -2 | GOTO END PROCESS |
| -3 | GOTO SUPER BEGIN PROCESS |
| -4 | GOTO SUPER END PROCESS |

Thus all processes can be accessed by the GOTO operator.

System variables 448 (current screen field) will be modified whenever the argument is a positive number.

System variable 507 (current process type) may be changed by this operator to reflect the new process type.

| Value in 507 | Meaning |
|---|---|
| -1 | BEGIN PROCESS |
| -2 | END PROCESS |
| -3 | SUPER BEGIN PROCESS |
| -4 | SUPER END PROCESS |
| -5 | a process related to a screen field |

Examples:

```
 3  =>>          ;transfer control to first line of
                 ;screen field process 3.
                 ; afterwards:
                 ;   #448 will be 3
                 ;   #507 will be -5

GOTO  3          ;same effect as  first example


'-1' =>>         ;transfer control to first line of
                 ;BEGIN process for this schematic
                 ; afterwards:
                 ;   #507 will be -1


GOTO  '-4'       ;transfer control to first line of
                 .SUPER END process for this mode
                 ; afterwards:
                 ;   #507 will be -4
```

N.B. In the case of '-3' =>> (GOTO the SUPER BEGIN PROCESS) The current field (448) is set to 1, and previous field (s.v. 510) is set to 0, and type of exit (s.v. 453) is set to 255.

```
Operator Name:          HELP                  ! for GT see EQ
                        ----                  !
                                              --------------------
Class:          Special

Arguments:      L*

Result:         Yes

Summary:        num  HELP       ->  num
```

Description:
------------

This operator can "drive" the help structure which has been set up by a previous SHELP operator.

This operator requires a left argument which must be a number. If the number is a non-integer then it is rounded to an integer. This operator returns a result.

The left argument should be a number in the range 0 to 7. If the left argument is zero then the schematic nominated by the most recent SHELP operator (i.e. its left argument) is brought up on the screen and the help structure is entered. If the left argument is 1 to 7 then the corresponding schematic in the right argument list of the most recent SHELP operator is brought up on the screen and the help structure is entered.

The result is the number of the last help schematic on the screen before the help structure was terminated. This number refers to the position of that schematic in the right argument list of the most recent SHELP operator. If the last selected help schematic did not exist then its number is negated and returned as the result.

The help structure is explained in more detail in the SHELP operator.

Examples:

```
        3   SHELP   <21,22,23,24,'SORT_EXP','ERRORS','BYE','HELP'>
        #501    ->  'DOCTORS'
```

This usage will define seven schematics for the help structure. System schematics 21, 22, 23, 24 are nominated as help schematics 1, 2, 3, and 4 respectively. User schematics 'SORT_EXP', 'ERRORS', and 'BYE' in the schematic group 'HELP' in the schematic file 'DOCTORS' are nominated as help schematics 5, 6, and 7 respectively.

```
        0   HELP    ->  1
```

The help structure is entered when this operator is executed. The schematic selected by the previous SHELP operator, system schematic 23, will be placed on the screen in the furthest corner from the cursor. After this, pressing the numbers 1 to 7 would bring up the corresponding schematics nominated in the right argument of the SHELP operator.

For example, pressing 7 would attempt to bring up a schematic called 'BYE' in the group 'HELP' from the schematic file 'DOCTORS'.

The result of this operator indicates that system schematic 21 was the last help schematic on the screen before the help structure was terminated.

Once the help structure has been entered then pressing space or F8 will terminate it and then return to the "pre-help structure" state.

```
        1   HELP         ->       -7
```

This would bring up system schematic 21 and enter the help structure. The result would tend to indicate that 7 was pressed while in the help structure after which the corresponding schematic ('BYE' in group 'HELP' in schematic file 'DOCTORS') was not found.

Operator Name: IF

Other operators described here: THEN ELSE ENDIF

Class: Special

Arguments: L* for THEN (none for others)

Results: None

Summary:

```
IF  cc   THEN          ; 'cc' can be an expression
                       ;  which results in a condition
                       ;  code
         ......        ; VPL expressions
ELSE
         ......        ; VPL expressions
ENDIF
```

Description:

These operators allow the conditional execution of VPL code within a VPL process.

Of the four operators (IF THEN ELSE and ENDIF) only the THEN operator requires an argument. The THEN operator requires a left argument and it must be a number (a condition code is a number). None of these operators will attempt to pick up a right argument (i.e. they will not scan to the right of the operator). None of these operators return a result.

The IF operator is "cosmetic" and is not required. In any case the use of the IF operator is recommended for readability and it should appear in the same expression (and therefore the same line) as the THEN operator.

The ELSE operator is only required when the two possible conditions (true or false) need mutually exclusive paths through the VPL code. The ELSE operator should lie between a THEN operator and an ENDIF operator (not necessarily on the same line). The true condition will cease to execute when an ELSE operator is detected. Execution will recommence after the corresponding ENDIF operator is detected. The false condition will start execution when the corresponding ELSE operator is detected.

The decision is made on the basis of the left argument of the THEN operator. The argument is rounded to an integer if necessary. If the argument resolves to zero (0) then the test is considered to have failed (to be false); the following VPL code is not executed. The execution of VPL will recommence:

1) when a corresponding ELSE is detected

otherwise 2) when the corresponding ENDIF is detected

otherwise 3) the rest of the current process is skipped and execution recommences at the beginning of the next process.

If the argument resolves to anything else but zero (after rounding to an integer) then VPL code execution continues after the THEN operator. In this case the condition is said to be true.

IF, THEN, ELSE, ENDIF stuctures can be nested to any level. Because of the "nested" ability of these stuctures the term "corresponding" is used in the above description to qualify ELSE and ENDIF so only those at the same level of nesting will be recognized.

Care should be taken with the use of the BRANCH operator together with IF, THEN ELSE, ENDIF, structures especially jumping into such structures.

Examples:

```
IF  #3  GT  0   THEN      ; if the contents of field 3 is greater
                          ; than zero then...
    #3  SQR =   #3        ; get the square root of it and put
                          ; result back in field 3
ENDIF                     ; end of structure


IF  #901  EMPTY  THEN     ; if status line field 901 is empty
                          ; then....
    '***' = #901          ;   place '***' in status field 901
    BELL                  ;   ring bell
ELSE                      ; now if #901 is not empty then...
    #901 JOIN #1 = #1     ;   get #901 and join it to #1 and
                          ;   put the result in #1
ENDIF                     ; end of structure
902  INPUT                ; whether or not #901 was empty get
                          ; input from status line field 902
```

Example of nested "IF"s:

```
IF  DATETIME  PICK <5,2> NE 2 THEN
    "It's not February" =  #1
ELSE
    "It's February" = #1
    IF  DATETIME  PICK <7,2> EQ 29 THEN
        "A *** day to have a birthday!" = #2
    ENDIF
ENDIF
```

Operator Name: . INPUT

Symbolic representation: ?

Class: Special

Arguments: L R1 R2 (R3 see extension)

Result: Yes

Summary: fld INPUT <pos,len> -> str

N.B. This operator has several extensions

Description:

This operator will accept keyboard input.

In the simplest case (with no arguments and "A" as field verification) the cursor will be placed in the first position (i.e. left hand side) of the current field. The system will wait for user input. Printable characters will be entered into the field until it is full. Various control keys (e.g. CR, down arrow, tab) will cause the system to continue VPL execution with the final contents of the current field being returned as the result of this operator.

This operator can have a left argument and up to two right arguments (in a right argument list). All arguments given to this operator must be numbers. This operator returns a result.

The left argument represents the field number. If given this should refer to a screen field or a status line field which currently exists on the screen. If not the error "** VPL ** Field number (or system variable) out of range" appears on the status line. The number zero refers to the current screen field and is assumed if no left argument is given.

The first right argument is the position within the field where the cursor is to be placed. The position at the left hand end is taken to be 1, the next is 2, etc. The position at the right hand end is taken to be -1, the one before that is -2, etc. The position 0 is treated as the left hand side of the field. The default (when position is not given) is the left hand side of the field.

The second right argument is the length of the input field. If this is not given or given as 0 (zero) then the field length of the field in question is assumed. If the length is given as positive then a "sub-field" including the given position and those to the right of it is used. If the length is given as negative then a "sub-field" including the given position and those to the left of it is used. By giving lengths long enough (positive and negative) it is possible to have the so-called "sub-field" partly outside the original field. If a "sub-field" is specified then the contents of that "sub-field" form the result of this operator.

Some keys will have special actions in the INPUT operator. VISTA's theoretical keyboard can be functionally divided up as follows:

a) Normal printable ASCII characters (ASCII codes 32-126)
b) Extended printable ASCII characters (chunky graphics) (ASCII codes 128-254)
c) Left arrow, right arrow
d) Up arrow, down arrow
e) Carriage return (return, CR)
f) Eight function keys F1 to F8
g) character insert, character delete, character erase(RUB)
h) Line insert, line delete, line erase
i) TAB

Classes a) and b) are similar to the INPUT operator. While the cursor is not at the right hand extremity of the field (or "sub-field") then these printable characters are echoed (type and verification permitting) in the field and the cursor moves one position to the right. At the right hand extremity one of two things happen:

-if the verification is "E" or "F" (upper or lower case) then the newly input character will be put in the last position in the field after the contents of the field is pushed one position to the left.
-if the verification is other than "E" or "F" (upper or lower case) then after the rightmost position of a field is filled then input terminates.

Class c) keys will cause input to continue until an attempt is made to move the cursor outside the field (or "sub-field").

Class d) keys will terminate input.

Class e) keys will terminate input.

Class f) keys will terminate input and place a value of 1 to 8 in system variable 453 corresponding to the function key pressed (F1 to F8).

Class g) keys will have their described action and input will continue.

Class h) keys will terminate input without altering the field's contents.

Class i) keys will terminate input.

In all cases a code for the key that caused input to terminate is recorded in system variable 509.

Examples:

Assume that the current field is field 3 and it has 5 positions which are currently blank. Assume the verification is variety "A".

```
            INPUT                        ->  'test '
```

This expression will put the cursor in the first position of the current field (field 3) and await input. One possible interpretation of what happened during the execution of this expression was that the user typed "test" followed by a CR (carriage return).

```
        4   INPUT   =   #5
```

Get input from field 4 and when that is done place the contents of field 4 (returned by the INPUT operator) into field 5.

```
        4   INPUT   6
```

Get input from field 4. Before input is accepted the cursor should be placed in position 6 of field 4.

```
        901 INPUT   '-1'
```

Get input from field 901 (a status line field). Before input is accepted the cursor is placed in the last position of the field.

```
        35  INPUT   <,4>  = #301
```

Using screen field 35 as a base then define a "sub-field" for the purposes of this INPUT operator. The "sub-field" starts at position 1 of screen field 35 and is 4 positions long (regardless of the defined length of screen field 35. The contents of the "sub-field" is returned as the result and put in hidden field 301.

```
        21  INPUT   <'-3','-8'>
```

Using screen field 21 as a base then define a "sub-field" for the purposes of this INPUT operator. The "sub-field" starts at the third last position of screen field 21 and extends to the left of that point. The "sub-field" will be 8 characters long.

Extension:
----------

Syntax: fld INPUT <pos,num,type> -> str

A third right argument is also allowed. If it is given it must be a number.

The third right argument is the type of input to be obtained from the field. If the type is zero or not given then field verification given when the field in question was defined is taken. Below is a list of the field verifications currently allowed:

| Symbol | ASCII code | Meaning (characters that are accepted) |
|---|---|---|
| " " | 32 | everthing |
| A | 65 | everything |
| B | 66 | alphabetic, digits, space |
| D | 68 | space, digits, slash, comma, plus |
| E | 69 | everthing (push left at right end of field) |
| F | 70 | as "N" (push left at right end of field) |
| M | 77 | space, digits, comma, plus, minus |
| N | 78 | space, digits, comma, plus |
| P | 80 | space, digits, comma, plus, point |
| Q | 81 | space, digits, comma, plus, point, minus |
| Z | 90 | space, alphabetic |

If the type is positive it should be one of the above "ASCII codes" and will mean that the original given field verification will be overridden for this input as indicated by the above table.

If the type is -32 or less (e.g. -32, -33, etc) then keyboard input will be acknowledged by the character indicated by the absolute value of this number (e.g. -63 will mean that all characters input are to be echoed with "?").

If the type is given as -1, -2, -3, or -4 then a special action is being requested:

type = -1 (no wait case)
The keyboard and its type-ahead buffer will be checked for waiting characters. Those characters relevant to the current field are echoed and control returns immediately to the VPL interpreter.

type = -2 (no echo case)
INPUT will proceed as usual but printable characters will be echoed on the screen as spaces. The result returned will contain the actual keys pressed.

type = -3 (no echo, no wait case)
The keyboard and its type-ahead buffer will be checked for waiting characters. Those characters relevant to the current field are taken and those characters which are printable are echoed as spaces. Control returns immediately to the VPL interpreter. The result returned will contain the actual keys pressed.

type = -4 (no echo, no wait, clear type-ahead buffer case) The keyboard and its type-ahead buffer will be checked for waiting characters. Those characters relevant to the current field are taken. Any further characters in the type ahead buffer are cleared. Those characters which were taken and are printable are echoed as spaces. Control returns immediately to the VPL interpreter. The result returned will contain the actual keys pressed.

The "no wait" cases (types -1, -3, and -4) will accept characters if they are waiting. The fact that something has been added in a field can be detected by writing an impossible value in system variable 509 (e.g. zero) and seeing if it is still there after the "no wait" INPUT operators. System variable 509 encodes a value for the last key pressed.

Examples:

```
        12  INPUT   <,,65>
```

Accept input into screen field 12. Regardless of the verification associated with this field by the SKJEMA program (or for that matter by system variable 537), take any character pressed as valid input. N.B. 65 -> "A" which means accept everything.

```
        12  INPUT   <2,5,65>     =    #301
```

Accept input from the "sub-field" based on screen field 12. The "sub-field" starts at position 2 and is 5 characters long. Regardless of the verification associated with screen field 12 by SKJEMA (or s.v. 537), take any character pressed as valid input. The contents of the "sub-field" is returned as the result and placed in hidden field 301.

```
        12  INPUT   <2,5,'-63'> =   #301
```

Similar to above example in terms of "sub-field" but now the original verification (or that in s.v. 537) is taken. Any printable character which is accepeted will be echoed by "?". The result of the INPUT operator will be the contents of the "sub-field" with the actual keys pressed (i.e. not full of "?"s).

```
        12  INPUT   <2,5,'-2'> =   #301
```

Same as above example but echo is a space.

The following little program will loop until one character is pressed unless characters are already waiting in the buffer.

```
        0  =  #509                          ;to be able to see if INPUT
                                            ;gets anything
:L1:    12  INPUT   <,,'-1'>                ;no wait, no echo
        ...                                 ;perhaps check the time
        ...
        IF #509 EQ 0 THEN                   ;if s.v. still zero then
            BRANCH :L1:                     ;nothing pressed so loop
        ENDIF
                                            ;now field 12 can be read
```

N.B. The cursor will be placed where it is directed (position 1 in this case). If this operator is being used in a loop then system variable 508 may be very useful as the position of re-entry.

```
Operator Name:            JOIN              !
                          ----              !
                                            ----------------------
```

Class: String

Arguments: L* R*

Result: Yes

Summary: str JOIN str -> str

Description:
------------

The left argument and the right argument are concatenated to form the result.

The left and right arguments must exist. This operator produces a result.

When screen fields or status line fields are picked up then trailing spaces are not included.

The result should not exceed 255 characters or an error will occur.

Examples:

```
        'abcdef'     JOIN    '1234'      ->   'abcdef1234'

        ' TEST  '    JOIN    'ING '      ->   ' TEST  ING '


                         ==================
    Screen field 3:    ! 1234             !
                         ==================

                         ==================
    Screen field 4:    !  5678            !
                         ==================

            #3  JOIN    #4       ->        ' 1234  5678'
```

Operator Name: LAND

Other operators described here: LOR LNOT LXOR

Class: Database

Arguments: L R1 R2

Result: Yes

Summary: ln1 LAND <ln2,ln3> -> num

Description:

These operators form a resultant occurrence list from two input occurrence lists according to some given rule.

Each operator can have a left argument. If so, it must be a number. Each operator can have two right arguments. Any right arguments given must be numbers. Non-integers are rounded to integers if necessary. Each operator returns a result.

If given, the left argument should be a number in the range 1 to 101. These numbers refer to occurrence lists. Occurrence list 101 is called the current occurrence list and is assumed if the left argument is not given. The left argument is one of the input occurrence lists.

If given, the first right argument should be a number in the range 1 to 101. These numbers refer to occurrence lists. Occurrence list 101 is called the current occurrence list and is assumed if the first right argument is not given. The first right argument is the other input occurrence list.

If given, the second right argument should be a number in the range 1 to 101. These numbers refer to occurrence lists. Occurrence list 101 is called the current occurrence list and is assumed if the second right argument is not given. The second right argument indicates the number by which the resultant occurrence list will be accessed. Any occurrence list previously associated with this number is replaced.

The four operators described here are LAND, LOR, LXOR, and LNOT. They all combine two occurrence lists to generate a new (resultant) occurrence list. Except in the case of LNOT, the position of the input occurrence lists (left or first right) is irrelevant.

The result of this operator is the number of documents in the resultant occurrence list. If no database is open when this operator is used then zero will be returned as the result. If the result is non-zero then the document pointer points to the first document in the resultant occurrence list.

Occurrence lists which result from the SORT operator cannot be used as input lists to any of these four operators.

The action of the four operators is listed in table form below:

| Operator | Resultant occurrence list contains |
|---|---|
| LAND | documents which are found in both lists; i.e both the first list AND the second list. |
| LOR | documents which are found in either list; i.e. in the first list OR the second list. N.B. Documents found in both lists will only appear once in the resultant list. |
| LXOR | documents which are in the first list OR the second list BUT not in BOTH lists. This operation is sometimes called an exclusive OR. |
| LNOT | documents which are found in the first list but NOT in the second list. N.B. This operation is not reflexive, i.e. the order of the input lists is significant. |

Both the LAND and LOR operators can be used for making a separate copy of an occurrence list. In this case both input lists should be the same. This may be useful if the same occurrence list is to be independently accessed. It may also be useful to keep a copy of an occurrence list before it is sorted.

The LOR operator can be used to accumulate an occurrence list. In this context it could be used with the LDOC operator.

The LXOR operator can be used to clear an occurrence list. In this case all three argument should be the same list number. A more efficient way is normally to search for something which is not there.

Examples:

```
'SMITH'     SEARCH  <'NAMREG:NAME',1>    ->  43
'WORKING'   SEARCH  <'NAMREG:TOWN',2>    ->  525
```

In this case occurrence list 1 would contain the 43 documents in the 'NAMREG' register whose 'NAME' was 'SMITH'. Occurrence list 2 would contain the 525 documents in the 'NAMREG' register whose 'TOWN' was 'WORKING'.

```
1   LAND    <2,3>                        ->  5
```

So occurrence list 3 would contain 5 documents from register 'NAMREG' containing both the 'NAME' of 'SMITH' and the 'TOWN' of 'WORKING'. Interpreting the data a little, it would seem that in 'NAMREG' there are 5 people of the name 'SMITH' who live in 'WORKING'.

```
        1   LOR     <2,4>                      ->  563
```

So occurrence list 4 would contain 563 documents from register 'NAMREG' that contain the 'NAME' of 'SMITH' or the 'TOWN' of 'WORKING'. Interpreting the data a little, it would seem that in 'NAMREG' there are 563 people with the name 'SMITH' or who live in 'WORKING'.

```
        1   LXOR    <2,5>                      ->  558
```

So occurrence list 5 would contain 558 documents from register 'NAMREG' that contain the 'NAME' of 'SMITH' or the 'TOWN' of 'WORKING' but not both. Interpreting the data a little, it would seem that in 'NAMREG' there are 558 people with the name 'SMITH' or who live in 'WORKING' not including those who both are named 'SMITH' and live in 'WORKING'.

```
        1   LNOT    <2,6>                      ->  38
```

It follows from above that there are 38 people named 'SMITH' who do not live in 'WORKING'.

```
        2   LNOT    <1,7>                      ->  520
```

And there must be 520 people in 'WORKING' not called 'SMITH'.

Operator Name: **LAST**

Class: String

Arguments: L* R

Result: Yes

Summary: str LAST type -> pos

Description:

This operator will return the position of the last non-blank character in a string in the simplest case (type not given or =0).

This operator must be given a left argument. It can optionally have a right argument. If it has a right argument then it must be a number. This operator returns a result.

The left argument is checked to find the position (origin one) of the last non-blank character when type is not given or is given as zero. If the left argument is a null string or full of blanks then zero will be returned as the position of the last non-blank character. System variables 524 and 525 can be adjusted (default is space) so the term "last non-blank character" can be generalized to "last non-delimiter character".

If type is given as 1 then the position of the last character in the left argument is returned. If the string is null (length zero) then zero will be returned. This option (type=1) is unaffected by the setting of system variables 524 and 525.

If type is -1 then the position of the first non-blank character in the left argument is returned. If the left argument is null or full of blanks then the position of the character after the last is returned as the result. System variables 524 and 525 can be adjusted (default is space) so the term "first non-blank character" can be generalized to "first non-delimiter character".

Examples:

```
'hello'    LAST          ->  5   ;position of last non-blank
'hello  '  LAST          ->  5   ;character is invariant

'hello  '  LAST   1      ->  7   ;but the position of the last
                                 ;character may vary

'hello  '  LAST  '-1'    ->  1   ;the position of the first
                                 ;non-blank character

' hello'   LAST  '-1'    ->  2   ;position of first non-blank
                                 ;character
```

```
''            LAST          ->  0   ;returns zero for a null string

''            LAST   1      ->  0   ;returns zero for a null string

''            LAST   '-1'   ->  1   ;returns 1 for a null string
                                    ;i.e. one after last!


':'=#524                            ;set 1st delimiter to colon
':'=#525                            ;set 2nd delimiter to colon
' hello'     LAST   '-1'    ->  1   ;first "non-colon"
'::hello'    LAST   '-1'    ->  3   ;first "non-colon"

' '=#524=#525                       ;reset 1st and 2nd delimiters
' :hello ;'  LAST           ->  9   ;last non-blank character
' :hello ;'  LAST   1       ->  9   ;last character
';'=#524=#525                       ;1st + 2nd delimiters to ";"
' :hello ;'  LAST           ->  8   ;last "non-semicolon" character

' '=#524                            ;1st delimiter to space
';'=#525                            ;2nd delimiter to semicolon
' :HELLO ;'  LAST           ->  7   ;last character position which
                                    ;is not a space or semicolon
' ;HELLO ;'  LAST   '-1'    ->  3   ;first character position which
                                    ;is not a space or semicolon
```

Operator Name: LCHANGE

Class: Database

Arguments: L R

Result: No

Summary: ln1 LCHANGE ln2

Description:

This operator will change the list number associated with a given occurrence list.

This operator can have a left argument. If so, it must be a number. This operator can have a right argument. If so, it must be a number. Non-integers are rounded to integers if necessary. This operator does not return a result.

If given, the left argument should be a number in the range 1 to 101. These numbers refer to occurrence lists. Occurrence list 101 is called the current occurrence list and is assumed if the left argument is not given. The left argument is the input occurrence list number.

If given, the right argument should be a number in the range 1 to 101. These numbers refer to occurrence lists. Occurrence list 101 is called the current occurrence list and is assumed if the right argument is not given. The right argument is the resultant occurrence list number.

For identification purposes occurrence lists have numbers. Up to 101 occurrence lists can be held concurrently by the system. This operator simply changes the identification number by which an occurrence list is accessed. If the number given for the resultant occurrence list had an occurrence list associated with it then it is replaced. The input occurrence list number will have no occurrence list associated with it when this operator has finished (N.B. there is NO swapping of lists). The document pointer of the "changed" list is not altered.

Examples:

```
        1    LCHANGE      2
```

The previous contents of occurrence list 2 is replaced. The occurrence list previously referred to as list 1 can now be referred to as list 2. List 1 now has no occurrence list associated with it.

```
             LCHANGE      2
```

Current list (list 101) "changed" to list 2.

Operator Name: LDOC

Class: Database

Arguments: L R

Result: No

Summary: ln1 LDOC ln2

Description:

This operator will make a new occurrence list containing one document (ln2) out of the current document in the given occurrence list (ln1).

This operator can have a left argument. If so, it must be a number. This operator can have a right argument. If so, it must be a number. Non-integers are rounded to integers if necessary. This operator does not return a result.

If given, the left argument should be a number in the range 1 to 101. These numbers refer to occurrence lists. Occurrence list 101 is called the current occurrence list and is assumed if the left argument is not given. The left argument is the input occurrence list number.

If given, the right argument should be a number in the range 1 to 101. These numbers refer to occurrence lists. Occurrence list 101 is called the current occurrence list and is assumed if the right argument is not given. The right argument is the resultant occurrence list number.

The action of this operator is to make a resultant occurrence list containing one document which is the current document in the input occurrence list. If the resultant list number previously had an occurrence list associated with it then it is replaced. If the input occurrence list was empty then the resultant occurrence will also be empty. If no database is open then this operator has no effect.

Examples:

```
        2   LDOC    3
```

The previous contents of list 3 are replaced. If list 2 contains any documents then the one addressed by the document pointer will be made the only document in occurrence list 3. If list 2 is empty then list 3 will be empty. List 2 is not altered.

```
        2   LLENG          ->  525    ;list 2 contains 525 documents
        2   LPOS           ->  432    ;current document is 432nd
        2   LDOC
```

The previous contents of list 101 are replaced. After the LDOC operator list 101 will contain 1 document which will be the 432nd document of list 2. List 2 is not altered.

```
Operator Name:            LLENG                  ! for LE see EQ
                          -----                  !
                                                 --------------------
Class:          Database

Arguments:      L

Result:         Yes

Summary:        ln   LLENG      ->  num
```

Description:

This operator will return the number of documents in the given occurrence list.

This operator can have a left argument. If so, it must be a number. Non-integers are rounded to integers if necessary. This operator returns a result.

If given, the left argument should be a number in the range 1 to 101. These numbers refer to occurrence lists. Occurrence list 101 is called the current occurrence list and is assumed if the left argument is not given.

The action of this operator is to return the number of documents in the given occurrence list. If the given list is empty or no database is currently open then zero is returned.

Examples:

```
        'CRAMPON'   SEARCH  'FICHE:NOM'     ->  5
```

So now there are 5 documents in the current occurrence list.

```
            LLENG                           ->  5
        101 LCHANGE     13
        101 LLENG                           ->  0
        13  LLENG                           ->  5
```

The LLENG operator is now used to illustrate the action of the LCHANGE operator.

```
Operator Name:          LOG                 !
                        ---                 !
                                            ---------------------
Class:          Arithmetic

Arguments:      L  R  (one or the other, right takes precedence)

Result:         Yes

Summary:        num  LOG              ->  num
                     LOG   num        ->  num
                     LOG  (num)       ->  num
              N.B. These are all equivalent
```

Description:
------------

This operator will yield the natural logarithm of its argument.

This operator requires an argument. It can be either a left argument or a right argument. If both a left argument and a right argument are given then the right argument is used. The argument must be a number. This operator returns a result.

The argument must be a positive number. Negative numbers or zero cause a "** VPL ** Attempt to divide by zero" error. The natural logarithm is a logarithm base "e". The number "e" is approximately 2.71828182845904 . The result is that number which "e" needs to be raised to in order to be equal to the given argument.

Examples:

```
1    LOG                  ->  0
     LOG  (1)             ->  0
10   LOG  (1)             ->  0

22026.4657948067 LOG      ->  10
2.71828182845904 LOG      ->  1
```

To get the quadratic root of a number this operator could be used together with the EXP operator.

```
16  LOG  /  4  EXP        ->  2
```

```
Operator Name:          .    LOOKPROC           ! for LNOT see LAND
                             --------           !
                                                ---------------------
Class:          Special

Arguments:      L*

Result:         Yes, error code

Summary:        str  LOOKPROC      ->  err
```

Description:

This operator will indicate whether a procedure exists or not.

This operator requires a left argument. This operator returns a result which is an error code.

The left argument should represent a procedure name. Procedure names can be up to 20 characters long. Procedure names must not contain embedded spaces and must not start with a digit.

The result of this operator is zero if a procedure of the name indicated by the left argument exists. If no procedure of that name is found then 22 is returned. Error code 22 is associated with the message "Procedure not found".

Examples:

```
        IF 'WEEKDAY' LOOKPROC EQ 0 THEN
            DATETIME WEEKDAY = #13
        ELSE
            'SOMEDAY'         = #13
        ENDIF
```

This piece of code will check if a procedure called 'WEEKDAY' exists and if so it will be called with a left argument which is the output of the 'DATETIME' operator. The result of the procedure will be placed in screen field 13. If a procedure of that name is not found then 'SOMEDAY' is placed in screen field 13.

```
        'XYZ'   LOOKPROC    MESSAGE ->  '** VPL ** Procedure not found'
        'MENU'  LOOKPROC    MESSAGE ->  ''
```

The latter example indicates that a procedure of the name 'MENU' exists.

```
Operator Name:              LPOS                ! for LOR see LAND
                            ----                !
                                                --------------------
Class:          Database

Arguments:      L

Result:         Yes

Summary:        ln   LPOS        ->  num
```

Description:
------------

This operator will return the position of the current document within the given list.

This operator can have a left argument. If so, it must be a number. Non-integers are rounded to integers if necessary. This operator returns a result.

If given, the left argument should be a number in the range 1 to 101. These numbers refer to occurrence lists. Occurrence list 101 is called the current occurrence list and is assumed if the left argument is not given.

The action of this operator is to return the position of the current document within the given occurrence list. If the given list is empty or no database is currently open then zero is returned.

When an occurrence list is generated the current document pointer is set to the first document in that list. The STEP operator can be used to move the current document pointer.

Examples:

```
        'CRAMPON'   SEARCH  'FICHE:NOM'      ->  5
```

So now there are 5 documents in the current occurrence list.

```
            LLENG                            ->  5
            LPOS                             ->  1
            STEP
            LPOS                             ->  2
        101 LCHANGE      13
        101 LLENG                            ->  0
        101 LPOS                             ->  0
        13  LLENG                            ->  5
        13  LPOS                             ->  2
```

After an occurrence list is generated the current document is the first. The STEP operator moves the current document pointer to the second document. Notice that the LCHANGE operator does not effect the current document pointer.

Operator Name: MESSAGE

! for LT see EQ<br>
! for LXOR see LAND

Class: Special

Arguments: L*

Result: Yes

```
Summary:        num     MESSAGE        ->    str
```

Description:

This operator will return a string containing the error message (or informative message) corresponding to the number given as the left argument.

This operator must have a left argument and it must be a number (non-integers are rounded to integers). A result is returned.

If the left argument is zero or less a null string is returned as the result. Positive integers which have a corresponding message defined in the system file (defined and modified by VISETUP) will return that message as the result. If a message has not been defined for a positive number then a string containing the message number surrounded by "*"s will be returned.

The advantage of using this method over explicitly defining a string between quotes is that MESSAGE will pick up a message in the currently defined language. The currently defined language can be changed, added to, or modified by the VISETUP program. The VISUP program can be used to select one of the defined languages.

For a list of messages currently available (in English) via this operator see appendix B.

Examples:

```
   1  MESSAGE    -> '**VPL** Unrecognizable statement'

  70  MESSAGE    -> "Hit 'space' to continue"

 301  MESSAGE    -> '*301*'    ;not yet defined

   0  MESSAGE    -> ''         ;null string

'-3'  MESSAGE    -> ''         ;null string
```

```
Operator Name:            MODE                    !
                          ----                    !
                                                  ---------------------
Class:         Special

Arguments:     L

Result:        Only if operator fails, then result is error code

Summary:       num     MODE         [ ->    err ]
                       MODE         [ ->    err ]
```

Description:
------------
The supplied user interface in VIPS is sub-divided into modes. Each mode performs an application oriented task such as document input, document search, edit, sort, report generation, etc. VIPS has 50 modes which are numbered 0 to 49. Each mode has two special processes associated with it. These processes are called the SUPER BEGIN and the SUPER END process. This operator will switch between the 50 modes.

If a left argument is given it must be a number (non-integers rounded) in the range 0 - 49. Since this operator causes an immediate control transfer if the requested mode exists then no result is returned. If an error occurs then the relevant error code is returned as the result.

If a left argument is not given then the value in system variable 430 is assumed. From the point of view of the following explanation if a left argument is given it can be thought to overwrite the previous contents of system variable 430.

The following transfers then take place:

```
      Current Mode Number    --> Previous Mode Number
      (in VPL:         #404 = #505)

      Next Mode Number       --> Current Mode Number
      (in VPL:         #430 = #404)

      Constant  0            --> Next Mode Number
      (in VPL:            0 = #430)

      Constant  1            --> First use of SUPER BEGIN
      (in VPL:            1 = #515)
```

When this is done then control is transferred to the first line of the SUPER BEGIN of the new mode.

Mode 8 is reserved for an exit from VIPS to the host operating system. All files opened during the session of VIPS which have not already been closed will be automatically closed by this usage.

If the SUPER BEGIN of a particular mode is empty and so is its SUPER END then according to the default flow of control between processes, this would represent a loop. This condition is detected (both SUPER BEGIN and SUPER END being empty) and after one loop a 0 MODE operator will be forced. If the current mode was 0 then a 8 MODE operator will be forced (i.e. return to operating system).

Examples:

```
        MODE            ; transfer passed to SUPER BEGIN
                        ; of mode number in #430
                        ; #404 = #505
                        ; #430 = #404
                        ; #0   = #430
                        ; #1   = #515

   5    MODE            ; transfer passed to SUPER BEGIN
                        ; of mode 5
                        ;  5   = #430
                        ; #404 = #505
                        ; #430 = # 404
                        ;  0   = #430
                        ;  1   = #515

   8    MODE            ;exit to host operating system
                        ;close all currently open files

   0    MODE            ;useful for returning control to the
                        ;document level handling system in
                        ;VIPS. Mode 0 is the status line prompt
                        ;showing current schematic, list
                        ;length, time, and asking for next mode
```

Operator Name: NUMERIC

!for NE see EQ
!for NEMPTY see EMPTY
!for NNUMERIC see <-

Other operators described here: NNUMERIC

Class: Arithmetic conditional

Arguments: L R (at least one, right takes precedence)

Result: Yes, condition code

Summary:

```
str     NUMERIC      ->    cc
str     NNUMERIC     ->    cc
```

Description:

The NUMERIC and the NNUMERIC (read not numeric) operators return condition codes depending on whether their arguments are in a suitable form to be interpreted as numbers by the system.

Both operators need either a left argument or a right argument. If they have both then the right argument is taken. For readability it is recommended that only the left argument is used.

The NUMERIC operator returns the true condition code (i.e. 1 ) if its argument can be interpreted as a number. If its argument cannot be interpreted as a number then the false condition code is returned.

The NNUMERIC operator returns the false condition code if its argument can be interpreted as a number. If its argument cannot be interpreted as a number then the true condition code is returned.

What is a number?
VISTA does not have strict data types. All data items throughout the system can be viewed as strings. So there is a subset of strings which the system can interpret as numbers. The rules for valid representations of numbers (numeric strings) are set out below:

A) The only valid characters in a numeric string are:
0 1 2 3 4 5 6 7 8 9 + - . , (space)
B) There must be no imbedded spaces within the numeric string
C) There must only be one number per numeric string
D) If + is used it must be before the first non-blank character
E) If - is used it must be before the first non-blank character
F) Neither + nor - are necessary but both cannot be used
G) Commas can appear anywhere in the numeric string except in the first non-blank position
H) The numeric string may contain one (no more) decimal point "."

Other things to note:
1) A null string or a string full of spaces will be interpreted as the valid number zero for numeric purposes.
2) The result of arithmetic operators is always numeric

3) Resulting condition codes can be considered as numeric (i.e. true => 1 , false => 0 )
4) Error codes are numeric

Other operators in VPL which expect a number as an argument will fail with the error message "** VPL ** Non-numeric argument to arithmetic operator" if a string is given which cannot be interpreted as a number. If there is any chance of this happening (e.g. via user input) then it is recommended that these operators (NUMERIC and NNUMERIC) be utilized to check. Even when numeric verification are being used on field input the user can still enter embedded spaces or two decimal points.

The VPL interpreter has quite a wide interpretation of what is a number within a string. When strings are stated explicitly in VPL code then they should be surrounded by quotes (or double quotes). As a convenience positive numbers can be written without quotes. This "convenience" has a narrower interpretation of what is a valid number. The number can only be made up of the digits 0 to 9 and decimal point "." .

Examples:

The following example shows various types of tests in conjunction with IF-THEN-ELSE-ENDIF structures and a DO-WHILE-ENDDO loop. The idea is to prompt the user in field three and then check that a valid number is given. After it is established that a valid number is given then it is checked to see if it is positive. If so the natural logarithm is taken of it and the result is put back in field three. If these conditions are not met then a message is placed on the status line and the bell is rung and field 3 is blanked. This latter action is used as the "loop variable". Until the user enters a positive number the loop will continue.

```
'Please enter a positive number, then press CR' SW
DO
    3   INPUT                           ;get user input
    IF  #3  NUMERIC  THEN               ;check field 3
        IF  #3  GT  0   THEN            ;if number then check if
                                        ;positive
            #3  LOG =  #3
        ELSE
            'Can only take logs of positive numbers!'  SW
            BELL
            ''=#3                       ;clear field three
        ENDIF
    ELSE
        'Please input a number!!'  SW   ;here if non-numeric
                                        ;in field three
        BELL BELL                       ;wake up user
        ''=#3
    ENDIF
WHILE (#3 EMPTY)
ENDDO
```

```
Operator Name:           OR                !
                         --                !
                                           ---------------------
Class:            Arithmetic conditional

Arguments:        L*   R*

Result:           Yes,    condition code

Summary:          cc      OR      cc     ->    cc
```

Description:
------------
This operator performs a logical OR operation between its arguments and produces the appropriate result. The truth table for OR is:

| LEFT | RIGHT | RESULT |
|---|---|---|
| false | false | false |
| false | true | true |
| true | false | true |
| true | true | true |

```
                  where:  false   <-> 0 [ -0.5 < x < 0.5]
                          true    <-> not false
```

NB ! Care should be taken if using the OR operator with arguments not resulting from conditional expressions, as unexpected results may occur. The OR is done by addition of its arguments.

IF-THEN-ELSE-ENDIF and DO-WHILE-ENDDO structures can both be controlled by condition codes. Sometimes only one of several conditions is required to be true for some action to be taken. This operator can be placed between two other conditions so that the net result is true when either component conditions is true.

Examples:

```
        1    OR      1    ->  1           ; from above table
        0    OR      1    ->  1           ; from above table

  NB !  5    OR    '-5'   ->  0           ; unexpected result !

        if   #3 eq 33 OR (#201 lt 0)  then ....
                                          ; if field 3 is equal to
                                          ; 33 OR hidden field 201
                                          ; is less than 0 then....

        do  while ( #901 empty OR (#1 numeric) OR (#2 gt 0) )
            ...                           ; many ORs can be used
        enddo
```

```
Operator Name:              PICK                    !
                            ----                    !
                                                    ---------------------
Class:          String

Arguments:      L*  R1  R2

Result:         Yes

Summary:        str     PICK    <pos,num>    ->   str
```

Description:
------------

This operator will pick the indicated number of characters from the indicated position of the given string.

The left argument must be given. If either (or both) of the right arguments are given then they must be numbers (non-integers are rounded to integers). This operator returns a result.

The first right argument is the position. The position is origin one (i.e. 1 indicates the first position). If the position is a negative number it indexes the string from the right hand end. Thus a position of '-1' indicates the last position. If the position is given as zero it is treated as the first position. If the position is not given then the first position is assumed.

The number of characters required is the second right argument. If the number is not given then '1' is assumed. If the number is positive then the indicated number starting with the indicated position is taken. If the number is negative then the indicated number (absolute value) ending with the indicated position is taken. If the number is zero a null string is returned. The maximum string length is 255 characters.

Strings larger than the original string can be selected. The resulting string will always contain the requested number of characters. The left argument can be envisaged as having spaces joined to each end of it in order to meet criteria.

Examples:

```
'testx'  PICK                 -> 't'
'testx'  PICK  2              -> 'e'
'testx'  PICK '-1'            -> 'x'

'testx'  PICK <, 2 >          -> 'te'
'testx'  PICK < 3 , 2 >       -> 'st'
'testx'  PICK < 3 ,'-2'>      -> 'es'
'testx'  PICK <'-2','-6'>     -> '  test'
```

Operator Name: PICKW

Class: String

Arguments: L* R1 R2 R3

Result: Yes

Summary: str PICKW <pos,num,type> -> str

Description:

This operator will pick words out of a string.

The left argument must be given. If any (or all) of the right arguments are given then they must be numbers (non-integers are rounded to integers). This operator returns a result.

The first right argument is the word position. The word position is origin one (i.e. 1 indicates the first word). If the word position is a negative number it indexes the string from the right hand end. Thus a word position of '-1' indicates the last word. If the word position is given as zero it is treated as the first position. If the word position is not given then the first word is assumed.

The number of words required is the second right argument. If the number of words is not given then '1' is assumed. If the number is positive then the indicated number of words starting with the indicated word position is taken. If the number is negative then the indicated number of words (absolute value) ending with the indicated word position is taken. If the number of words is zero a null string is returned.

Two types of word identification are available. These are:

a) when type is not given or type=0
-not all spaces (delimiters) are considered significant. Leading, trailing and repeated imbedded spaces (delimiters) are ignored for the purpose of calculating word position. If more than one word is requested and available then the result will contain the words separated by a single space (delimiter). If multiple delimiters separate words then the first one is returned as a word separator in the result.

b) when type=1
-all spaces (delimiters) are considered significant for the purpose of calculating word position. Only one word will be returned (regardless of the number of words indicated by the second right argument). If the position indicates a word lying between to spaces (delimiters) then a null string is returned.

The resulting string will only contain the requested number of words if the left argument contains that many words from the indicated position.

Words are normally delimited by spaces. Other characters can be used as delimiters by writing to system variables 524 and 525.

Examples:

```
'this is a   test '  PICKW                ->  'this'
'this is a   test '  PICKW  4             ->  'test'
'this is a   test '  PICKW  <'-2',2>      ->  'a test'
'this is a   test '  PICKW  <'-2',3>      ->  'a test'
'this is a   test '  PICKW  <,9>          ->  'this is a test'

';'=#524                ;set 1st string delimiter to ";"
' '=#525                ;set 2nd string delimiter to space
'this-is a;  test '  PICKW                ->  'this-is'
'this-is a;  test '  PICKW  <2,2>         ->  'a;test'
' '=#524                ;reset 1st string delimiter to space

';'=#524=#525           ;set both string delimiters to ";"
';this;is;a;;test'   PICKW  <,,1>         ->  ''     ;null string
';this;is;a;;test'   PICKW  <2,,1>        ->  'this'
';this;is;a;;test'   PICKW  <4,,1>        ->  'a'
';this;is;a;;test'   PICKW  <5,,1>        ->  ''
';this;is;a;;test'   PICKW  <6,,1>        ->  'test'
';this;is;a;;test'   PICKW  <'-3',,1>     ->  'a'
```

```
Operator Name:              PLACE            !
                            -----            !
                                             ---------------------
Class:          String

Arguments:      L*  R1*  R2  R3

Result:         Yes

Summary:        istr    PLACE   <ostr,pos,num>  ->  str

N.B.            This operator has several extensions
```

Description:
------------

This operator can be viewed as a sophisticated version of assignment (i.e. "=" ). Where assignment obliterates the previous contents of a field this operator can be used to overwrite the contents of a field.

This operator must be given a left argument. It can have up to three right arguments. The first right argument must be given. If given the second and third right arguments must be numbers (non-integers will be rounded to integers). This operator returns a result.

The left argument will overwrite the first right argument to produce the result. The first right argument itself is not modified by this operator. The left argument is referred to as istr below. The first right argument is referred to as ostr below.

The second right argument is the position. If a positive position is given then overwriting commences from the nominated position (origin one) in ostr. If a negative position is given it is assumed to be from the right hand end of ostr (e.g. -1 => last). Positions larger than the number of characters in ostr will cause it to be extended with spaces. Large negative positions will assume the start of the ostr. If the second right argument is not given then the first position of ostr is assumed.

The third right argument is the number of characters to be taken from istr. Positive numbers will take from the start of istr while negative numbers will take from the rear of istr. If the number is zero or istr is a null string then the result is ostr. When the number exceeds the number of characters in istr then the appropriate number of spaces are added to its end (or its start if the number is negative). If the third right argument is not given then all characters in istr are taken.

Examples:

```
    'testx'  PLACE  '1234567890'                -> 'testx67890'
    'testx'  PLACE  <'1234567890',3>            -> '12testx890'
    'testx'  PLACE  <'1234567890',9,6>          -> '12345678testx '
    'testx'  PLACE  <'1234567890','-3','-2'>    -> '1234567tx0'
    'end'    PLACE  <'the',8>                   -> 'the    end'
```

N.B. The third example has a trailing space because 6 characters where requested from 'testx' which only has 5.

Extension:
----------
This operator (PLACE) can have up to five right arguments. The summary then looks like:

```
        istr    PLACE   <ostr,pos,num,type,decimals>    ->  str
```

If the fourth and fifth right arguments are given they must be numbers. Non-integers will be rounded to integers.

The fourth right argument is the type. The default type is overwrite (explained above). The type can be given explicitly as 0 to get overwrite. If type is 1 then the indicated position and all those to its right in ostr are moved right to accomodate istr. The result is as large as necessary.
If type is -1 then the indicated position and all those to its left in ostr are moved left to accomodate istr. The result will be the same length as ostr so characters "falling off" the left are ignored.

The fifth right argument is the number of decimals to be added to istr. If istr cannot be decoded as a number then this argument has no effect. If istr can be decoded as number then this number of decimals will be added to it before it is used to overwrite or insert. "Decimals" are digits to the right of the decimal point.

If the fifth right argument is NOT given then this operator will make it own decision how to treat istr. If istr is a string then it is used as is. If istr was the result of an arithmetic operation which resulted in an integer then it is used as is (without decimals). If istr was the result of an arithmetic operation which resulted in a non-integer then the number of decimals indicated by system variable 540 is used.

Examples:

```
    'testx' PLACE  <'1234567890',,,1>       -> 'testx1234567890'
    'testx' PLACE  <'1234567890',3,2,1>     -> '12te34567890'

    'testx' PLACE  <'1234567890',3,2,'-1'> -> '3te4567890'
    'h'     PLACE  <'    04AC','-1',,'-1'> -> '   04ACh'


    '33'    PLACE  <'',,,,3>                -> '33.000'
    29+4    PLACE  <'',,,,3>                -> '33.000'

    66/2    PLACE  <'********','-1',,'-1',2> -> '***33.00'
```

```
Operator Name:           PRCHAR               !
                         ------               !
                                              --------------------
```

| | |
|---|---|
| Class: | Printing and sequential file handling |
| Arguments: | L* R |
| Result: | Yes, error code |

```
Summary:        num    PRCHAR         ->  err
                num    PRCHAR  un     ->  err
                str    PRCHAR  un     ->  err
```

Description:
------------

This operator is designed to send control codes to the printer.

This operator must be given a left argument. If it is given a right argument then it must be a number (non-integers rounded to integers if necessary). This operator returns a result.

In the simplest case the left argument is a number in the range 0 to 255. This code will be output to the printer. Assuming the printer handles normal ASCII codes then 10 would be a linefeed while 13 would be a carriage return.

To save repeated usage of this operator it is possible to give a left argument which is a string. This string is a list of codes to be output to the printer. Each element in the list is separated by a comma. Each element in this list should be a number or a number followed by "R" (or "r") followed by a repeat count (e.g. 10R4 output four linefeeds).

The right argument is the unit number. If the right argument is not given then the value in system variable 536 is taken. The initialized value in # 536 is -1 which indicates the printer. To redirect output from the printer to a file it is necessary to open the file with the SOPEN operator and then either put that unit number in # 536 or give it as the right argument to this operator.

The result is an error code. If this operator is successful then zero is returned. If output is going to a printer it is not envisaged that an error report will be returned by the host operating system. If however the output is being redirected to a file then some error may be returned.

| Error code | Meaning |
|---|---|
| 55 | Open for read only |
| 58 | Unit number not in use (is the file open?) |
| 61 | Drive or device full |

If the host operating system allows it then it may be possible to redirect the printer output to another byte orientated device (e.g. communication channel, console, etc.).

Examples:

Assuming the printer in question uses normal ASCII control sequences.

```
'65'            PRCHAR          ->  0   ;print "A"
'97'            PRCHAR          ->  0   ;print "a"
'84,69,83,84'   PRCHAR          ->  0   ;print "TEST"
'13,10'         PRCHAR          ->  0   ;send CR-LF to printer
'13,10R4'       PRCHAR          ->  0   ;send CR followed by
                                        ;four LFs to printer
'12'            PRCHAR          ->  0   ;quite often formfeed


'VIPS.PRN'      SOPEN    10     ->  0   ;open a file called
                                        ;"VIPS.PRN", create it
                                        ;if necessary
10 = #536                               ;default unit for
                                        ;PRCHAR, PRSTR and
                                        ;PRINT
'65'            PRCHAR          ->  0   ;send "A" to file
'97'            PRCHAR          ->  0   ;send "a" to file
'84,69,83,84'   PRCHAR          ->  0   ;send "TEST" to file
'13,10'         PRCHAR          ->  0   ;send CR-LF to file
'13,10R4'       PRCHAR          ->  0   ;send CR followed by
                                        ;four LFs to file
'-1' = #536                             ;direct default output
                                        ;back to printer
'84,69,83,84'   PRCHAR          ->  0   ;send "TEST" to printer
'84,69,83,84'   PRCHAR   10     ->  0   ;send "TEST" to file
```

```
Operator Name:            PRINT          !
                          -----          !
                                         ---------------------
```

Class: Printing and sequential file handling

Arguments: R1 R2 R3 R4

Result: Yes, error code

Summary: PRINT <un,from,to,type> -> err

Description:

This operator will send the current contents of the screen (or part of it) to the printer.

This operator does not have a left argument. If one is accidently given then it is ignored. This operator can have up to 4 right arguments. Any that are given must be numbers (non-integers rounded to integers if necessary). This operator returns an error code as a result.

In the simplest case (no arguments) this operator will send the screen image (less the status line) to the printer. If the printer is capable of echoing every character on the screen then a true replica of the screen (less status line-usually the bottom line) will appear on the printer.

The second and third right arguments are "from" line number "to" line number respectively. If the "from" line is not given then the first (top) line is assumed. If the "to" line is not given then the last line of the schematic (not the status line which is usually underneath it) is assumed. If the status line is also required in the output then its line number must be stated explicitly in the "to" argument.

If the "to" line number (third right argument) is given as zero then trailing blank text lines are not output. In this case a blank data field will cause output to at least the line it is on. If the "to" line number is -1 then trailing blank lines are not output. In this latter case the number of lines output by this operator could vary depending on whether lower data fields were blank or not.

The fourth right argument is the type. If the type is not given or is zero then there is no expansion of special characters. The meaning of the other values of type are listed below:

| type | meaning |
|---|---|
| 1 | Expand special characters lying in text |
| 2 | Expand special characters lying in text and don't output a trailing new line (usually CR-LF) |
| -1 | Expand special characters lying in text and data fields |
| -2 | Expand special characters lying in text and data fields and don't output a trailing new line (usually CR-LF) |

The first right argument is the unit number. If the first right argument is not given then the value in system variable 536 is taken. The initialized value in #536 is -1 which indicates the printer. To redirect output from the printer to a file it is necessary to open the file with the SOPEN operator and then either put that unit number in #536 or give it as the right first argument to this operator.

The result is an error code. If this operator is successful then zero is returned. If output is going to a printer it is not envisaged that an error report will be returned by the host operating system. If however the output is being redirected to a file then some error may be returned.

| Error code | Meaning |
|---|---|
| 55 | Open for read only |
| 58 | Unit number not in use (is the file open?) |
| 61 | Drive or device full |

If the host operating system allows it then it may be possible to redirect the printer output to another byte orientated device (e.g. communication channel, console, etc.).

Examples:

```
PRINT                      ->  0   ;print the screen less  status
                                   ;line
PRINT   <,4,14>            ->  0   ;print from the 4th to the 14th
                                   ;line inclusive
```

If the status line is on line 24 then:

```
PRINT   <,24,24>           ->  0   ;print the status line

PRINT   <,,0>              ->  0   ;print the screen less status
                                   ;line and trailing blank text
                                   ;lines

PRINT   <,3,'-1'>          ->  0   ;print the screen from line 3
                                   ;and less status line and
                                   ;trailing blank lines

'VIPS.PRN'  SOPEN   10     ->  0   ;open a file called
                                   ;"VIPS.PRN", create it
                                   ;if necessary
10 = #536                          ;default unit for
                                   ;PRCHAR, PRSTR and PRINT
PRINT                      ->  0   ;send a screen image less
                                   ;status line to file
PRINT   <,1,24>            ->  0   ;send a screen image to file
PRINT   <'-1',1,24>        ->  0   ;send a screen image to printer
PRINT   <'33',1,24>        ->  58  ;not such unit

'-1' = #536                        ;restoreprinteras default unit
PRINT   <,1,24>            ->  0   ;send a screen image to printer
```

```
Operator Name:            PRSTR                   !
                          -----                   !
                                                  ---------------------
```

Class: Printing and sequential file handling

Arguments: L* R1 R2

Result: Yes, error code

Summary: str PRSTR <un,type> -> err

Description:

This operator will print a string.

This operator requires a left argument. It may have up to 2 right arguments which, if given, must be numbers (non-integers rounded to integers if necessary). This operator returns a result which is an error code.

The left argument will be sent to the printer. If the left argument is a null string then no characters will be sent to the printer (perhaps the type may cause some CR-LFs to be sent).

The second right argument is the type. If the type is not given or zero then nothing is appended to the string sent to the printer. If the type is a positive number then that number of CR-LFs are appended to the string sent to the printer. If the type is -1 then the left argument is treated as a field and encoded into CBASIC format.

The first right argument is the unit number. If the first right argument is not given then the value in system variable 536 is taken. The initialized value in #536 is -1 which indicates the printer. To redirect output from the printer to a file it is necessary to open the file with the SOPEN operator and then either put that unit number in #536 or give it as the right first argument to this operator.

The result is an error code. If this operator is successful then zero is returned. If output is going to a printer it is not envisaged that an error report will be returned by the host operating system. If however the output is being redirected to a file then some error may be returned.

| Error code | Meaning |
|---|---|
| 55 | Open for read only |
| 58 | Unit number not in use (is the file open?) |
| 61 | Drive or device full |

If the host operating system allows it then it may be possible to redirect the printer output to another byte oriented device (e.g. communication channel, console, etc.).

Examples:

```
    'this is a test' PRSTR            ->  0  ;send that string to
                                             ;the printer
    'this is a test' PRSTR   <,1>     ->  0  ;send that string
                                             ;followed by a CR-LF
                                             ;to the printer
    'this is a test' PRSTR   <,3>     ->  0  ;send that string
                                             ;followed by 3 CR-LFs
                                             ;to the printer


    'VIPS.PRN'  SOPEN   10            -> 0   ;open a file called
                                             ;"VIPS.PRN", create it
                                             ;if necessary
    10 = #536                                ;default unit for
                                             ;PRCHAR, PRSTR and PRINT
    'this is a test' PRSTR            -> 0   ;send string to file
    'this is a test' PRSTR  '-1'      -> 0   ;send string to printer
    'this is a test' PRSTR  <,1>      -> 0   ;send string to file
                                             ;followed by CR-LF
```

Examples of Datastar (CBASIC) format usage:

```
    'this is a test' PRSTR  <,'-1'> -> 0     ;send string to file as is
    'this is,a test' PRSTR  <,'-1'> -> 0     ;send string to file
                                             ;surrounded by double
                                             ;quotes (because of comma)
```

Operator Name: PUT

Class: Database

Arguments: L* R1* R2 R3

Result: Yes, error code

Summary: str PUT <fds,ln,type> -> err

N.B. This operator has an extension

Description:

This operator will place the given string into the field of current document of the given list. The required field is addressed by its field descriptor.

This operator requires a left argument. It can have three right arguments. The first right argument is compulsory. If the second and third arguments are given they must be numbers (non-integers are rounded to integers if necessary). This operator returns a result which is an error code. Zero indicates no error.

The left argument is the string to be stored in the database.

If the current document in the given occurrence list does not contain a field with the given descriptor then a new field is created containing the string with the indicated attributes.

If the current document in the given occurrence list does contain a field with the given descriptor then that field is suitably modified to contain the new string with the indicated attributes.

The database stores data as characters, while non-integers resulting from arithmetic may be held in an internal form (double precision real format). If the left argument is in such a form then it is converted into a string with the number of decimals specified by system variable 540 before it is stored in the database.

To save space the database does not store trailing spaces given in the left argument. It is possible to have leading spaces stored by placing 1 in system variable 539. The default is that leading spaces are not stored.

The first right argument is the field descriptor.

The format of the field descriptor is as follows:

```
reg:nam.ext

    where:
              reg  is register name (ignored by PUT)
              nam  is searchable part of name
              ext  is non-searchable part of name
```

The register name is not required by the PUT operator and will be ignored. It may be useful to have the register name present from the point of view of checking that the register name is the same as that which the referenced occurrence list was generated by (i.e. a CREATE or SEARCH operator). In the future the interpreter may check this.

The searchable part of the name must be given and be non-blank. The field name "0" (zero) is reserved for a field containing the register name of the document (put in there by the CREATE operator). Two methods of field naming are supported. The first method is by number in which the field name can contain up to three digits. The second method is by a string which can be up to 31 characters long and must not start with a digit (or contain ":", ".", or space).

The extension is optional and can be up to 3 alphanumeric characters long. If the field is defined with an extension (i.e. by this operator) then the same extension must be given to the GET operator which fetches it.

The second right argument is the list number. It should be in the range 1 to 101 where 101 represents the current list. When the list number is not given then the current occurrence list is assumed. This operator will modify the current document in the given occurrence list. If the given occurrence list is empty then this operator has no effect.

The third right argument is the type. This is for defining whether or not the field is key or non-key.

| type | attribute |
|---|---|
| positive | store field as a key field |
| 0 | store field as it was previously stored. If the field did not previously exist then store it as a non-key |
| negative | store field as a non-key field |

If type is not given then type=0 is assumed.

The result is an error code. If no error occurs then zero is returned. Some possible error codes are:

| error code | meaning |
|---|---|
| -997 | drive full |
| negative | low level error in database system |
| 0 | no error |
| 43 | trying to put data in a deleted document |
| 47 | database not open |

Examples:

```
        CREATE      'NAMREG'           ;create a new document in a
                                       ;register called 'NAMREG'
                                       ;The new document will be
                                       ;referenced via the current
                                       ;occurrence list
        'Peter' PUT <'namreg:fname'>          ->  0
```

This will put the string 'Peter' into the field called 'fname' (no extension) of the newly created document in the current occurrence list associated with the register 'NAMREG'. 'Peter' will be stored as a non-key. The result indicates the operator has been successful.

```
        'Smith' PUT <'namreg:surname',,1>   ->  0
```

This will put the string 'Smith' into the field called 'surname' (no extension) of the newly created document in the current occurrence list associated with the register 'NAMREG'. 'Smith' will be stored as a key. The result indicates the operator has been successful.

```
        '1.86m' PUT <'namreg:class.a',,1>   ->  0
```

This will put the string '1.86m' into the field called 'class.a' ("a" is extension) of the newly created document in the current occurrence list associated with the register 'NAMREG'. '1.86m' will be stored as a key. The result indicates the operator has been successful.

```
        '64 Kg' PUT <'namreg:class.b',,1>   ->  0
```

This will put the string '64 Kg' into the field called 'class.b' ("b" is extension) of the newly created document in the current occurrence list associated with the register 'NAMREG'. '64 Kg' will be stored as a key. The result indicates the operator has been successful.

Now it may be realized that the surname wasn't 'Smith' but 'Smithe'. This can be altered as follows:

```
        'Smithe'    PUT <'namreg:surname'>  ->  0
```

Note that 'Smithe' will also be stored as a key because the previous contents of the field 'surname' was a key (type defaults to 0 when not given).

Advanced example:

When storing fields on the screen the PUTDOC operator can be used to store a whole document at once. It may be instructive to look at the operation performed by PUTDOC in terms of the more primitive (but flexible) PUT operator.

```
;a new document is CREATEd or an old one is obtained (by SEARCH)
;(the current occurrence list is assumed)
;
        1=#201
        DO  WHILE (#201 LE #447)           ;s.v.447 - fields in schema
            ##201   PUT  <#201,,#201 FSTAT>
        ENDDO
;
```

This loops for each field on the screen. The field number is used as the field name (no extension). The key/non-key attribute for the field in question is obtained by the FSTAT operator which returns a positive number if the screen field was defined as a key and a negative number if the screen field was defined as a non-key.

Extension: Storing multiple keys

When a string is being stored in a key field then a transformed version of that string is stored in the database dictionary. This transformation involves removing all spaces and folding to upper case. This transformed version of the original string is sometimes referred to as a "key".

In the normal case one key is entered into the database dictionary for each string stored in a key field. It is possible to have the string stored as several keys by separating the component parts by semicolons. Semicolon is the default key delimiter and can be changed by writing to system variable 526.

N.B. Regardless of what happens in the database dictionary the untransformed string (less trailing spaces- and perhaps leading spaces also) is stored in the document.

N.B. If a string (or component string) is blank or null then no entry is made in the database dictionary associated with it.

Example:

```
        'tall;blue eyes'    PUT     <'namreg:class.c',,1>    ->  0
```

This would store 'tall;blue eyes' in the current document in the current occurrence list. The register name associated with the current occurrence list should be 'namreg' and the field it will be stored in is called 'class.c' (where "c" is the extension).
Since this string is to be stored in a key field and since it contains one semicolon separating two non-blank component strings then two keys are stored in the database dictionary. In their transformed state they would be 'TALL' and 'BLUEEYES'. The point of doing this is that the following 4 searches would find this document.

```
        'tall'              SEARCH      'namreg:class'        -> 1+
        'blue eyes'         SEARCH      'namreg:class'        -> 1+
        'tall;blue eyes'    SEARCH      'namreg:class'        -> 1+
        'blue eyes;tall'    SEARCH      'namreg:class'        -> 1+
Operator Name:          PUTDOC              !
                        ------              !
                                            --------------------
Class:         Database

Arguments:     L   R

Result:        Yes,   error code

Summary:       ln      PUTDOC  type    ->  err
```

Description:
------------

This operator will place the contents of the screen fields into the current document in the given list.

This operator may have a left argument. If so, it must be a number. This operator may have a right argument. If so, it must be a number. Non-integers are rounded to integers if necessary. This operator returns a result which is an error code.

The left argument is a list number. It should be in the range 1 to 101. Occurrence list 101 is referred to as the current occurrence list and is assumed if no left argument is given.

The right argument is the type. If given, it should either be 0 or 1. If the right argument is not given then a type of zero is assumed. If the type is zero then all screen fields are stored in the indicated document. If the type is 1 then only non-blank screen fields are stored in the indicated document.

The result of this operator is an error code. If the operation is successful then zero is returned. If there is no database open then error code 47 is returned. If the current document in the indicated list has been deleted then error code 43 is returned. If the given list contains no documents then error code 45 is returned.

The action of this operator is to get the screen fields from the current schematic and store them in the current document of the given occurrence list. The system notes whether each screen field was defined as a key or non-key and stores the contents of that screen field accordingly.

This operator can be used both for storing new documents and editing old ones. If it is used to store new documents it should follow a CREATE operator. In this case the setting of type would make no difference. If it is used to edit an old document this operator would normally follow a SEARCH operator. The significance of the type in this case is that setting it to 1 will leave fields in the old document corresponding to blank screen fields unaltered.

Currently the fields in a document are named. Each field in a document can have up to a 31 character field name and optionally a three letter extension. The field name in a document can start with either an alphabetic character or a numeric character (i.e. 0 to 9). Screen fields, however, are numbered in sequence by the SKJEMA program which is used to create schematics. In the future it will be possible to optionally associate a field name and an extension to a screen field. To distinguish the compulsory screen field number from the optional screen field name, the latter must not commence with a numeric character (i.e. 0 to 9).

In the future this operator will check if a screen field has a name (and optionally an extension) associated with it and if so this screen field name will become the name of the field in the document. If a screen field does not have a name associated with it then its field number will become the name of the field in the document.

Examples:

```
        CREATE
        PUTDOC
```

This sequence will create a new document in the current occurrence list with the register name the same as the name of the schematic on the screen (which is indicated by the contents of system variable 403). The PUTDOC operator will store the contents of the screen fields currently in the schematic in the newly created document.

```
        #1  SEARCH  1   =   #201
        IF #201 EQ 1 THEN
            PUTDOC 1
        ENDIF
```

This would search for documents with the same register name as the name of the current schematic which in field 1 had the same contents as screen field 1. If one such document is found then the non-blank screen fields are edited into that document.

The "advanced example" in the description of the PUT operator shows PUTDOC (type 0) defined in terms of more primitive operators. It may also be instructive to read the GETDOC operator description.

```
Operator Name:          REGISTER           !
                        --------           !
                                           --------------------
Class:          Database

Arguments:      L*

Result:         Yes

Summary:        num     REGISTER      ->    reg
```

Description:
------------

This operator returns the register names defined in the currently open database file.

This operator requires a left argument. It must be a number, non-integers will be rounded to integers if necessary. This operator returns a result.

Inside the database a table is kept of all the register names currently in use in the system. There must be one or more documents stored associated with a register name for that name to be considered "in use". This table is ordered alphabetically. The left argument of this operator should be a positive integer. The number 1 will return the first (in sorted sequence) register name in use. The number 2 will return the second, etc. When there are no more register names in use a null string is returned as the result. A null string has a length of zero.

If no database file is currently open then all left arguments will cause a null string to be returned by this operator.

Examples:

```
        1   REGISTER    ->  'ADDREG'
        2   REGISTER    ->  'NAMES'
        3   REGISTER    ->  'REPORT'
        4   REGISTER    ->  'ZQW'
        5   REGISTER    ->  ''          ;no more register names in
                                        ;use
        1   REGISTER    ->  'ADDREG'    ;as expected

            DBCLOSE                     ;close current database
        1   REGISTER    ->  ''          ;now null string
```

```
Operator Name:              RETURN              !
                            ------              !
                                                --------------------
```

Class: Special

Arguments:

Result: No

Summary: RETURN

N.B. This operator can only be used within a procedure

## Description:

This operator terminates VPL execution within a procedure and returns to whatever invoked the procedure.

This operator requires no arguments. This operator does not return a result.

This operator can appear anywhere in a procedure. If the VPL interpreter executes this operator then no further interpretation will be done inside the current procedure. Control will be passed back to whatever invoked the procedure. Procedure calls can be nested and if necessary can be recursive, i.e. a procedure may directly or indirectly call itself. A procedure does not have to have a RETURN operator on its last line but it is recommended. If a procedure does not have RETURN operator at its end then any attempt to fetch the line after the last will have the same effect as a RETURN operator.

The RETURN operator must only be used within a procedure. If it is used elsewhere a "** VPL ** (Internal error) Stack unexpectably empty" error message will appear on the status line.

Example:

```
;Assume this is a procedure to get the fourth root of a number
;
;       %1  QUADROOT    ->      %0
;
;If the left argument is negative or not numeric then zero will
;be returned as the result and a message put on the status line
;N.B. Also need to check if left argument is given.
;
    0   =   %0
    IF %1 EXIST  THEN       ;so far so good
    ELSE
        RETURN              ;else return with result 0
    ENDIF
;
    IF %1 NNUMERIC  THEN
        'QUADROOT needs a numeric left argument'  SW
        RETURN
    ENDIF
```

```
;
    IF %1 LT  0  THEN
        "Can't get QUADROOT of negative number"   SW
        RETURN
    ENDIF
;
    IF %1 EQ  0  THEN
        RETURN                  ;already have answer
    ENDIF
;
    %1 SQR SQR =  %0
    RETURN                      ;end of procedure
```

The second last line takes advantage of the left-to-right nature of VPL. Notice there is no hierachy between operators.
That expression could be written as follows:

```
    SQR(%1)    = #201           ;use short hidden field for temporary
    SQR(#201) = %0              ;    storage
```

There would be little difference execution speed.
Which approach is easiest to comprehend?

```
Operator Name:               SA                    !
                             --                    !
                                                   --------------------
Class:         Status line

Arguments:     L   R1  R2  R3  R4  R5  R6  R7  R8  R9

Result:        No

Summary:       af1      SA      <af2,af3,af4,af5,af6,af7,af8,af9,af10>
```

Description:

This operator defines the attributes of each status line field. This operator is passive. The status line fields are set to these attributes after the next SL operator.

This operator can have a left argument. If so, it must be a number. This operator can have up to 9 right arguments. Any right arguments that are defined must be numbers. This operator does not return a result.

All numbers given to this operator are rounded to integers. Only numbers between 0 and 63 inclusive are meaningful.

The left argument represents the attribute of the first status line field (addressed as 901). The first right argument represents the attribute of the second status line field (addressed as 902). The second right argument represents the attribute of the third status line field (addressed as 903), and so on. If an argument is not given then an attribute of zero is assumed. Up to 10 status line fields are allowed.

A suggested mapping of available attribute numbers to actual screen attributes (half/full intensity, reverse video, flashing, underline, colours, etc) is given in Appendix C.

Examples:

```
    #521       ->  80      ;thus 79 usable characters on status
                           ;line
    10  SP  30             ;either field 901 will have 10 chars
                           ;and field 902 will have 30 chars or
                           ;they will have the proportion 1:3

    'N' SV                 ;#901 will only accepts digits and
                           ;space while #902 will accept anything

    0   SA  7              ;#901 will have attribute 0 while #902
                           ;will have attribute 7

    2   SL                 ;now redefine the status line to have
                           ;two fields of length 10 and 30 chars.
                           ;respectively
```

```
Operator Name:          SCHDEF          !
                        ------          !
                                        --------------------
Class:          Special

Arguments:      L  R1  R2  R3  R4  R5  R6  R7  R8  R9

Result:         No

Summary:        lc1  SCHDEF  <lc2,lc3,ntl,nbl,aac1,abc1,ac2,ac3,atbl>
```

Description:
------------

This operator will dynamically make a new schematic on the screen.

This operator may have a left argument. If given it must be a number. This operator may have up to 9 right arguments. Any given arguments must be numbers. Non-integers are rounded to integers if necessary. Negative numbers should not be given as arguments to this operator. This operator returns a result.

The current schematic on the screen will be replaced by a schematic which is made up almost completely of fields (i.e. very few text positions). The only text positions will be after the "c3" column and will vary depending on how many "c1" columns can fit across one line. The number of data fields defined and their attributes in the "dynamic" schematic will depend on the arguments to this operator. The verification of the fields in the "dynamic" schematic is space (i.e. key and accept everything).

The status line is unaffected by this operator. The "dynamic" schematic will take all lines available to a schematic which will be the number in system variable 520 less one (for the status line). The "dynamic" schematic is made up of a given number of "top lines" and a given number of "bottom" lines. Each top and bottom line is one field. The remaining lines in the middle of the screen are divided into columns. Each column on each line is a field. Each line in the middle of the screen is made up of a left hand field ("c2") and a right hand field ("c3") and repeated "main" fields ("c1"). See the accompanying diagram.

The meaning of the arguments and their default values follows:

| Argument | Meaning | Default |
|---|---|---|
| left arg. | length of c1 (main columns) | 11 |
| 1st right | length of c2 (left hand side) | 3 |
| 2nd right | length of c3 (right hand side) | 0 |
| 3rd right | number of top lines | 1 |
| 4th right | number of bottom lines | 0 |
| 5th right | attibute a for c1 (odd numbers on each line) | 4 |
| 6th right | attibute b for c1 (even numbers on each line) | 5 |
| 7th right | attibute for c2 | 7 |
| 8th right | attibute for c3 | 7 |
| 9th right | attibute for top and bottom lines | 0 |

The following is an example of how a screen would be divided up by this operator. Note that the "c1" fields could also appear more or less than three time across a line.

```
!-----------------------------------------------------------!
!              top line(s)                                  !
!-----------------------------------------------------------!
! c2    !     c1 (1)   !    c1 (2)   !     c1 (3)   ! c3 !
!-------!--------------!-------------!--------------!----!
! c2    !     c1 (1)   !    c1 (2)   !     c1 (3)   ! c3 !
!-------!--------------!-------------!--------------!----!
! c2    !     c1 (1)   !    c1 (2)   !     c1 (3)   ! c3 !
!-------!--------------!-------------!--------------!----!
! c2    !     c1 (1)   !    c1 (2)   !     c1 (3)   ! c3 !
!-------!--------------!-------------!--------------!----!
! c2    !     c1 (1)   !    c1 (2)   !     c1 (3)   ! c3 !
!-----------------------------------------------------------!
!              bottom line(s)                               !
!-----------------------------------------------------------!
!       Status line  (not altered by this operator)         !
!-----------------------------------------------------------!
```

The result of this operator is the number of "c1" columns.

System variable 447 which reflects the number of fields on the schematic will be modified by this operator to return the number of fields in the "dynamic" schematic. The current screen field (s.v. 448) will not be modified by this operator. The "dynamic" schematic has no processes related to it so those processes related to the previous schematic on the screen are still in force. Those system variables related to the next, current, and previous schematic name (s.v. 431, 403, and 511 respectively) are not modified by this operator.

This operator can be used for "spread-sheet" like displays. The ATTR operator can be used to override attributes while the INPUT operator can be used to override field verification (and accept keyboard input into fields).

Operator Name: SCHEMA

Class: Control

Arguments: L R1 R2

Result: Yes, error code

Summary: sch SCHEMA <grp,fil> -> err

Description:

This operator will place the selected user or system schematic on the screen immediately.

This operator may have a left argument. This operator may have two right arguments. This operator returns a result which is an error code.

The left argument is a schematic name. User schematics must commence with an alphabetical character while system schematics must be decodable as numbers (i.e. the name must be made up of digits). If the given name is a null string, or a string full of spaces, or an invalid system schematic number then system schematic 1 will be placed on the screen.

If the left argument is not given then the schematic name in system variable 431 is assumed. If the left argument is given then it replaces the previous contents of system variable 431. If this operation is successful then the previous schematic name is placed in system variable 511. If this operation is successful then the new schematic name is placed in system variable 403.

The first right argument is the schematic group name. It is only significant for getting user schematics. If not given then the group name in system variable 502 is used. If a new group name is given then this new name is placed in system variable 502.

The second right argument is the schematic file name. It is only significant for getting user schematics. If not given then the file name in system variable 501 is used. If a new file name is given then this new name is placed in system variable 501.

The result of this operator is an error code. If the operation is successful then zero is returned. The most common error codes for this operator are listed below:

| Error code | Meaning |
|---|---|
| 0 | Operation successful |
| 27 | No such user schematic file name |
| 28 | No such user schematic group name |
| 29 | No such user schematic name |
| 42 | Schematic file in wrong format |
| negative | The schematic file is corrupted |

User schematics are created and edited by the module called SKJEMA. System schematics are created and edited by the module called VISETUP. System schematics are referenced by number. A system file can hold up to 49 system schematics which are numbered 1 to 49.

The action of this operator is to place a new schematic on the screen.

If this action is successful, zero is returned as the result and the following occur (as well as those things already noted above): The current screen field system variable (448) is set to 1. The previous screen field system variable (510) is set to zero. Note that there is no control transfer thus execution continues on the line where the SCHEMA operator was found. Care should be taken when this operator is executed from within a schematic related process (i.e. BEGIN, END, or a screen field related process). In this case VPL's fetch of the next line to be interpreted will be in the context of the new schematic.

If this action is unsuccessful the appropriate non-zero error code is returned. The context is not changed. If the SCHEMA operator was executed from within a schematic related process then execution can continue as if nothing happened. Note that s.v. 431 will reflect the schematic name which was unable to be placed on the screen. System variable 403 (current schematic) and 511 (previous schematic) will remain unaltered.

Schematics which were defined by the SKJEMA module to be smaller than the current screen will be centered by this operator.

In the current implementations of VISTA16 the file name extension '.VUS' is assumed in the host operating systems.

Summary of system variables related to the SCHEMA operator:

| System variable | Comments |
|---|---|
| 403 | Always reflects the name of the schematic currently on the screen. |
| 511 | Always reflects the name of the schematic previous on the screen. |
| 431 | If this operator had a left argument then it is recorded here. |
| 502 | If this operator had a first right argument (group name) and if the schematic name is one of a user schematic then the group name is recorded here. |
| 501 | If this operator had a second right argument (file name) and if the schematic name is one of a user schematic then the file name is recorded here. |
| 448 | If the operation is successful then current screen field is set to 1. |
| 510 | If the operation is successful then previous screen field is set to zero. |

Examples:

```
        IF SCHEMA EQ 0 THEN GOTO '-1' ENDIF
```

This is a safe way of using the SCHEMA operator. The whole conditional clause is on 1 VPL line so that it will not be affected by the successful execution of the SCHEMA operator regardless of where this line is executed from (this could not be done from inside a procedure since the GOTO operator is illegal there). In this case the schematic name currently in #431 will be used. If it indicates a user schematic then the group name and file name in #502 and #501 respectively are used. If successful the following will happen:

```
                #403    ->  #511
                #431    ->  #403
                1       ->  #448
                0       ->  #510
```

```
        13      SCHEMA  ->  0       ;bring up system schematic 13

        '13'    SCHEMA  ->  0       ;bring up system schematic 13

        0       SCHEMA  ->  0       ;illegal system schematic number so
                                    ;system schematic 1 is brought up

        'test'  SCHEMA  ->  27      ;no such user schematic name in the
                                    ;current user schematic group and
                                    ;file
```

```
Operator Name:            SCLOSE             !
                          ------             !
                                             ---------------------
```

Class: Printing and sequential file handling

Arguments: R

Result: Yes, error code

Summary:

```
                  SCLOSE  un     ->   err
```

Description:

This operator will close a sequential file.

This operator may have a right argument. If so, it must be a number. This operator returns a result which is an error code.

The right argument is the unit number. If not given it is assumed to be 10.

The result is an error code. If the file is successfully closed then zero is returned. The most common error codes are listed below:

| Error code | meaning |
|---|---|
| 0 | No errors |
| 53 | Illegal unit number |
| 58 | Sequential file not open |

When the VIPS is terminated (e.g. by using 8 MODE) all open files will be closed. It is recommended that the application designer close sequential files after use rather than waiting until the termination of VIPS. Most operating systems limit the number of files that can be concurrently open.

Examples:

```
'TEXT.TMP'  SOPEN   18        ->  0   ;file successfully opened
                                      ;with unit number 18
250       SREAD   <18,,#201>          ;read 250 bytes

          SCLOSE  18          ->  0   ;file successfully closed

          SCLOSE  18          ->  58  ;file no longer open
```

Operator Name: SDELETE

| | |
|---|---|
| Class: | Printing and sequential file handling |
| Arguments: | L* |
| Result: | Yes, error code |
| Summary: | str SDELETE -> err |

Description:

This operator will delete a file.

This operator requires a left argument. This operator returns a result which is an error code..

The left argument should be the file name to be deleted. The format of the file name depends on the host operating system. The left argument cannot exceed 255 characters in length.

The result is an error code. If the file is successfully deleted then zero is returned. The most commonly returned error codes are:

| Error code | Meaning |
|---|---|
| 0 | No error |
| 52 | File not found |

Examples:

```
'TEXT.TMP'  SDELETE     ->  0      ;successfully deleted

'TEXT.TMP'  SDELETE     ->  52     ;file not found
```

```
Operator Name:              SEARCH              !
                            ------              !
                                                --------------------
Class:          Database

Arguments:      L   R1* R2  (R3 see extensions)

Result:         Yes

Summary:        sp      SEARCH  <fds,ln>    ->  num

N.B.            This operator has extensions
```

Description:

This operator will apply the given search profile to a database register and produce a list of all documents satisfying that profile. The produced list is called an "occurrence list".

This operator may have a left argument. The first right argument must be given. If a second right argument is given it must be a number; non-integers are rounded to integers if necessary. This operator returns a result.

The left argument is the search profile. A search profile is essentially a string which some documents in the database are thought to contain within a given field.

A search profile can only be used to match a given field that was stored as a key field.

A search profile can only be used for "exact" matches. The term "exact" is written thus because the search profile is transformed before it is applied to the database's dictionary. This transformation comprises of folding the search profile to upper case and removing all spaces. This transformation makes an "exact" match a little more likely! A key stored as 'Peter' will match with ' Peter', 'PETER', 'pETER', ' p ET e R ', and of course 'Peter'.

The SEARCH operator is the fundamental (and only) operator for retrieving information from the database. It is a fast operation. Even though search time increases with the number of documents in a register (and the database as a whole) the increase is much better than linear. For matches on non-key fields and for inexact matches the SELECT operator can be used. The SELECT operator's speed is directly proportional to the number of documents in its input list.

The first right argument is a field descriptor.

The format of the field descriptor is as follows:

```
        reg:nam.ext
                      reg  the is register name
                      nam  the is searchable part of name
                      ext  the is non-searchable part of name
                           (not required by SEARCH operator)
```

The register name should be given to the SEARCH operator. It must be followed by a semicolon. When no field name is given then the search profile is ignored and an occurrence list of all documents in this register is generated. If no register name is given but a field name is given then the current schematic name in system variable 403 is assumed as the register name. This latter technique is not recommended.

The searchable part of the name may be given. If so it will be the field name in the given register in which the search profile is to be applied. Two methods of field naming are supported. The first method is by number in which the field name can contain up to three digits. The second method is by a string which can be up to 31 characters long and must not start with a digit (or contain ":", ".", or space).

The extension will be ignored if given.

The second right argument is the list number. If it is given then it should be a number in the range 1 to 101. The current occurrence list is list 101. If the second right argument is not given then the current occurrence list is assumed. In all cases the contents of the given occurrence list before the execution of the SEARCH operator will be replaced by the occurrence list generated by the search. If the SEARCH operator does not find any documents then the given occurrence list will be empty.

The result of this operator is the number of documents found.

It should be noted that blank or null strings are never stored in the database's dictionary, therefore a blank search profile (e.g. '  ') will always find zero documents.

If no documents exist in the register being searched then 1 is put in system variable 523. If documents exist in the register being searched then 0 is put in #523.

If a database is not open when this operator is executed then zero will be returned and system variable 523 set to 1.

Examples:

```
          SEARCH   'client:'                    ->  47
          #523                                  ->  0
```

This will find all documents associated with the register name 'client' and generate a list which replaces the previous contents of the current occurrence list. The result of this operator indicates 47 documents have been found in that register. The contents of system variable 523 indicates that documents where found in the register being searched.

```
            'Smith' SEARCH  'client:name'       ->  2
```

This will find the all documents in the 'client' register that have 'Smith' (or 'SMITH' or 'SMI  th', etc) in a key field called 'name'. The newly generated list will replace the previous contents of the current occurrence list. The result of this operator indicates 2 documents have been found.

```
            'Smith' SEARCH  <'client:name',37>  ->  2
```

This example is similar to that above. This time the generated occurrence list replaces the previous contents of occurrence list 37.

Extension 1:          Third right argument
------------

As a convenience this operator can have a third right argument. This third right argument is a register name. The register name must not be followed by ":".

```
        sp      SEARCH      <fds,ln,reg>     ->  num
```

During the programmatic use of the SEARCH operator it may be easier to put the register name as the third right argument rather than concatenate it with a semicolon and the field name (using the JOIN operator). If a third right argument is given and the field descriptor also contains a register name then the third right argument takes precedence.

Extension 2:          Multiple keys
------------

The left argument may be a search profile containing several keys separated by semicolons. In a similar fashion the field descriptor can have several field names separated by semicolons. Successful documents must have all the component keys and corresponding field names matching (implied LAND operation between component lists). If the number of component keys exceeds the number of field names then the last field name is considered to be repeated as often as required.

The case of blank or null component keys is treated differently in this extension. A blank or null component key (and its corresponding field name) is ignored.

The key delimiter can be altered by placing a character in system variable 526. This system variable is initialized to semicolon. Regardless of the character in #526 multiple field names are always separated by semicolons.

Example:

```
        'Smith;33'  SEARCH  'client:name;age'       ->  1
```

This will search in the register 'client' for 'Smith' in a key field called 'name' AND '33' in a field called 'age'. The result indicates that 1 such document has been found.

Extension 3: No occurrence list generated ln=-1

-----------

If no output occurrence list is required then a list number (second right argument) of -1 can be given. None of the existing occurrence lists will be effected. The number of documents satisfying the search profile will still be returned as the result of this operator.

Extension 4: Number of occurrences ln=-2

-----------

When a document is stored it is possible to store multiple keys in one field. This is usually done by placing semicolon between the required keys in the string to be stored in a key field. In the most complicated case it would be possible to store the same key twice in the same field of one document. In this case a SEARCH for that key will produce an occurrence list with that document entered only once, and the result reflects the number of documents in the generated occurrence list.

If the list number (second right argument) is given as -2 then no occurrence list is generated and the result is the number of times the given key "occurs" in the given field name in the given register.

| | | |
|---|---|---|
| Operator Name: | SELECT | ! |

Class: Database

Arguments: L R1 R2* R3 R4 R5 R6 R7 R8 R9 R10

Result: Yes

Summary: lni SELECT <lno,fds1,str1,type1,fds2,...,type3> -> num

Description:

This operator will allow documents from one list to be selected on up to three criteria and the successful documents placed in a list.

This operator may have a left argument. If it does it must be a number; non-integers are rounded to integers if necessary. This operator can have up to 10 right arguments. The second right argument must be given. If given the first, fourth, seventh, and tenth right arguments must be numbers; non-integers are rounded to integers if necessary. This operator returns a result.

The left argument is the input list number. If given it should be a number in the range 1 to 101. The current occurrence list is referred to as list 101. If no left argument is given the current occurrence list is assumed. The input occurrence list will be scanned in a linear fashion by the SELECT operator. Therefore the speed of this operator is proportional to the length of the input occurrence list.

The first right argument is the output occurrence list number. If given it should be in the range 1 to 101. The current occurrence list is referred to as list 101. If the first right argument is not given the current occurrence list is assumed. The output occurrence list will contain the "successful" documents found in the input list which meet the criterion. The input and output occurrence lists can have the same number (or both default to the current list) if necessary. The previous contents of the output occurrence list are replaced.

The second, third, and fourth right arguments are associated with the first select criterion, while the fifth, sixth, and seventh right arguments are associated with the second select criterion, leaving the eighth, ninth, and tenth right arguments to be associated with the third select criterion. Only the first select criterion is required. Of the right arguments associated with it only the second right argument (field descriptor) must be given. If more than one select criterion is given then a document must meet all the given criteria to be "successful".

The three arguments associated with each select criterion are called the "field descriptor", "match string", and "type". Their right argument position is shown in the following table:

| Right argument pos. | field decriptor | match string | type |
|---|---|---|---|
| 1st select criterion | 2 | 3 | 4 |
| 2nd select criterion | 5 | 6 | 7 |
| 3rd select criterion | 8 | 9 | 10 |

The format of the field descriptor is as follows:

```
        reg:nam.ext

            where:
                     reg  is register name (ignored by SELECT)
                     nam  is searchable part of name
                     ext  is non-searchable part of name
```

The register name is not required by the SELECT operator and will be ignored. It may be useful to have the register name present from the point of view of checking that the register name is the same as that which the input occurrence list was generated by. In the future the interpreter may check this.

The searchable part of the name must be given and be non-blank. The field name "0" (zero) is reserved for a field containing the register name of the document (put in there by the CREATE operator). Two methods of field naming are supported. The first method is by number in which the field name can contain up to three digits. The second method is by a string which can be up to 31 characters long and must not start with a digit (or contain ":", ".", or space).

The extension is optional and can be up to 3 alphanumeric characters long. If the field was defined with an extension, then the same extension must be given to the SELECT operator which references that field.

The "match string" is used to check the given field of the documents in the input list. The "match string" will have leading, trailing and repeated imbedded spaces (delimiters) removed before the comparison is performed.

The "type" controls the comparison between the "match string" and the given field of the documents in the input list. Currently 12 types of comparison are allowed and they are listed below:

| type | meaning | |
|---|---|---|
| 6 | Less than | (document field < match string) |
| 5 | Greater than or equal | (document field > match string) |
| 4 | Excluded wild select | (use of ? and * ) |
| 3 | Wild select | (use of ? and * ) |
| 2 | Not equal | |
| 1 | Equal | |
| 0 | Equal | (default) |
| -1 | Equal after fold to upper case | |
| -2 | Not equal after fold to upper case | |
| -3 | Wild select after fold to upper case | |
| -4 | Excluded wild select after fold to upper case | |
| -5 | Greater than or equal after fold to upper case | |
| -6 | Less than after fold to upper case | |

The fields obtained from the documents have leading, trailing, and repeated imbedded spaces (delimiters) removed before comparison with the "match string". If the type is negative then both the field and the "match string" are folded to upper case before the comparison is made. If the given field does not exist in a document then it is treated as a null string.

The "equal" and the "not equal" types should be obvious.

The "wild select" takes all characters literally except for "?" and "*".
A "?" in the "match string" will match any character in the corresponding position in the field. The corresponding field position must have a character in that position. Thus a "match string" of "?" will not match with a field which is a null string (no characters).
A "*" in the "match string" will match with a variable number (0 to 255) of characters from the corresponding position in the field. If the "match string" has a character after the "*" then character for character matching will recommence when that character is detected in the field.
If the "match string" does not contain either "*" or "?" then "wild select" has the same effect as "equal".

The "excluded wild select" is the logical complement of "wild select". Thus a document which is "unsuccessful" in a "wild select" will be "successful" in an "excluded wild select".
If the "match string" does not contain either "*" or "?" then "excluded wild select" has the same effect as "not equal".

The "greater than or equal select" will compare the field in the document with the "match string" and judge a document as "successful" if it is the same or greater. The comparison is performed left to right and spaces are added so both strings are equal length. The character ordering is assumed to be ASCII. For example, if the "match string" is "B" and the field is "CHARLES" then this criterion would be successful.

The "less than select" will compare the field in the document with the "match string" and judge a document as "successful" if it is the smaller. The comparison is performed left to right and spaces are added so both strings are equal length. The character ordering is assumed to be ASCII. For example, if the "match string" is "D" and the field is "Charles" then this criterion would be successful.

Delimiters other than space (the default) can be used. This can be done by writing the new delimiters to system variables 524 and 525. After the removal of redundant delimiters for the purposes of the comparison all delimiters are transfomed to spaces (the lowest numbered printable character in the ASCII sequence). Note that the database system never stores trailing spaces and the storage of leading spaces is conditional on system variable 539 (default is storage without leading spaces).

After the above-mentioned transformation to remove redundant delimiters no more than 80 characters are significant in the comparison for each criterion.

Examples:

```
            SEARCH       'namreg:'           ->       47
```

Make a list (current list) of all documents in register 'namreg'. The result indicates 47 documents have been found.

```
            SELECT  <13,'surname'>           ->       2
```

This would scan the 47 documents in the current list and form a new list (list 13) of documents which have nothing in the field 'surname'. Since "match string" is not given it defaults to '' (a null string) and since type is not given it defaults to "equal".
N.B. This SELECT could not be performed by the SEARCH operator even if 'surname' was a key field because it is not possible to search for a null string.

```
            SELECT  <13,'surname','????',3>          ->  7
```

This would scan the 47 documents in the current list and form a new list (list 13) of documents which have 4 letters (no more, no less) in the field 'surname'.

```
            SELECT  <13,'surname','Smith','-5'>      ->  9
```

This would scan the 47 documents in the current list and form a new list (list 13) of documents which, after folding to upper case, are greater than or equal to 'SMITH'. Thus surnames such as 'Thomas', 'Smithe' and 'Smith' would be "successful".

```
        SELECT  <,'surname','Smith','-6','surname','Jones','-5'>  -> 15
```

This would scan the 47 documents in the current list and form a new list (current list) of documents which, after folding to upper case, are less than 'SMITH' AND greater than or equal to 'JONES'. Both parts of the two criteria have to be true for the document to be considered "successful".
N.B. The the current list would contain 15 documents after this SELECT operator.

```
        SELECT  <7,'surname','*er','-4'>    ->  14
```

This would scan the 15 documents in the current list and form a new list (list 7) of documents which, after folding to upper case, do not end with 'ER'.

Operator Name: SEQ

Other operators
described here: SNE

Class: String

Arguments: L* R1* R2

Result: Yes, condition code

Summary:

| | | | | |
|---|---|---|---|---|
| str1 | SEQ | <str2,type> | -> | cc |
| str1 | SNE | <str2,type> | -> | cc |

Description:

These operators will compare two strings and return a condition code to indicate whether they are equal or unequal.

These operators must have a left argument. These operators must have a first right argument. If these operators have a second right argument then it must be a number. These operators return a result.

The left argument and the first right argument of these operators are the strings to be compared. The second right argument is the type of comparison to be performed.

The SEQ operator will return the true condition code (1) if the two strings are equal and the false condition code otherwise (0).

The SNE operator will return the false condition code (0) if the two strings are equal and the true condition code otherwise (1).

The valid types and their meaning is listed below:

| type | meaning |
|---|---|
| (not given) | Fold both strings to upper case and remove leading, trailing, and repeated imbedded spaces (delimiters). |
| 1 | Remove all spaces (delimiters) from both strings. |
| 2 | Remove leading, trailing, and repeated imbedded spaces (delimiters) |
| 3 | Compare strings as is |
| 11 | Fold both strings to upper case and remove all spaces (delimiters) |
| 12 | Fold both strings to upper case and remove leading, trailing, and repeated imbedded spaces (delimiters). This is the same as the default (type not given). |
| 13 | Fold both strings to upper case. |

The type 3 compare is literally exact. Both strings must be the same length and be character for character identical. The type 13 compare is similar but is performed after both strings are folded to upper case.

The type 1 compare will remove all spaces (delimiters) from both strings before the comparison. The type 11 compare is similar but is performed after both strings are folded to upper case.

The type 2 compare will remove leading, trailing, and repeated imbedded spaces (delimiters) from both strings before the comparison. The type 12 compare is similar but is performed after both strings are folded to upper case. Experience has shown that this last type of comparison is the most commonly used so it has been made the default.

Delimiters other than space (the default) can be used. This can be done by writing the new delimiters to system variables 524 and 525.

Examples:

```
            'John    Smith '    SEQ     'JOHN SMITH'    ->  1   ;true
            'John    Smith '    SNE     'JOHN SMITH'    ->  0   ;false


            'John    Smith '    SEQ     'JOHNSMITH'     ->  0   ;false
```

The space between the "John" and the "Smith" is significant when type is not given (same as type=12)

```
            'John    Smith '    SEQ     <'JohnSmith',2> ->  1   ;true
```

Now spaces are not significant at all.

```
            'John Smith '       SEQ     <'John Smith ',3> -> 1  ;true
```

An exact match has been called for and the strings are identical.

```
            ';' =   #524
            ' ' =   #525
            ';john; ;smith '  SEQ     'John Smith'      ->  1   ;true
```

Both strings are folded to upper case while leading, trailing, and repeated imbedded delimiters are removed before the comparison. Note that the two delimiters are taken to be equal to one another.

```
'Return to "open" system? (y/N)'   =   #1
IF  2   INPUT <,1> SEQ  'y'    THEN
    0   MODE
ENDIF
```

This would place the question in the first screen field then wait for input in the second field. Only one character will be accepted and if it is "Y" or "y" then control will return to the super begin process of mode 0. This is a way for a "closed" application to re-enter the "open" system.

Operator Name: SHELP

| | |
|---|---|
| Class: | Special |
| Arguments: | L* R1 R2 R3 R4 R5 R6 R7 R8 |
| Result: | No |
| Summary: | num SHELP <sch1,sch2,sch3,sch4,sch5,sch6,sch7,hgrp> |

Description:

This operator will define the schematics which are to be used for the help structure. It will either disable the help structure or enable it by nominating which schematic will be displayed when the F2 key is pressed. The help structure can be driven by the HELP operator.

This operator requires a left argument which must be a number. This operator can have up to 8 right arguments. This operator does not return a result.

The left argument should be a number (rounded to an integer if necessary) in the range 0 to 7. Zero indicates the help structure is to be disabled. When the help structure is disabled pressing the F2 key will pass codes to the system in the same way that the other function keys do.

If the left argument is 1 to 7 then the help structure is enabled. The right argument corresponding to the number (e.g. 1 -> first right argument, 2-> second right argument, etc. ) will be the schematic which will be placed on the screen when the F2 key is pressed.

The first seven right arguments are schematic names. Schematic names can be up to 20 characters long. User schematic names must not start with digits. System schematics can be nominated as help schematics by writing their numbers. Any of the first seven right arguments which are not defined will cause the bell to ring if they are selected in a help structure.

The eighth right argument is the schematic group from which user schematics are to be fetched while in the help structure. If this argument is not given then the schematic group name in system variable 502 at the time of execution of this operator is assumed. Thus all user schematics in a help structure must belong to the same schematic group.

The user schematic file name at the time when this operator is executed (current contents of system variable 501) will be assumed within the help structure. Thus all user schematics in a help structure must belong to the same schematic file.

## Help Structure

This refers to the mechanism that allows the current state to be "interrupted" and a schematic to be brought up on the screen with the option of more schematics being selected. Pressing the space bar (or F8) will restore the screen to the state just prior to the "interrupt".

Once the help structure is enabled (positive left argument to the SHELP operator) then pressing the F2 key will cause the nominated schematic to be placed on the screen. If the nominated schematic cannot be found then the bell is rung and the help structure is terminated.

Once the help structure has been successfully invoked then the following keys are active:

| Active keys in help structure | action |
|---|---|
| 1 | get 1st help schematic (1st right arg. of SHELP) |
| 2 | get 2nd help schematic (2nd right arg. of SHELP) |
| 3 | get 3rd help schematic (3rd right arg. of SHELP) |
| 4 | get 4th help schematic (4th right arg. of SHELP) |
| 5 | get 5th help schematic (5th right arg. of SHELP) |
| 6 | get 6th help schematic (6th right arg. of SHELP) |
| 7 | get 7th help schematic (7th right arg. of SHELP) |
| Space | return to the pre-help structure state |
| F8 | return to the pre-help structure state |
| F2 | get help schematic nominated by SHELP (left arg.) |

If the schematic corresponding to one of the numbers 1-7 cannot be found then the bell is rung and the help structure is terminated.

The help schematics may have less rows and columns than the screen they are overwriting. In this case the first schematic of the help structure (when F2 is pressed) is placed in the furthest corner from the cursor position. Thus if the help schematic is relatively small (e.g. less than one quarter of the area of the schematic it is covering) then the section of the original schematic around the cursor will not be overwritten.

If more help schematics are selected in the help structure then they use the same corner as the original help schematic as a reference. It should be noted that when "small" help schematics are being made by the SKJEMA program then they should be written in the top left hand section. SKJEMA decides the number of lines and columns in a schematic in order to include all significant positions (including data fields). This "decision" by SKJEMA effects the amount of the original schematic overwritten by a help schematic.

Examples:

```
        3    SHELP    <21,22,23,24,'SORT_EXP','ERRORS','BYE','HELP'>
        #501    ->    'DOCTORS'
```

This usage will define seven schematics for the help structure. System schematics 21, 22, 23, 24 are nominated as help schematics 1, 2, 3, and 4 respectively. User schematics 'SORT_EXP', 'ERRORS', and 'BYE' in the schematic group 'HELP' in the schematic file 'DOCTORS' are nominated as help schematics 5, 6, and 7 respectively.

The help structure is enabled and when F2 function key is pressed system schematic 23 (3rd right argument) will be placed on the screen in the furthest corner from the cursor. After this pressing the numbers 1 to 7 would bring up the corresponding schematics nominated in the right argument of the SHELP operator.

For example, pressing 7 would attempt to bring up a schematic called 'BYE' in the group 'HELP' from the schematic file 'DOCTORS'.

Once the help structure has been entered (by pressing F2) then pressing space or F8 will return to the "pre-help structure" state.

```
        0        SHELP
```

Disable the help structure. Pressing F2 will place -2 in system variable 509 and 2 in system variable 453 (i.e. same action as other function keys).

```
        6    SHELP    <21,22,23,24,'SORT_EXP','ERRORS','BYE','HELP'>
```

Re-enable the help structure. This is similar to the first example but the schematic called 'ERRORS' in group 'HELP' in schematic file 'DOCTORS' will be brought up when the F2 key is pressed to enter the help structure.

```
Operator Name:          .   SL                    !
                            --                    !
                                                  ---------------------
Class:          Status line

Arguments:      L*  R

Result:         No

Summary:        num     SL      type
```

Description:
------------

This operator enforces the new status line fields with proportions, attributes, and field verification as defined by previous SP, SA, and SV operators.

This operator must have a left argument which must be a number. This operator can have a right argument. If a right argument is defined it must be a number. This operator does not return a result.

Numbers given to this operator are rounded to integers.

The left argument is the number of status line fields to be defined. It should be a non-negative number. It should not exceed 10 which is the maximum number of fields allowed on the status line.

The right argument is the type. If given, it should be 0 or 1. If it is not given then a type of 0 is assumed. If the type is zero then the numbers given to the most recent SP operator are taken as field lengths for the given number of status line fields. If the type is 1 then the numbers given to the most recent SP operator are taken as the relative proportions that the given number of status line fields will assume.

When this operator is executed the previous contents of the status line are replaced by a blank line which may or may not have attributes set in some positions (depending on the most recent SA operator).

Due to problems with screens scrolling, the last position on the status line is not available. Thus if there is 80 columns on the screen the maximum length of the status line is 79 characters.

Examples:

```
        #521        ->  80      ;thus 79 usable characters on status
                                ;line
        10  SP  30              ;either field 901 will have 10 chars
                                ;and field 902 will have 30 chars or
                                ;they will have the proportion 1:3

        'N' SV                  ;#901 will only accepts digits and
                                ;space while #902 will accept anything

        0   SA  7               ;#901 will have attribute 0 while #902
                                ;will have attribute 7
```

```
 2   SL                  ;now redefine the status line to have
                         ;two fields of length 10 and 30 chars.
                         ;respectively

 901 FLENG   ->  10
 902 FLENG   ->  30
 903 FLENG   ->  0       ;status line field 903 (and 904 to 910)
#903 EXIST   ->  0       ;doesn't exist


 2   SL  1               ;now redefine the status line to have
                         ;two fields of length 19 and 59 chars.
                         ;respectively (relative proportioning)

 901 FLENG   ->  19
 902 FLENG   ->  59
 903 FLENG   ->  0       ;status line field 903 (and 904 to 910)
#903 EXIST   ->  0       ;doesn't exist
```

Operator Name: **SOPEN**

Class: Printing and sequential file handling

Arguments: L* R1 R2

Result: Yes, error code

Summary: str SOPEN <un,type> -> err

## Description:

This operator will open a sequential file. If the operating system treats all files in a similar fashion then all types of files may be accessed by this operator. Some operating systems will allow byte-oriented devices to be accessed as files in which case they may also be accessed by this operator.

This operator assumes that the host operating system will allow a file to be viewed as a stream of characters.

This operator requires a left argument. This operator may have a first and second right argument and if so it (they) must be a number(s). This operator returns a result which is an error code.

The left argument is a filename. No extension is assumed so that the file name must be given in its entirety. The VPL interpreter will allow a string to be up to 255 characters long so that is the maximum length of a file name that can be passed through to the operating system.

The first right argument is the unit number. If an non-integer is given then it is rounded to an integer. If the unit number is given it must be 10 or greater. If the first right argument is not given then the unit number 10 is assumed. If the SOPEN operator is successful then the given unit number can be used with the other operators in this class (i.e. SCLOSE, SREAD, SPOS, PRINT, PRCHAR, and PRSTR). The number of units that can be opened simultaneously is operating system dependent.

The second right argument is the type. If the second right argument is not given then a type of zero is assumed. The following table lists the valid types:

| type | meaning |
|---|---|
| 0 | Open for read/write. Create new file if one of this name does not already exist. Position at first character in file. |
| 1 | Open for read only. File must exist. Position at first character in file. |
| 2 | Open for write only. Create new file if one of this name does not already exist. Position after the last byte in file. |

The positioning of the byte (character) pointer is meant to be useful and can easily be modified by the SPOS operator.

This operator returns a result which is an error code. A result of zero indicates no error. A list of the more commonly encountered errors follows:

| error number | meaning |
|---|---|
| 52 | file not found (only when type is 1) |
| 53 | illegal unit number (must be 10 or greater) |
| 56 | unit number already in use |
| 57 | too many files open (OS dependent) |

Examples:

```
        'VIPS.LST'  SOPEN        ->0      ;this will open the file
                                          ;'VIPS.LST' and associate it
                                          ;with unit 10

        'VIPS1.LST' SOPEN        ->56     ;unit 10 already in use
                                          ;N.B. 'VIPS.LST' still open

        .....                             ;assume some intervening code

        PRINT       10           ->0      ;send an image of the current
                                          ;schematic to 'VIPS.LST'

        1    SPOS                         ;position to first byte of
                                          ;'VIPS.LST'

        SREAD   <,10>    ->  #301         ;read the first line of
                                          ;'VIPS.LST' into #301
                                          ;assume line terminated by LF

        SCLOSE           ->  0            ;'VIPS.LST' is now closed
```

The above example has succeeded in reading the first line of the schematic (regardless or whether it is text or data fields) into long hidden field 301.

```
        'VIPS.LST'  SOPEN   <11,2>  ->0 ;Re-open 'VIPS.LST' for
                                        ;write only and position to end
                                        ;of file.

        SREAD   <,11,#201>      ->  ''
        #201                    ->  54  ;the file is open for write
                                        ;only

        PRINT   11                      ;append the current schematic
                                        ;image to the previous contents
                                        ;of 'VIPS.LST'

        SCLOSE  11                      ;recommendation: close a
                                        ;sequential file after use
```

```
Operator Name:          SORT                 !
                        ----                 !
                                             --------------------
Class:         Database

Arguments:     L  R1* R2  R3  R4  R5  R6  R7  R8  R9  R10

Result:        Yes, error code

Summary:       ln  SORT   <fds1,type1,fds2,...fds5,type5> ->  err
```

Description:
------------

This operator will sort an occurrence list. Up to five fields may be sorted at one time. If more are needed the SORT may be invoked several times, each time using the result of the previous SORT as input. Sorting can be done on a character or numeric interpretation of the data in an ascending or descending order.

The documents themselves are not physically re-ordered but the occurrence list which is a list of pointers to documents is re-ordered.

This operator may have a left argument. If it does it must be a number; non-integers are rounded to integers if necessary. This operator can have up to 10 right arguments. The first right argument must be given. If given the 2nd, 4th, 6th, 8th, and 10th right arguments must be numbers; non-integers are rounded to integers if necessary. This operator returns a result.

The left argument is the list number. If given it should be a number in the range 1 to 101. The current occurrence list is referred to as list 101. If no left argument is given the current occurrence list is assumed. At the completion of the SORT operator the newly re-ordered occurrence list will replace the original occurrence list in the given list number.

The meaning of the the ten right arguments is shown in the following table:

| Right argument pos. | field decriptor | type |
|---|---|---|
| 1st sort field | 1 | 2 |
| 2nd sort field | 3 | 4 |
| 3rd sort field | 5 | 6 |
| 4th sort field | 7 | 8 |
| 5th sort field | 9 | 10 |

The 2nd sort field will only be taken into account if two or more documents are judged to be equal on the 1st sort field. The 3rd sort field will only be taken into account if two or more documents are judged to be equal on the 1st and 2nd sort field. The 4th and 5th sort fields are treated in a similar way. It can be seen that each of the five field descriptors has its own type so that each field can be compared independently of the type of other fields.

The field descriptors have the following form:

```
reg:nam.ext

     where:
                reg  is register name (ignored by SORT)
                nam  is searchable part of name
                ext  is non-searchable part of name
```

The register name is not required by the SORT operator and will be ignored. It may be useful to have the register name present from the point of view of checking that the register name is the same as that which the given occurrence list was generated by (i.e. a SEARCH operator). In the future the interpreter may check this.

The searchable part of the name must be given and be non-blank. The field name "0" (zero) is reserved for a field containing the register name of the document (put in there by the CREATE operator). Two methods of field naming are supported. The first method is by number in which the field name can contain up to three digits. The second method is by a string which can be up to 31 characters long and must not start with a digit (or contain ":", ".", or space).

The extension is optional and can be up to 3 alphanumeric characters long. If the field was defined with an extension (i.e. by a PUT operator) then the same extension must be given to the SORT operator which references that field.

The "type" controls the comparison between the documents in the given list. Currently 6 different types of comparison are allowed and they are listed below:

| type | meaning |
|---|---|
| 3 | Numeric sort in ascending order |
| 2 | Unfolded character sort in ascending order |
| 1 | Characters folded to upper case then ascending sort |
| 0 | Characters folded to upper case then ascending sort |
| -1 | Characters folded to upper case then descending dort |
| -2 | Unfolded character sort in descending order |
| -3 | Numeric sort in descending order |

All fields obtained from the documents have leading, trailing, and repeated imbedded spaces (delimiters) removed before comparison. From the point of view of sorting, all delimiters are treated as spaces. Delimiters other than space can be used by writing them to system variables 524 and 525.

If the type is 1 or -1 then all fields are folded to upper case before they are compared.

If the type is 3 or -3 then all fields will be interpreted as numbers. If a field is non-numeric then an attempt is made to remove all non-numeric characters from the field. Numeric characters are taken as 0 1 2 3 4 5 6 7 8 9 . - . If what is left cannot be interpreted as a number (e.g. two "."s) then both "-"s and "."s will be ignored leaving just the ten digits (or a null string) which must be a valid number. If the given field does not exist in a document then it is treated as a null string and numerically as zero.
Once a number is obtained from a field it is converted into a double real internally so that it should be nearly impossible to have a number which is too big. The accuracy of comparison will depend on the underlying hardware but could be in the order of 15 digits and thus more than sufficient for most applications.

For character sorts (types 1, -1, 2, and -2) the ASCII sequence is assumed. In this sequence the space is the lowest while "z" is close to the highest. If a field is a null string then it can be considered to contain one space and thus would be appear at the beginning of an ascending sort. In character sorts after leading, trailing, and repeated imbedded delimiters are removed only the first 32 characters of each field is significant in the comparison for the sort.

The result is an error code. Zero indicates no error.

It is not possible to interrupt a sort (or any other operator whose execution is in progress).

If the given occurrence list is empty or only contains one document then no sort is performed.

The limits on the sort are implementation dependent. On the smallest configuration in use at time of development this would mean a five field sort overflowing at around 390000 documents. Such an error would by indicated by '** VPL ** Error during sort' on the status line.

The occurrence list which is generated by a sort must not be used as input to the LAND, LOR, LXOR, or LNOT operators.

A point of interest about sorted occurrence list is that if the LDOC operator is used to extract each document from the list and these extracted documents are accumulated in another list with the LOR operator, then the pre-sorted list will be obtained.

The occurrence list which is generated by a sort may be used by another SORT operator or a SELECT operator.

Examples:

```
          SEARCH      'namreg:'          ->      470
```

Make a list (current list) of all document in register 'namreg'. The result indicates 470 documents have been found.

```
            SORT        'namreg:name'        ->       0
```

This would sort the current occurrence list (containing 470 documents) using the field 'name'. This would be a character sort in ascending order after fields were folded to upper case. The resulting occurrence list would be the current occurrence list (still containing 470 documents). The result of this operator indicates that no error was detected.

```
            SORT       <'namreg:name',1,'namreg:zip',3> ->  0
```

This sort would be similar to the previous example except that those documents with 'name' fields that contained the same names would then be sorted numerically on a field called 'zip' in ascending order. The result of this operator indicates that no error was detected.

```
            SORT       <'namreg:name',,'namreg:zip',3>  ->  0
```

This sort is identical to the previous example.

### Extension:

To show the system is still alive in long sort (1000+ documents) it is possible to get some progress digits printed out on the status line.

To get this effect the list number (left argument) should be negated (e.g. -101 for the current list).

Numbers will be output on the status line during the sort. Their meaning is related to the internal sorting algorithm:

| Digit output during sort | Meaning |
|---|---|
| 1 | Sort filled internal buffer, store information awaiting merge. |
| 2 | Commence short merge |
| 3 | Commence medium merge |
| 4 | Commence long merge |

These numbers being written out on the status line may cause the screen to scroll (after 80 digits- 10000+ documents) but the screen will be re-written at the end of the sort.

Operator Name: SP

Class: Status line

Arguments: L R1 R2 R3 R4 R5 R6 R7 R8 R9

Result: No

Summary: pf1 SP <pf2,pf3,pf4,pf5,pf6,pf6,pf7,pf8,pf9,pf10>

Description:

This operator defines the relative proportions or length in characters of each status line field. This operator is passive. The status line fields are set to these proportions/lengths after the next SL operator.

This operator can have a left argument. If so, it must be a number. This operator can have up to 9 right arguments. Any right arguments that are defined must be numbers. This operator does not return a result.

All numbers given to this operator are rounded to integers. Thus relative proportions between status line fields should be expressed in whole numbers (integers). Negative numbers should not be given.

The left argument represents the first status line field (addressed as 901). The first right argument represents the second status line field (addressed as 902). The second right argument represents the third status line field (addressed as 903), and so on. If an argument is not given then the number zero is assumed. Up to 10 status line fields are allowed.

Whether the numbers given by this operator are relative proportions or field lengths depends on the right argument of the SL operator which enforces the new status line fields. If the SL operator does not have a right argument or it is given as 0 then the numbers given by this operator are taken to be field lengths in characters. If the SL operator has a right argument which is 1 then the numbers given by this operator are taken to be relative proportions.

If field lengths have been chosen and if the sum of the character positions to be defined exceeds the usable number of positions on the status line then relative proportioning is used.

If this operator is given no arguments (or all given are zero) then the left argument (corresponding to #901) is taken to be 1.

Due to problems with screens scrolling, the last position on the status line is not available. Thus if there is 80 columns on the screen the maximum length of the status line is 79 characters.

Examples:

```
        #521          ->   80        ;thus 79 usable characters on status
                                     ;line
        10   SP   30                 ;either field 901 will have 10 chars
                                     ;and field 902 will have 30 chars or
                                     ;they will have the proportion 1:3

        'N'  SV                      ;#901 will only accepts digits and
                                     ;space while #902 will accept anything

        0    SA   7                  ;#901 will have attribute 0 while #902
                                     ;will have attribute 7

        2    SL                      ;now redefine the status line to have
                                     ;two fields of length 10 and 30 chars.
                                     ;respectively

        2    SL   1                  ;now redefine the status line to have
                                     ;two fields of length 20 and 59 chars.
                                     ;respectively (relative proportioning)

        901 FLENG     ->   19
        902 FLENG     ->   59
        903 FLENG     ->   0         ;status line field 903 (and 904 to 910)
        #903 EXIST    ->   0         ;doesn't exist
```

| | | |
|---|---|---|
| Operator Name: | SPOS | ! |
| | ---- | ! |

Class: Printing and sequential file handling

Arguments: L R1 R2

Result: Yes

Summary: pos SPOS <un,type> -> pos

Description:

This operator will re-position the byte pointer in a sequential file.

A sequential file can be viewed as a stream of bytes. When a file is open for read/write (SOPEN type=0) the byte pointer is set to the first byte in the file. Reading the whole file with a series of SREAD operators will automatically move the byte pointer away from the beginning and towards the end of file.

The byte pointer is origin one so that 1 is the first position of the file. Reading and writing to the file commences with the byte position pointed to by the byte pointer. At the end of the operation the byte pointer will point to the position after the last byte read or written.

This operator may have a left argument. If so, it must be a number. This operator may have two right arguments. If the first right argument is given it must be a number. If the second right argument is given it must be a number. All number input to this operator will be rounded to integers if necessary. This operator returns a result.

The left argument is the position to move the byte pointer to. If the left argument is not given the byte pointer will not be moved by this operator but the byte pointer position will be returned as the result. If the left argument is given it can either be an absolute position or a position relative to the current byte pointer position. This depends on the second right argument which is the type. If it is not given or zero then absolute positioning is assumed. If the type is given as 1 then relative positioning is assumed.

In the case of absolute positioning positive numbers will move the byte pointer as directed. Positive numbers which exceed the number of bytes in the file will move the byte pointer to the position after the last byte in the file. If the position is given as zero then the byte pointer will be moved to the position after the last byte of the file. Negative numbers will position the byte pointer from the back of the file. Thus -1 will be the position of the last byte in the file, -2 the second last, etc..

In the case of relative positioning positive numbers will move the byte pointer towards the end of file. Negative numbers will move the byte pointer towards the beginning of file. Zero will not move the byte pointer. Positive numbers which are too large will move the byte pointer to the position after the last. Negative numbers which are too large will position to the first byte of the file.

Summary of byte pointer positioning: (N.G.) -> Not Given)

| Value of "pos" | type | meaning |
|---|---|---|
| (N.G.) | (N.G.) | return current byte pointer position |
| (N.G.) | 0 | return current byte pointer position |
| + | 0 or (N.G.) | byte pointer to absolute position |
| - | 0 or (N.G.) | byte pointer to absolute position addressed from the last byte position |
| 0 | 0 or (N.G.) | byte pointer to position after last |
| + | 1 | move byte pointer towards EOF |
| - | 1 | move byte pointer towards SOF |
| 0 | 1 | return current byte pointer position |

The first right argument is the unit number. If not given it is assumed to be 10. This is the unit number used to open the file (see SOPEN), or perhaps may be pre-defined by the host operating system and indicate a device.

The second right argument is the type. If not given it is assumed to be zero. Its action is to modify the interpretation of the left argument ("pos") and is explained above.

The result of this operator is the byte pointer position after re-positioning (if any) has taken place. If this operator cannot be performed because of some file problem (e.g. file not open) then zero is returned as the result.

Examples:

```
'TEXT.TMP'  SOPEN  15  ->  0    ;file opened successfully
            SPOS   15  ->  1    ;pointing at first byte
    0       SPOS   15  -> 345   ;file is 344 bytes long
    1       SPOS   15  ->  1    ;re-position to first byte

22  SREAD   <15,,#201>          ;read first 22 bytes
    ->  'This should be the fir'

            SPOS   15  ->  23

    SREAD   <15,10,#201>        ;read rest of first line
    ->  'st line  '

            SPOS   15  ->  32

    '-8'    SPOS <15,1> ->  24  ;move pointer back 8 bytes

    SREAD   <15,10,#201>        ;re-read rest of first line
    ->  't line  '

'-1'    SPOS    15      -> 344 ;position to last byte

    SCLOSE  15          ->  0   ;file closed successfully
```

```
Operator Name:          .   SQR                      !
                            ---                      !
                                                     ---------------------
```

```
Class:           Arithmetic

Arguments:       L   R (one or the other, right takes precedence)

Result:          Yes

Summary:         num    SQR              ->       num
                        SQR     num      ->       num
                        SQR     (num)    ->       num
             N.B. These are all equivalent
```

Description:
------------

This operator will take the square root of its argument.

This operator requires an argument. It can be either a left argument or a right argument. If both a left argument and a right argument are given then the right argument is used. The argument must be a number. This operator returns a result.

The argument must be a non-negative number. Negative numbers cause a "** VPL ** Attempt to divide by zero" error. The square root of the given argument will be returned as the result.

Examples:

```
          9   SQR          ->      3

              SQR  (9)     ->      3

          2   SQR          ->      1.4142135623731

          2   SQR  49      ->      7
```

```
Operator Name:                SR                  !
                              --                  !
                                                  --------------------
```

Class: Status line

Arguments:

Result: Yes

Summary: SR -> str

Description:
------------
This operator will return the contents of the status line as its result.

This operator requires no arguments. This operator returns a result.

The contents of the status line will be concatenated into one string and returned as the result. Thus it makes no difference if the status line is made up of several fields (by SL operator). Trailing spaces are not returned in the result.

This operator will in no way effect what is on the status line or the relative dispositions of its component fields.

Examples:

```
          2   SP  1                      ;proportion status line fields
                                         ; 2 to 1
          0   SA  0                      ;default attribute on both
          ' ' SV  ' '                    ;default verification on both
          2   SL  1                      ;define 2 fields, relative
                                         ; proportioning
          'Is this a test?' = #901       ;
          902  INPUT                     ;assume "Yes" CR entered


          ---------------------------------------------------------
Status line: !Is this a test?                   Yes               !
          ---------------------------------------------------------

          SR  ->  'Is this a test?                    Yes'
```

| | | |
|---|---|---|
| Operator Name: | SREAD | ! |
| Class: | Printing and sequential file handling | |
| Arguments: | L R1 R2 R3 | |
| Result: | Yes | |
| Summary: | num SREAD <un,match,err> -> str | |

## Description:

This operator will read from a sequential file. If the host operating system is flexible enough then byte oriented devices may also be read from.

The file is viewed as a stream of bytes from which a given number can be fetched, or all those up to and including a given character. In the below explanation the terms "byte" and "character" are interchangeable.

This operator may have a left argument. If so, it must be a number. This operator may have up to three right arguments. If the first right argument is given it must be a number. If the second right argument is given it must be a number. If the third right argument is given it must be writable (i.e. a screen field, a status line field, a hidden field, a writable system variable etc.). If the third right argument is given then an error code is written to it. This operator returns a result.

The left argument is the number of bytes to be read from the file (device). If not given it is assumed to be 1. If the second right argument ("match") is not given or zero then an attempt is made to read the given number of bytes. If that number of bytes is successfully read then an error code of zero is placed in the third right argument and the bytes read are returned as the result of this operator. The number of bytes to be read cannot be less than zero or greater than 255.

It should be noted that the resultant string from this operator may contain any character represented by a number from 0 to 255. For screen handling purposes, VISTA interprets numbers according to an 8 bit ASCII convention. . If control characters or unmapped characters in this sequence are displayed on the screen they will appear as spaces (or some special character as defined for that terminal).

The first right argument is the unit number. If not given it is assumed to be 10. This is the unit number used to open the file (see SOPEN), or perhaps may be pre-defined by the host operating system and indicate a device.

The second right argument is the match character ("match"). The match character is expressed as a number 1 to 255 (N.B. the ASCII null character (value 0) cannot be matched), or alternatively as a number -1 to -255.

When the "match" value is positive then characters will be read until and including the character whose value corresponds to the "match" value is found.

When the "match" value is negative then characters will be read until and including the character whose value corresponds to the magnitude of the "match" value is found. Control characters within the returned string (values of 31 or less) are converted to spaces (value 32).

If a match is not found using the given "match" value and no error is detected (such as end of file) then 255 characters will be returned. Below is a summary of values in the second right argument:

| "match" value | meaning |
|---|---|
| not given | read number of bytes given by left argument |
| 0 | read number of bytes given by left argument |
| 1 to 255 | read bytes until this "match" detected |
| -1 to -255 | read bytes until "match" of this magnitude, control characters converted to spaces |
| 256 | (see extension) |

The third right argument is for a variable in which the error code will be placed. If the third right argument is not given then no error code is given. If given, the variable must be writable (similar to the right argument of an assignment). If the variable is not writable then a "** VPL ** Invalid right argument to current operator" error is placed on the status line. If no error is detected in the sequential file operation then zero is placed "in" the third right argument. The most common errors are listed below:

| Error code | Meaning |
|---|---|
| 0 | No error |
| 53 | Illegal unit number for sequential file |
| 54 | Sequential file open for write only |
| 58 | Sequential file indicated is not open |
| 59 | End of file detected |
| 62 | (see extension) |

In the case of error code 59 (EOF) then as many characters as possible will be returned in the resultant string.

The result of this operator is a string which can vary in length between 0 (null string) and 255 characters. This string may contain ASCII control characters.

Examples:

```
 'TEXT.TMP'  SOPEN   19  ->  0      ;open a file called 'TEXT.TMP'
                                    ;and associate unit number 10
                                    ;position to 1st byte in file

 22 SREAD   <19,,#201>              ;read first 22 bytes in file
     -> 'This should be the fir'    ;resultant string is 22 bytes
                                    ;long
 #201      ->  0                    ;indicates no errors


 SREAD   <19,10,#201> =  #301       ;read up to next LF
     -> 'st line  '                 ;last character will be LF
                                    ;but displayed as space
 #301 PICK '-1' ASCII '-1' -> 10    ;last character is LF
 #301 PICK '-2' ASCII '-1' -> 13    ;second last could well be CR

 SREAD   <19,'-10',#201> = #301     ;read up to next LF
     -> 'This should be the second line  '
 #301 PICK '-1' ASCII '-1' -> 32    ;last character is now space
 #301 PICK '-2' ASCII '-1' -> 13    ;second last is now space

 SREAD   19       ->   'T'          ;read one byte
 SREAD   19       ->   'h'          ;read one byte
 SREAD   19       ->   'i'          ;read one byte
 SREAD   19       ->   's'          ;read one byte
 SREAD   19       ->   ' '          ;read one byte

 200 SREAD   <19,,#201>             ;read 200 bytes
     -> 'is the last line  '
 #201      ->  59                   ;end of file detected

 23  SPOS     19                    ;position to 23rd byte in file
 SREAD   <19,10,#201> =  #301       ;same effect as 2nd SREAD
     -> 'st line  '                 ;see above
 #301 PICK '-1' ASCII '-1' -> 10    ;last character is LF
 #301 PICK '-2' ASCII '-1' -> 13    ;second last could well be CR

 SCLOSE  19       ->  0             ;successful close of 'TEXT.TMP'
```

Extension:
----------
This operator can support CBASIC format.  It the has the form:

```
        SREAD   <un,256,err>     ->  str
```

This operator will read the next field in CBASIC format. The extra double quotes added by CBASIC format will be stripped so that the original contents of that field will be returned. If the field being read is the last in a document (indicated by following CR LF) then the special error code 62 is given. This error code indicates end of line detected and is only given when "match"=256. If the field being read is followed by the end of file then error code 59 is given.

This extension to the SREAD operator will decode fields (strings) which have been encoded by the PRSTR operator (when its type=-1).

Example:

```
'TEST.CBA'  SOPEN   19  ->  0       ;open a file
                                    ;assume it didn't exist
'Field 1'   PRSTR   <19,'-1'>
','         PRSTR   19              ;N.B. output commas
'Field,2'   PRSTR   <19,'-1'>
','         PRSTR   19
'Field"3'   PRSTR   <19,'-1'>
''          PRSTR   <19,1>          ;N.B. add trailing CR
                                    ;LF
```

This would be the code to write a single three field document in CBASIC format to a file. For this example the contents of these fields are stated explicitly. The contents of the file would be:

```
    Field 1,"Field,2","Field""3" <CR> <LF>  <EOF>
```

Now re-positioning to the beginning of the file:

```
        1   SPOS    19
```

And now read back the document:

```
SREAD   <19,256,#201>   ->  'Field 1'
#201    ->  0
SREAD   <19,256,#201>   ->  'Field,2'
#201    ->  0
SREAD   <19,256,#201>   ->  'Field"3'
#201    ->  62                  ;indicates end of document

SREAD   <19,256,#201>   ->  ''  ;null string returned
#201    ->  59                  ;indicates end of file

SCLOSE  19  ->  0               ;successful close of file
```

```
Operator Name:            STEP                     !
                          ----                     !
                                                   --------------------
Class:          Database

Arguments:      L   R1   R2

Result:         Yes,  error code

Summary:        ln      STEP    relative          ->  err
                ln      STEP    <absolute,1>      ->  err
```

Description:
------------

This operator will move the document pointer within the given occurrence list.

This operator may have a left argument. It may also have two right arguments (in a right argument list). All given arguments must be numbers. The result of this operator is an error code. Zero indicates no error.

In its simplest usage this operator is used without any arguments. In this case the document pointer in the current occurrence list is moved one position forward (i.e. towards the end of the list). If the document pointer was previously at the last position in the list then it is moved to the first position in the list. If the current occurrence list was empty then this operator would have no effect.

If another occurrence list (apart from the current list) is to have its document pointer moved then it can be identified by number (1-100) by giving a left argument.

If the first element of the right argument list is given then its is taken as the number of places to move the document pointer in the given occurrence list. This movement can be done two ways: the default is relative to the current document position and the other way is absolute. If absolute positioning is required then this would be indicated by giving 1 as the second element of the right argument list.

Relative positioning of the document pointer will mean that positive or negative numbers (rounded to integers if necessary) can be given. Positive numbers will move the document pointer towards the end of the given list. Negative numbers will move the document pointer towards the start of the given list. Offsets which are too large (or too negative) will result in the document pointer being set to the first document.

Absolute positioning of the document pointer will mean that the document pointer will be moved to the document whose number in the list corresponds to the given number. The counting is origin one. The term "rewind" a list is sometimes used for positioning the document pointer to the first document in a list. If an absolute position of zero, a negative number, or a number exceeding the number of documents in the given list is given then the document pointer will be set to the first document.

If the STEP operator is successful then the number zero is returned indicating no error. If no database was open when this operator was used then 47 is retu.rned.

Examples:

```
            STEP         ->  0   ;step to next document in
                                 ;current occurrence list
      101   STEP   1     ->  0   ;step to next document in
                                 ;current occurrence list
                                 ;(same action as first)

      33    STEP  <1,1>  ->  0   ;rewind list 33

      33    STEP  <45,1>->   0   ;move document pointer to the
                                 ;45th document in list 33

      33    STEP  '-4'   ->  0   ;move document pointer
                                 ;backwards 4 positions
                                 ;i.e. from 45th to 41st
                                 ;document
```

| | | | | |
|---|---|---|---|---|
| Operator Name: | STRIP | | | |
| Class: | String | | | |
| Arguments: | L* R | | | |
| Result: | Yes | | | |
| Summary: | str | STRIP | type | -> str |

## Description:

This operator will remove spaces (delimiters) from a string.

This operator must be given a left argument. If a right argument is given it must be a number. This operator returns a result.

The left argument is taken to be the string from which the spaces (delimiters) are to be stripped from.

The right argument is the type. If it is not given or zero then leading and trailing spaces (delimiters) are removed. If it is 1 then leading spaces (delimiters) are removed. If it is 2 then trailing spaces (delimiters) are removed. If it is 3 then all spaces (delimiters) are removed. To summarize:

| type | meaning |
|---|---|
| Not given | Remove leading and trailing spaces |
| 0 | Remove leading and trailing spaces |
| 1 | Remove leading spaces |
| 2 | Remove trailing spaces |
| 3 | Remove all spaces |

The result of this operator is a string from which spaces (delimiters) have been removed.

The system is initialized so that spaces are the only characters considered to be delimiters. Other characters can be used as delimiters by writing to system variables 524 and 525. Only those characters lying in these two system variables are considered as delimiters.

It may be useful to strip all "redundant" spaces (delimiters). By redundant is meant leading, trailing, and repeated embedded spaces (delimiters). This cannot be done by this operator but can be done by the PICKW operator. The PICKW operator should then have the form:

| code | action |
|---|---|
| str PICKW <,255> | Remove leading, trailing, and repeated embedded spaces |

Examples:

```
'  This is  a   test  ' STRIP       ->  'This is  a   test'
'  This is  a   test  ' STRIP   0   ->  'This is  a   test'
'  This is  a   test  ' STRIP   1   ->  'This is  a   test  '
'  This is  a   test  ' STRIP   2   ->  '  This is  a   test'
'  This is  a   test  ' STRIP   3   ->  'Thisisatest'

'  This is  a   test  ' PICKW <,255>->  'This is a test'
```

```
Operator Name:          SV                  !
                        --                  !
                                            ---------------------
```

Class: Status line

Arguments: L R1 R2 R3 R4 R5 R6 R7 R8 R9

Result: No

Summary: vf1 SV <vf2,vf3,vf4,vf5,vf6,vf7,vf8,vf9,vf10>

Description:
------------

This operator defines the verification of each status line field. This operator is passive. The status line fields only receive this verification after the next SL operator.

This operator can have a left argument. This operator can have up to 9 right arguments. This operator does not return a result.

The left argument represents the verification of first status line field (addressed as 901). The first right argument represents the verification of the second status line field (addressed as 902). The second right argument represents the verification of the third status line field (addressed as 903), and so on. If an argument is not given or a null string then a space is assumed. If more than one character is in the string the the first is taken. Up to 10 status line fields are allowed.

The verification associated with a field are the characters which will or will not be accepted into it from the keyboard. The verification is encoded as a single character. Field and character verification codes are outlined in the description of the operator FSTAT.

Examples:

```
 521         ->  80      ;thus 79 usable characters on status
                         ;line
10  SP  30               ;either field 901 will have 10 chars
                         ;and field 902 will have 30 chars or
                         ;they will have the proportion 1:3

'N' SV                   ; 901 will only accepts digits and
                         ;space while  902 will accept anything

0   SA  7                ; 901 will have attribute 0 while  902
                         ;will have attribute 7

2   SL                   ;now redefine the status line to have
                         ;two fields of length 10 and 30 chars.
                         ;respectively
```

```
Operator Name:            SW                !
                          --                !
                                            --------------------
```

Class: Status line

Arguments: L*

Result: No

Summary: str SW

Description:

This operator will place the given left argument on the status line. The previous contents of the status line will be replaced.

This operator must be given a left argument. It returns no result.

The previous contents of the status line will be replaced. This includes information set-up by previous status line operators such as SA, SP, SV, and SL. After this operator is executed there is only one status line field. That status line field (addressed as 901) takes up all the available space on the status line. It has the zero attribute associated with it and its verification is ' ' (all keys accepted).

Examples:

```
             -------------------------------------------------------
Status line  !Is this a test?                      Yes              !
before:      -------------------------------------------------------

            'Then nothing is stored, enter data then press F1'  SW

             -------------------------------------------------------
Status line  !Then nothing is stored, enter data then press F1      !
after:       -------------------------------------------------------

            #514      ->  1     ;indicates only one field on status
                                ;line now

            901 FSTAT  ->  32   ;indicates the above field has ' '
                                ;as verification

            ''  SW              ;this will blank the status line

             -------------------------------------------------------
Status line  !                                                      !
after:       -------------------------------------------------------
```

In order to set attributes on the status line then the following sequence could be used:

```
            'This should be flashing on the status line'    SW
            8    ATTR    901
```

```
Operator Name:          THEN                    ! see IF
                        ----                    !
                                                --------------------
Class:          Control

Arguments:      L*

Result:         No

Summary:        IF      cc      THEN
                    ....
                ELSE
                    ....
                ENDIF
```

Description:
------------
See the IF operator description.

```
Operator Name:            WHILE                ! see DO
                          -----                !
                                               --------------------
Class:          Control

Arguments:      L   R  (one or the other,  right takes precedence)

Result:         No

Summary:        DO      WHILE   (  cc   )
                   ....
                ENDDO
```

## Description:

See the DO operator description.

System variable number: 401 Database file name

Access: Read only

Initialized value: ''

Description:

This system variable contains the the current database file name or the last opened database file name. It should be used in conjunction with system variable 528 (database open flag) to find out if the given database file name is currently opened.

This variable contains the database name as given to the DBOPEN operator. If the system adds a default extension to this name then this extension is not shown in this variable.

Related operators: DBOPEN<br>DBCLOSE

---

System variable number: 402 Terminal name

Access: Read only

Initialized value: current terminal type name

Description:

This system variable contains the current terminal type name. This is selected by the VISUP or VISETUP module and cannot be altered in the VIPS module.

| | | |
|---|---|---|
| System variable number: | **403** | Current schematic name |
| Access: | Read only | |
| Initialized value: | '' | |

**Description:**

This system variable contains the name of the schematic currently on the screen. If a user schematic is on the screen then this name starts with an alphabetic character. If a system schematic is on the screen then this name will be decodable as a number.

A schematic name can be up to 20 characters long.

| | | |
|---|---|---|
| Related operators: | CREATE<br>SEARCH<br>SCHEMA | |
| Related system variables: | 431 | Next schematic name |
| | 511 | Previous schematic name |
| | 501 | User schematic file name |
| | 502 | User schematic group name |

---

| | | |
|---|---|---|
| System variable number: | **404** | Current mode |
| Access: | Read only | |
| Initialized value: | 0 | |

**Description:**

This system variable contains the current mode number.

| | | |
|---|---|---|
| Related operators: | MODE | |
| Related system variables: | 430 | Next mode |
| | 505 | Previous mode |

System variable number: **405** Calculation precision

Access: Read only

Initialized value: Host system dependent

**Description:**

This system variable is purely informative, and contains the number of digits accuracy which can be expected from calculations. On most host systems this will be 15.

---

System variable number: **406** Division precision

Access: Read Write

Initialized value: 1

**Description:**

This system variable effects the way in which division is performed. When it is 1 (its initiaized value) then normal division is performed. When it is 0 then the result of the division is truncated towards zero to an integer.

Related operators: /

System variable number: 418 Type of execute

Access: Read Write

Initialized value: 1

Description:

This system variable effects the operation of the EXECUTE operator. It can have three values: 0, 1, and 2. Their meanings are listed below:

| Value in 418 | Meaning |
|---|---|
| 0 | EXECUTE operators can be nested (subroutine action). Results are passed back. |
| 1 | the original EXECUTE operator is nested but other EXECUTE operators called from the original chain to one another. Result and/or pending operator are passed back. |
| 2 | all EXECUTE operators chain to one another. When the final EXECUTE operator is exhausted then the interpreter continues on the line following the original EXECUTE operator. |

Related operators: EXECUTE

Related system variables: 504 Executing flag

---

System variable number: 430 Next mode

Access: Read Write

Initialized value: 0

Description:

This system variable contains the number of the next mode. It is used when the MODE operator is not given any arguments. The valid mode numbers are 0 to 49.

Related operators: MODE

Related system variables: 404 Current mode
505 Previous mode

| | | |
|---|---|---|
| System variable number: | 431<br>--- | Next schematic name |
| Access: | Read Write | |
| Initialized value: | '' | |

Description:
------------

This system variable contains the name of the next schematic name. It is only used when the SCHEMA operator is not given a left argument.

A schematic name can be up to 20 characters long.

| | | |
|---|---|---|
| Related operators: | SCHEMA | |
| Related system variables: | 403 | Current schematic name |
| | 511 | Previous schematic name |
| | 501 | Current schematic file name |
| | 502 | Current schematic group name |

---

| | | |
|---|---|---|
| System variable number: | 447<br>--- | Fields in current schematic |
| Access: | Read only | |
| Initialized value: | 0 | |

Description:
------------

This system variable is set by the SCHEMA and SCHDEF operators to show the number of screen fields in the current schematic.

| | | |
|---|---|---|
| Related operators: | SCHEMA<br>SCHDEF | |
| Related system variables: | 448 | Current field number |

System variable number: 448 Current field number

Access: Read only

Initialized value: 1

Description:

This system variable is set to the current field number. When the VPL being executed is not directly related to the schematic it remains at its last value. It is set to 1 by the SCHEMA operator. It is set to 1 when the SUPER BEGIN process is commenced.

Related operators: SCHEMA

Related system variables: 447 Fields in current schematic

---

System variable number: 450 Processing rules on/off

Access: Read Write

Initialized value: 1 (on)

Description:

This system variable controls the execution of VPL in screen related field processes. When it has the value 1 then processes associated with screen related fields are executed.

When it has the value 0 then processes associated with screen related fields are not executed. In this case when control is passed to a screen field then input will be requested after which control will pass to the process indicated by the key that terminated the input (e.g. up arrow -> previous field, down arrow -> next field, F8 -> END process, etc.).

System variable number: 451 Rounding precision for "="

Access: Read Write

Initialized value: 70

Description:

This system variable contains the value that will be used for rounding of numbers given as the left argument of the assignment operator "=". Strings (even if they represent numbers) will not be affected by this variable. As well as rounding numbers, if necessary, trailing zeros will be added to numbers.

This system variable can take the values 0 to 20 and 70. 70 is the initialized value and means that rounding and the addition of extra zeros to the right of the decimal point will not be performed. The values 0 to 20 will perform rounding to the given number of decimals and, if necessary, add trailing zeros.

Related operators: =

| | | |
|---|---|---|
| Related system variables: | 522 | Field justification |
| | 540 | Rounding for non-integer numbers converted to strings |

---

System variable number: 453 Type of exit

Access: Read Write

Initialized value: 255

Description:

This system variable contains a code for the function key that caused an exit. The function keys F1 to F8 are represented by the numbers 1 to 8 respectively. The value 255 is used to indicate no function key has been pressed "recently". When the SUPER BEGIN process is commenced this variable is reset to 255. The user should only place the values 1 to 8, or 255 in this variable. If the EXIT operator is used without any arguments then the value in this variable is assumed.

N.B. The codes for function keys in system variable 509 (last key pressed) for F1 to F8 are -1 to -8 respectively (c.f. 1 to 8 for this variable).

Related operators: EXIT

Related system variables: 509 Last key pressed

| | | |
|---|---|---|
| System variable number: | 501<br>--- | Current user schematic file name |
| Access: | Read Write | |
| Initialized value: | '' | |

Description:
------------

This system variable contains the name of the current user schematic file. This variable is used when a second right argument is not given to the SCHEMA operator.

If the system adds a default extension to this name then this extension is not shown in this variable.

| | | |
|---|---|---|
| Related operators: | SCHEMA | |
| Related system variables: | 403 | Current schematic name |
| | 502 | Current user schematic group name |

---

| | | |
|---|---|---|
| System variable number: | 502<br>--- | Current user schematic group name |
| Access: | Read Write | |
| Initialized value: | '' | |

Description:
------------

This system variable contains the name of the current user schematic group. This variable is used when a first right argument is not given to the SCHEMA operator.

User schematic group names can be up to 20 characters long.

| | | |
|---|---|---|
| Related operators: | SCHEMA | |
| Related system variables: | 403 | Current schematic name |
| | 501 | Current user schematic file name |

System variable number: 503 Current procedure name

**Access:** Read only

**Initialized value:** ''

**Description:**

This system variable contains the name of the currently executing procedure. If a procedure is not currently being executed then it contains a null string.

Procedure names can be up to 20 characters long.

Related operators: LOOKPROC<br>
RETURN

---

System variable number: 504 Executing flag

Access: Read only

Initialized value: 0

Description:

This system variable will indicate whether an EXECUTE operator is currently interpreting its left argument.

| Value in 504 | Meaning |
|---|---|
| 0 | not within an EXECUTE operator |
| 1 | currently interpreting the left argument of an EXECUTE operator. |

Related operators: EXECUTE

Related system variables: 418 Type of execute

System variable number: 505 Previous mode

Access: Read only

Initialized value: 0

Description:

This system variable contains the number of the previous mode. This will be from 0 to 49.

Related operators: MODE

Related system variables: 404 Current mode<br>
430 Next mode

---

System variable number: 506 Current line number in process

Access: Read only

Initialized value: not initialized

Description:

This system variable contains the line number of the process (procedure) currently being interpreted. Lines are numbers from 1 to a maximum of 400.

Related operators: BRANCH

Related system variables: 507 Code for type of process

|  |  |  |
|---|---|---|
| System variable number: | 507<br>--- | Code for type of process |
| Access: | Read only | |
| Initialized value: | not initialized | |

Description:
------------

This system variable contains a value which indicates what type of process is currently being interpreted.

| Value in 507 | Meaning |
|---|---|
| -1 | BEGIN process being interpreted |
| -2 | END process being interpreted |
| -3 | SUPER BEGIN process being interpreted |
| -4 | SUPER END process being interpreted |
| -5 | screen field related process |

|  |  |  |
|---|---|---|
| Related operators: | GOTO<br>EXIT<br>MODE | |
| Related system variables: | 506 | Current line number in process |
| | 448 | Current screen field number |
| | 453 | Type of exit |
| | 502 | Current user schematic group name |

---

|  |  |  |
|---|---|---|
| System variable number: | 508<br>--- | Cursor position in last departed field |
| Access: | Read only | |
| Initialized value: | 1 | |

Description:
------------

This system variable contains the position of the cursor when it left the last INPUT operator (or the implied INPUT operator when a screen field has no associated active VPL code in its process).

The position is origin one and measured relative to the left hand end of the screen or status line field in question.

Related operators: INPUT

System variable number: 509 Code for last key pressed

Access: Read Write

Initialized value: 32

Description:

This system variable contains a code (value) for the last key pressed. This will refer to the last keypress accepted by the previous INPUT operator (or implied INPUT operator when a screen field has no active VPL code associated with its process).

VIPS maintains a 256 character type-ahead buffer (on top of anything provided by the host operating system). Characters waiting in this buffer will not be reflected in this variable.

When a function key is pressed then system variable 453 is also modified. Note that s.v. 453 gets a value from 1 to 8 corresponding to F1 to F8 (c.f. -1 to -8 with this system variable).

| Value in 509 | Corresponding key |
|---|---|
| -8 | F8 |
| -7 | F7 |
| -6 | F6 |
| -5 | F5 |
| -4 | F4 |
| -3 | F3 |
| -2 | F2 |
| -1 | F1 |
| 1 | Tab |
| 3 | Up arrow |
| 4 | Down arrow |
| 5 | Left arrow |
| 6 | Right arrow |
| 7 | Bell |
| 8 | Delete character |
| 9 | Insert character |
| 10 | Line delete |
| 11 | Line insert |
| 12 | Line erase |
| 13 | CR (Carriage Return) |
| 32-126 | Printable ASCII characters (internal coding) |
| 127 | Erase character |
| 128-255 | Printable ASCII characters (internal coding) |

Related operators: INPUT

Related system variables: 453 Type of exit
508 Cursor position in last departed field

**System variable number:** 510 Previous field number

**Access:** Read only

**Initialized value:** 0

**Description:**

This system variable contains the number of the previous screen field. The SCHEMA operator resets this variable to zero.

Related operators: SCHEMA<br>GOTO

| Related system variables: | | |
|---|---|---|
| | 448 | Current screen field |
| | 447 | Number of screen fields |

---

**System variable number:** 511 Previous schematic name

**Access:** Read only

**Initialized value:** ''

**Description:**

This system variable contains the name of the previous schematic.

A schematic name can be up to 20 characters long.

Related operators: SCHEMA

| Related system variables: | | |
|---|---|---|
| | 403 | Current schematic name |
| | 431 | Next schematic name |
| | 501 | Current user schematic file name |
| | 502 | Current user schematic group name |

| | | |
|---|---|---|
| System variable number: | 512<br>--- | Name of field associated with DOCDECOD operator |
| Access: | Read only | |
| Initialized value: | '' | |

Description:
------------

This system variable contains the field name associated with the line in a document last accessed by the DOCDECOD operator.

This variable can be up to 35 characters long (31 in field name, one ".", and 3 in extension).

This system variable is meant mainly for debugging. See the DOCDECOD operator description for more details.

| | | |
|---|---|---|
| Related operators: | DOCDECOD | |
| Related system variables: | 513 | Line number from GET, DOCDECOD operators |
| | 523 | Last line in document from DOCDECOD |

---

| | | |
|---|---|---|
| System variable number: | 513<br>--- | Line number and key information from GET and DOCDECOD operators |
| Access: | Read only | |
| Initialized value: | 0 | |

Description:
------------

This system variable will return a positive number if the last field accessed by either a GET or a DOCDECOD operator was a key field. This system variable will return a negative number if the last field accessed by either a GET or a DOCDECOD operator was a non-key field.

The absolute value of this variable contains the line number of the field fetched by the most recent GET or DOCDECOD operator.

This system variable is meant mainly for debugging and is more fully explained in the DOCDECOD operator description.

| | | |
|---|---|---|
| Related operators: | DOCDECOD<br>GET | |
| Related system variables: | 512 | Field name from last DOCDECOD operator |
| | 523 | Last line in document from DOCDECOD |

|  |  |  |
|---|---|---|
| System variable number: | 514<br>--- | Number of fields on status line |
| Access: | Read only |  |
| Initialized value: | 0 |  |

Description:
------------

This system variable contains the number of fields currently defined on the status line.

This can be modified by the SL operator from 0 up to 10 fields. The SW operator causes 1 status line field to be defined so this variable is then set to 1.

If this variable has the value 5 then this implies that #901, #902, #903, #904, #905 exist while #906, #907, #908, #909, #910 do not exist.

|  |  |
|---|---|
| Related operators: | SL (SV SA SP)<br>SW |

-----------------------------------------------------------------------

|  |  |  |
|---|---|---|
| System variable number: | 515<br>--- | First use of SUPER BEGIN in this mode |
| Access: | Read only |  |
| Initialized value: | 1 |  |

Description:
------------

This system variable contains a value to indicate whether this usage of the SUPER BEGIN process is the first or otherwise in the current mode. At the beginning of each mode (i.e. after a MODE operator) this variable is set to 1. At the end of each SUPER BEGIN process it is reset to 0. Thus if this variable is tested in the second usage of the SUPER BEGIN process in a given mode it will be zero.

|  |  |  |
|---|---|---|
| Related operators: | MODE<br>GOTO |  |
| Related system variables: | 516 | First mode after initialization |

System variable number: 516 First mode after initialization

Access: Read only

Initialized value: 1

Description:

This system variable contains a number which indicates whether this is the first process of the first mode after the commencement (initialization) of the VIPS module.

If the current process is the first process of the first mode then this variable will be 1. If the current process is any other process than the first process of the first mode then this variable will be 0.

In practice the SUPER BEGIN process in mode 0 is the only process that can be first in the first mode.

Related system variables: 516 First use of SUPER BEGIN in this mode

---

System variable number: 517 Fuzz

Access: Read Write

Initialized value: ''

Description:

This system variable contains a value which represents the "fuzz" for arithmetic comparisons. The "fuzz" is a scaled comparison tolerance for deciding whether inexact representations of numbers are "equal" or otherwise.

The operators affected by this variable are EQ, NE, GE, GT, LE, and LT. These operators are only affected when one or both of the numbers being compared are non-integers. Integers have an exact representation within the system.

This variable can take values from 0 up to the number of digits precision claimed by the host system for double precision floating point calculations (DOUBLE PRECISION in FORTRAN). The formula used to calculate "equality" and examples are given in the description of the EQ operator.

The user is warned that setting the "fuzz" too close to the claimed number of digits precision runs the risk of having equalities fail which should not fail.

Related operators: EQ NE GE GT LE LT

System variable number: 518 Length of long hidden fields
---

Access: Read only

Initialized value: 80

Description:
------------

This system variable contains the number of character positions available in a long hidden field.

This variable is read only but can be modified indirectly by changing the number of long hidden fields (s.v. 519). 800 character positions are available for the long hidden fields. The default division is 10 fields of 80 characters each (numbered #301 to #310). No more then 20 long hidden field can be defined. Thus the length of long hidden fields can vary between 40 characters long and 255 characters long.

Related system variables: 519 Number of long hidden fields

---

System variable number: 519 Number of long hidden fields
---

Access: Read Write

Initialized value: 10

Description:
------------

This system variable contains the number of long hidden fields.

800 character positions are available for the long hidden fields. The default division is 10 fields of 80 characters each (numbered #301 to #310). No more then 20 long hidden fields can be defined. Thus the length of long hidden fields can vary between 40 characters and 255 characters long.

The minimum number of long fields is 3 (of length 255 characters each) and the maximum number is 20 (of length 40 characters each). Thus the value written to this variable should be in that range.

When a value is written to this variable then the previous contents of all long hidden fields is lost. They are all re-initialized to a null string.

Related system variables: 518 Length of long hidden fields

System variable number: 520 Number of lines on screen
---

Access: Read only

Initialized value: As set in current terminal handler

Description:

This system variable contains the number of lines on the screen being used by VISTA. This is defined when a terminal handler is defined in the VISETUP module. In some cases this will be all the available lines on the screen but it may be less. The status line will take up one of these lines (the lowest), leaving the rest for the schematic.

The SCHEMA operator "centres" schematics which have less lines than indicated by this variable. The HELP operator places the help schematic in the furthest corner (as defined by this variable and 521) from the cursor position.

Related operators: SCHEMA<br>HELP

Related system variables: 521 Number of columns on screen

---

System variable number: 521 Number of columns on screen
---

Access: Read only

Initialized value: as set in current terminal handler

Description:

This system variable contains the number of columns on the screen being used by VISTA. This is defined when a terminal handler is defined in the VISETUP module. In some cases this will be all the available columns on the screen but it may be less. The number of columns could also be stated as the number of characters positions allowed on a line. The number of character positions allowed for each line is the same with the exception of the status line which has one less character position.

The SCHEMA operator "centres" schematics which have less columns than indicated by this variable. The HELP operator places the help schematic in the furthest corner (as defined by this variable and 520) from the cursor position.

Related operators: SCHEMA<br>HELP

Related system variables: 520 Number of lines on screen

| | | |
|---|---|---|
| System variable number: | 522 | Field justification |
| Access: | Read Write | |
| Initialized value: | 0 | |

Description:

This system variable contains a code for field justification. This only affects the assignment operator ("=") when something is being placed in a screen or status line field.

This variable can have three values as noted below:

| Value in 522 | Meaning |
|---|---|
| 0 | Strings are left justified while numbers resulting from operators are right justified. |
| 1 | Everything is left justified |
| -1 | Everything is right justified. |

These are the default values which are read by the interpreter at the beginning of each VPL expression. They can be overridden for the rest of an expression by placing "R" or "L" between square brackets (e.g. [3R] would mean all assignments in the rest of the expression would be right justified with numbers rounded to 3 decimals).

| | | |
|---|---|---|
| Related operators: | = | |
| Related system variables: | 451 | Rounding precision for "=" |

---

| | | |
|---|---|---|
| System variable number: | 523 | Last line for DOCDECOD, or register not found for SEARCH |
| Access: | Read only | |
| Initialized value: | 0 | |

Description:

This system variable has two unrelated usages.

In conjunction with the SEARCH operator, 1 will be placed in this variable if there are no documents in the register just searched (or if no database was open), otherwise 0 is placed in this variable. See the SEARCH operator description for more details.

In conjunction with the DOCDECOD operator, 1 will be placed in this variable when the last line of a document is fecthed otherwise 0 will be placed there. See the DOCDECOD operator description.

| | | |
|---|---|---|
| Related operators: | SEARCH<br>DOCDECOD | |
| Related system variables: | 512<br>513 | Field name from DOCDECOD<br>Key/non key from DOCDECOD, GET |

System variable number: 524 First string delimiter

Access: Read Write

Initialized value: ' '

Description:

This system variable contains a single character. It is used as a word delimiter and in certain operations as an ignored character. Two separate string delimiters are allowed, the other being system variable 525. They are both initialized to the space character.

When a string is written to this variable the first character of that string is taken as the new string delimiter. If a null string is written to this variable then space is assumed. For more details see the descriptions of the related operators.

Related operators:

FOLD
LAST
PICKW
SELECT
SEQ (SNE)
SORT
STRIP

Related system variables: 525 Second string delimiter

---

System variable number: 525 Second string delimiter

Access: Read Write

Initialized value: ' '

Description:

This system variable contains a single character. It is used as a word delimiter and in certain operations as an ignored character. Two separate string delimiters are allowed, the other being system variable 524. They are both initialized to the space character.

When a string is written to this variable the first character of that string is taken as the new string delimiter. If a null string is written to this variable then space is assumed. For more details see the descriptions of the related operators.

Related operators:

FOLD
LAST
PICKW
SELECT
SEQ (SNE)
SORT
STRIP

Related system variables: 524 First string delimiter

System variable number: 526 Database profile delimiter

Access: Read Write

Initialized value: ';'

Description:

This system variable contains a single character. This character is used to delimit multiple keys in a field or a string. This character is significant in the left arguments of the PUT and SEARCH operators, and in the screen fields with a PUTDOC operator.

In the PUT and PUTDOC operators only strings (fields) identified as keys will be affected. Sub-strings separated by this delimiter will be stored in the database dictionary as separate keys.

In the SEARCH operator a search profile made up of several keys separated by this delimiter can be given. See the second extension to the SEARCH operator description for more details.

When a string is written to this variable the first character of that string is taken as the new string delimiter. If a null string is written to this variable then space is assumed. For more details see the descriptions of the related operators.

Related operators: SEARCH<br>
PUT<br>
PUTDOC

---

System variable number: 527 255 character hidden field

Access: Read Write

Initialized value: characters following "VIPS" in command line invocation

Description:

This system variable is similar to the hidden fields. Unlike the short hidden fields which are 16 characters long, the long hidden fields which can vary between 40 characters and 255 characters long, this variable is always 255 characters long. It may be useful for building up long lines to be sent to the printer or a sequential file.

Like the hidden fields the length of a string read from this variable is the same as the last stored string in that variable. Unlike the hidden fields numbers in some internal form are always converted to a string before they are stored in this variable.

This variable is initialized to those characters following "VIPS" during the host system's invocation of the VIPS module.

If one field fixed at 255 characters is not enough then the result parameter ( %0 ) can be used in any context in the same fashion as this variable.

| | | |
|---|---|---|
| System variable number: | 528 | Database open flag |
| Access: | Read only | |
| Initialized value: | 0 | |

Description:

This system variable contains a value which indicates whether a database is currently open and if so whether or not it has a checkpoint on.

| Value in 528 | Meaning |
|---|---|
| 0 | No database currently open |
| 1 | Database open that does not have a checkpoint |
| 2 | Database open that does have a checkpoint |

System variable 401 contains the name of the last opened database. If this variable indicates a database is open then the name in system variable 401 will be that of the currently open database.

| | | |
|---|---|---|
| Related operators: | DBOPEN<br>DBSAVE<br>DBCLOSE | |
| Related system variables: | 401 | Database name |

---

| | | |
|---|---|---|
| System variable number: | 529 | First mode number |
| Access: | Read Write | |
| Initialized value: | 0 | |

Description:

This system variable contains a number. Nominally this number should be the mode number that the VIPS module first commences in. Currently the VIPS module commences in mode zero. This variable is initialized to zero.

This variable could be used for other purposes.

System variable number: 530 First database file name

Access: Read Write

Initialized value: ''

Description:

This system variable contains a string of up to 20 characters. Nominally it is to be used for the name of the database first opened by the system. The VIPS module is commenced with no database open and this is reflected by the fact that this variable is initialized to a null string (and s.v. 528 is initialized to 0).

This variable could be used for other purposes.

---

System variable number: 531 First schematic file name

Access: Read Write

Initialized value: ''

Description:

This system variable contains a string of up to 20 characters. Nominally it is to be used for the name of the schematic file first opened by the system. The VIPS module is commenced with no schematic file open and this is reflected by the fact that this variable is initialized to a null string (and s.v. 501 is initialized to a null string).

This variable could be used for other purposes.

System variable number: 532 First schematic group name

Access: Read Write

Initialized value: ''

Description:

This system variable contains a string of up to 20 characters. Nominally it is to be used for the name of the schematic group first used by the system. The VIPS module is commenced with no schematic file open so there is no schematic group active and this is reflected by the fact that this variable is initialized to a null string (and s.v. 502 is initialized to a null string).

This variable could be used for other purposes.

---

System variable number: 533 First schematic name

Access: Read Write

Initialized value: ''

Description:

This system variable contains a string of up to 20 characters. Nominally it is to be used for the name of the schematic first used by the system. The VIPS module is commenced with no schematic active and this is reflected by the fact that this variable is initialized to a null string (and s.v. 403 is initialized to a null string).

This variable could be used for other purposes.

System variable number: 534 Template control

Access: Read Write

Initialized value: 0

Description:

This system variable contains a value which indicates the fashion in which the cursor will pass between the screen fields in the absence of VPL code to the contrary. The execution of code in screen related processes can be inhibited by system variable 450.

In the absence of VPL code the cursor will move to the first position of the first field when control is passed to a schematic. The cursor will pass between the fields in the sequence of the field numbers which has been defined in the SKJEMA module. This variable slightly modifies this action:

| Value in 534 | Meaning (in the absence of VPL code to the contrary) |
|---|---|
| 0 | "visit" all screen fields in a schematic |
| 1 | "visit" fields defined as key fields, don't stop at non-key fields |

Related system variables: 450 Processing rules on/off

System variable number: 535 Single step control

Access: Read Write

Initialized value: 0

Description:

This system variable contains a value which indicates whether the line of VPL code currently being interpreted should be displayed on the status line before it is executed. This system variable is meant for debugging VPL code.

| Value in 535 | Meaning |
|---|---|
| 0 | Don't display lines of VPL code before execution |
| 1 | Display lines of VPL associated with schematics |
| 2 | Display lines of VPL associated with schematics and procedures |
| 3 | Display all lines of VPL (i.e. associated with schematics, procedures, and mode control) |

If a single-stepping action is selected (1 to 3) then each line of VPL code is displayed before it is executed by the interpreter and the system waits for a response. The possibilities are described in the ERROR operator. Single-stepping will not display the error mark "??".

Related operators: ERROR

System variable number: 536 Print file unit
---

Access: Read Write

Initialized value: -1

Description:
------------

This system variable contains the unit number that the output from the PRINT, PRSTR, and PRCHAR operators will be sent to.

The initialized value is -1 and indicates the printer. The printer does not need to be "opened" by the SOPEN command.

All unit numbers used by the SOPEN command must be 10 or greater. This leaves the numbers 0 to 9 unassigned for the host operating system to use for devices. In some cases devices can be opened as files. Both devices accessible to these operators (PRINT etc.) and files are expected to be "byte oriented".

Related operators: PRINT
PRSTR
PRCHAR
SOPEN
SCLOSE
SPOS

------------------------------------------------------------------------

System variable number: 537 Override verification
---

Access: Read Write

Initialized value: 0

Description:
------------

This system variable contains a value that controls field verification.

If this variable is zero (its initialized value) then field verification will be as defined in the SKJEMA module for screen fields and as defined by the SV operator for status line fields. This field verification can be overridden by the third right argument of the INPUT operator.

In the same fashion that the INPUT operator can override field verification for one field, then this variable can override all field verification until reset to zero. The values for field verification (N.B. positive numbers greater than or equal to 32) are outlined in the extension of the INPUT operator description.

Related operators: INPUT
SV

System variable number: 538 Characters for YN (Yes/No) in current language

Access: Read only

Initialized value: As set in current language

Description:

This system variable contains a two character string. The first is the character that will be accepted as an abbreviation for "Yes" in the current language. The second is the character that will be accepted as an abbreviation for "No" in the current language.

If the current language is English then this variable will contain 'YN'; in French 'ON'; in German, Swedish, Danish, Norwegian 'JN' etc.

Languages can be defined for VISTA in the VISETUP module.

---

System variable number: 539 Store leading spaces in DB

Access: Read Write

Initialized value: 0

Description:

This system variable contains a value which indicates whether leading spaces should be stripped from strings before they are stored or not. Unless there is a good reason to the contrary, it is suggested that this variable be left at its initialized value of zero which means leading spaces will be stripped from strings before they are stored in the database. This will save space.

| Value in 539 | Meaning |
|---|---|
| 0 | Strip spaces from the front of strings (fields) to be stored in the database |
| 1 | Don't strip spaces from the front of strings (fields) to be stored in the database |

In all cases trailing spaces are always removed from strings (fields) being stored in the database. Embedded spaces are not altered.

Related operators: PUT<br>
PUTDOC

System variable number: 540 Rounding for non-integer numbers converted to strings

Access: Read Write

Initialized value: 70

Description:

This system variable contains the value which is used for rounding numbers held in an internal non-integer form (commonly called floating point) to a character representation of that number. This is an internal operation carried out when an operator that requires string type (e.g. PICK) is given the result of an arithmetic operation that yields a non-integer (e.g. 1 / 3).

The valid values for this variable are 0 to the number of digits precision claimed for floating point operations (DOUBLE PRECISION in FORTRAN), and 70. The value of 70 will convert the number in the most natural form (e.g. 3 / 2 will yield 1.5 while 1 / 3 will yield 0.333333... ). The value of 0 will round all internal non-integer numbers to integers. The value of 1 will yield 1 decimal, the value of 2 will yield 2 decimals, etc.

The initialized value of this variable (70) will be sufficient in the vast majority of cases.

Related operators: All operators requiring string type in their arguments

Related system variables: 451 Rounding precision for "="

## APPENDIX A: GLOSSARY OF TERMS

ARGUMENT Each operator can have up to 11 arguments. An operator can have no arguments, a left argument, a right argument, and a list of right arguments. An argument is either not given, or a constant, or a variable, or an expression (result of another operator).

ATTRIBUTE An attribute is something applied to a character or field on the screen to higlight it in some way. This may be reverse video, half intensity, flashing, colours, etc. The overall system supports 64 attributes. SKJEMA allows these to be selected on a character by character basis, VIPS allows them to be selected dynamically field by field.

BEGIN PROCESS Schematics have one process for each field and two extra processes. The extra processes are the BEGIN and the END. The BEGIN process is executed when control is passed to the schematic.

CHARACTER DELETE KEY A key which moves the contents of the current field (or line in screen and VPL editor) from the cursor position to its end once left. A space is added at the right hand end. This action occurs when there are non-blank characters to the right of the cursor. When the rest of the field (or line) is blank then the action is like the CHARACTER ERASE KEY.

CHARACTER ERASE KEY A key which blanks the character to the left of the cursor in the current field (or line in screen and VPL editor) and moves the cursor left one position.

CHARACTER INSERT KEY A key which moves the contents of the current field (or line in screen and VPL editor) from the cursor position to its end once right. A space is placed under the cursor position.

CLOSED APPLICATION This term refers to applications which "hide" the standard user interface of VIPS (mode 0 -control, mode 1 -input, mode 2 -search) from the end user.

COMMENT The ability to append a comment to any VPL line. A VPL line need not have anything else on it. A comment is indicated by a leading ";". Everything else on that line is ignored by the interpreter.

DATABASE This refers to a single physical file in VISTA. It can contain many registers. VISTA automatically maintains its dictionaries within this file. The file extension is ".VDB" .

DICTIONARY Within the database file the system maintains dictionaries containing keys. The maintenance of these keys is automatic and therefore does not involve the user or the application designer.

DOCUMENT A collection of related fields of information treated as a unit. When a document is created it is given a name. Documents with the same name form a register. Often the schematic name is used as a name for the document. The term "document" is equivalent to "record".

END PROCESS When an attempt is made by the user to exit from the screen fields by pressing a function key (F1-F8) then the END process associated with that schematic is executed.

EXECUTE a) To interpret VPL code in a process
b) To treat an argument (string) like a VPL expression and interpret it (see EXECUTE operator).

EXIT KEY The keys F1, F2, F3, F4, F5, F6, F7, and F8 are termed as exit keys or function keys.

EXPRESSION A collection of arguments and operators contained within one VPL line. An expression must contain neither a comment (prefixed by ";") nor an expression separator.

FIELD a) screen. Refers to the parts of the schematic into which the user (and VPL) can place information.
b) document. Data item which forms part of a document.

FIELD ATTRIBUTE In VIPS screen fields and status line fields can have their attributes changed dynamically (see ATTRIBUTE).

FIELD DESCRIPTOR Refers to a notation for identifying a field. This notation is made up of three parts: register name, searchable part of field name, and field name extension.

FIELD, KEY Refers to a field which is defined by SKJEMA as such. The contents of this field are stored in the database as a key(s). (A key field can contain more than one key.) Keys are searchable in the database (quickly).

FIELD NAME Similar to FIELD DESCRIPTOR. Possibly does not have leading register name which in many contexts is not required.

FIELD, NON-KEY Any field which is defined by SKJEMA which is not a key field. Non-keys can be selected from the database (slowly).

FIELD PROCESS Each schematic can have 0-200 fields. Associated with each screen field is a program called a field process. A field process may contain no VPL code. A field which has no active VPL code in its process will automatically be prompted for input.

FIELD VERIFICATION The ability to define in the SKJEMA program which keystrokes will be accepted into a field. The treatment of field overflow can be modified also.

FILE Probably refers to one of the datafiles accessed by the system: database (.VDB), user file (.VUS), or the system file (.VSF).

FUNCTION KEY The keys F1, F2, F3, F4, F5, F6, F7, and F8 are termed as function keys or exit keys.

GROUP Sub-division in the organization of the user file (.VUS). A group name can be up to 20 characters long.

INTERRUPT To stop the execution of VPL code. This can be done by pressing F8 twice while VPL is executing. The context can be examined and the current process aborted if necessary.

KEY This refers to part of a field, or a full field that is stored in a document and also in the database dictionary. Keys are used to search for documents. Virtually an unlimited number of keys can be held by the database.

LEFT ARGUMENT An operator can have a left argument which lies immediately to the left of the operator to which it refers. The result of an operator (if it has one) can be viewed as the left argument to the following operator in the current expression.

LINE a) document. For debugging purposes a document can be decomposed into its component lines where each line corresponds to a field. Fields are usually fetched by field descriptor but if this is not known...

b) on the screen. Sometimes it is convenient to talk about lines on the screen. In this case the top line is referred to as line 1. The status line is always the bottom line.

c) Screen editor. In SKJEMA when a schematic is being defined or edited then it is represented on the screen as a series of lines (e.g. 23 lines by 80 characters).

d) VPL. When VPL is being edited it is represented as lines in its own screen-based editor. More generally it is used in same the sense as STATEMENT.

LINE DELETE KEY The lines from the following line to the bottom line on the schematic (not the status line) are moved once up. The bottom line on the schematic is replaced by a blank line.

LINE ERASE KEY The current line is removed and replaced by a blank line.

LINE INSERT KEY The lines from the current line to the bottom line on the schematic are moved once down. The current line is then blank.

LIST A shortened form of occurrence list which is a list of pointers to documents.

MENU A type of schematic which implies the user has the chance to choose between various options for further action.

MODE The standard user interface of VIPS is sub-divided into modes. These modes allow the user to store and retrieve documents, view and edit them, and sort them and generate reports. The mode control is written in VPL thus everything that is offered is also available to the designer who wishes to make a closed application.

NUMBER In the context of VPL this is the result of an arithmetic operator or a numeric string.

NUMERIC STRING The database and VPL do not distinguish between characters and numbers. The database always stores strings. Any string that can be interpreted by VPL as a number is a numeric string.

OCCURRENCE The presence of a key in a field of a document is referred to as an occurrence.

OCCURRENCE LIST This is a list of document pointers. The system can hold up to 101 such lists and combine them logically, sort them, or select documents from them by given criteria.

OPERATOR This is the element which "does" something in VPL. Together with any arguments given, the execution of an operator will perform some action and perhaps return a result.

PRE-PROCESSOR This is a module in the SKJEMA program that is invoked after the user is finished with the VPL editor. It generates a new copy of the code which is compacted and slightly encoded. This speeds VPL execution. The pre-processor is not a compiler.

PROCEDURE A procedure is invoked from VPL code in exactly the same way as an operator. Procedures are written in VPL code. The definition of a procedure has the same structure as a VPL process.

PROCESS A process is a set of VPL lines associated with either a field within a schematic, or a schematic (BEGIN or END), or a mode (SUPER BEGIN or SUPER END).

REGISTER This is a sub-set of the documents in the database. A register contains all the documents created with the same name. It is possible to form an occurrence list of all documents in a register as well as sub-sets of it.

RESULT An operator in VPL may yield a result which can then be used as the left argument to following operators in the same expression.

RIGHT ARGUMENT An operator can have a right argument which lies immediately to the right of the operator to which it refers. An operator CANNOT have both a RIGHT ARGUMENT and a RIGHT ARGUMENT LIST.

RIGHT ARGUMENT LIST This term is used to describe one or more arguments enclosed between "<" and ">" which lie to the right of the operator to which they refer. An operator can have up to ten arguments in such a list. Each argument is separated from the next by a comma. Each argument can be itself a VPL expression.

SCHEMATIC This is the screen "template" into which the user enters data from the keyboard. Schematics are defined and modified by the program SKJEMA.

STATEMENT This refers to zero, one, or more VPL expressions on the same line followed optionally by a comment. In some contexts a "line" of VPL has the same meaning.

STATUS LINE Usually the bottom line on the screen or at least directly under the bottom line of a schematic. This is an independently controllable line. VPL can define, place messages, and accept input on the status line.

STRING An argument which contains explicit data. This is a sequence of characters. Strings are surrounded by quotes or double quotes. An exception exists whereby non-negative numeric strings can be written in VPL code without surrounding quotes or double quotes.

SUPER BEGIN This is a VPL process associated with mode control. Each mode has two processes: SUPER BEGIN and SUPER END. The SUPER BEGIN process is executed when a mode is first commenced.

SUPER END This is a VPL process associated with mode control. When a schematic has finished executing all its processes then control is passed to the SUPER END process.

SYSTEM FILE Special file referenced by both VIPS and SKJEMA containing information about messages (in various languages), terminal types, "system" schematics, VPL operator names, procedures, and the VPL mode control processes. The file extension is ".VSF".

USER FILE The file in which "user" schematics are found. Such a file is sub-divided into schematic groups which are further sub-divided into schematics. Thus it is possible for two schematics to have the same name in a schematic file as long as they are found in separate groups. Schematics in this file are defined and modified by the program SKJEMA, and they are referenced by the program VIPS. The user file has the extension ".VUS" .

VIPS Name of the run-time module in VISTA which can interact with databases and sequential files, display schematics generated by the SKJEMA module, and execute VPL code.

VPL The name of special purpose programming language associated with the VISTA. The name is an acronym for VISTA Programming Language. VPL is interpreted by the run-time module called VIPS.

## APPENDIX B: MESSAGES IN VIPS

```
 1:    ** VPL ** Unrecognizable statement
 2:    ** VPL ** Right argument already
 3:    ** VPL ** Work area full, simplify expression
 4:    ** VPL ** Stack full, simplify statement
 5:    ** VPL ** Illegal use of parenthesis
 6:    ** VPL ** Multiple results can only be followed by assignment
 7:    ** VPL ** End Of Process position illegal
 8:    ** VPL ** Verbal Filing System number:
 9:    ** VPL ** Left argument required for this operator
10:    ** VPL ** Argument expected
11:    ** VPL ** Field number (or system variable) out of range
12:    ** VPL ** Argument not found in work area
13:    ** VPL ** Named fields not supported yet
14:    ** VPL ** Arithmetic result exceeds 255 digits
15:    ** VPL ** Execution but no operator?
16:    ** VPL ** Too many DO loops in one line
17:    ** VPL ** Invalid right argument to current operator
18:    ** VPL ** Non-numeric argument to arithmetic operator
19:    ** VPL ** Attempt to divide by zero
20:    ** VPL ** (Internal error) Stack unexpectably empty
21:    ** VPL ** Unknown operator
22:    ** VPL ** Unknown procedure
23:    ** VPL ** Illegal inside procedure
24:    ** VPL ** String overflow (more than 256 characters)
25:    ** VPL ** System variable not defined
26:    ** VPL ** System variable is read-only
27:    ** VPL ** User schematic file not found
28:    ** VPL ** Group within user schematic file not found
29:    ** VPL ** Schematic not found
30:    ** VPL ** No such process
31:    ** VPL ** No such mode
32:    ** VPL ** Parameter not currently defined
33:    Schematic:
34:    Process:
35:    Line:
36:    SUPER BEGIN
37:    SUPER END
38:    BEGIN
39:    END
40:    Procedure:
41:    EXECUTING
42:    ** VPL ** Schematic file in incorrect format
43:    ** VPL ** Trying to put data in a deleted document
44:    ** VPL ** Trying to delete a non-existant document
45:    ** VPL ** Attempt to operate on non-existant document
46:    Press "F8" to ABORT current process, anything else to continue
47:    ** DATABASE NOT OPEN **
48:    Not a Vista-Verbal database
49:    Database left open ?
50:    ** VPL ** Error during sort
```

```
 51:   Exiting current process: continue processing?
 52:   ** SEQ ** File not found
 53:   ** SEQ ** Illegal unit number (must be >9 )
 54:   ** SEQ ** File open for write only
 55:   ** SEQ ** File open for read only
 56:   ** SEQ ** Unit number already in use
 57:   ** SEQ ** Too many files open (max. 4)
 58:   ** SEQ ** Unit number does not refer to open file/device
 59:   ** SEQ ** End of file detected
 60:   ** VPL ** Illegal file descriptor
 61:   ** SEQ ** File/device full
 62:   ** 62 **
 63:   ** 63 **
 64:   ** 64 **
 65:   ** 65 **
 66:   ** 66 **
 67:   ** 67 **
 68:
 69:
 70:   Hit 'space' to continue
 71:   Change schemaname to :
 72:   Fill in database name
 73:   Give password
 74:   Wrong password !
 75:   Checkpoint off/on
 76:   Fill in and hit F1 to execute, F8 = exit
 77:   Delete old database
 78:   Please wait ......
 79:   ** 79 **
 80:   ** 80 **
 81:   ** 81 **
 82:   ** 82 **
 83:   ** 83 **
 84:   ** 84 **
 85:   Pre-prosessing
 86:   Printing
 87:   Select.
 88:   F2=help
 89:    File:
 90:    Group:
 91:    Schema:
 92:   List len/pos
 93:   Mode:
 94:   ** 94 **
 95:   HELP-ENGLISH
 96:   ** 96 **
 97:    - Rolling out
 98:   Rolling in
 99:    F1=select, F7=select and show, F8=exit
100:   Data entry mode. Press F1 to store, or F8 to exit
101:   Press F1 to search, F7 to search and display, F8 to exit
102:   Searching all documents .......
103:   Searching .....................
104:   F1=store F2=help F3=delete F4=prev. F5=next F8=exit
```

```
105:    First list no. (1 - 101):
106:    Second list no. (1 - 101):
107:    Resultant list (1 - 101):
108:    Length before operation
109:    Length after operation
110:    Old list no. (1-101):
111:    New list no. (1-101):
112:    No current list
113:    List numbers: First list:
114:       Second list:
115:       Resultant list:
116:    Delete (y/N) ?
117:    Database already open:
118:    Database:
119:    No database open !
120:    Closing database:
121:    Securing database:
122:    Only working with checkpoint on !
123:    Forgetting ......
124:    List number:
125:     documents found
126:    Number
127:    of
128:    Current list empty
129:    Indicate field(s) for sorting.  F1 to sort, F8 to exit
130:    Sorting
131:     documents.......
132:    Give new schematic group name:
133:    Give new schematic filename:
```

## VERBAL FILING SYSTEM LOW LEVEL ERROR MESSAGES

The following are the error messages "recognized" by VFS. Low level error messages peculiar to a particular machine may be returned (negated) if one of the following is not appropriate.

```
-900 : Not a VFS file
-901 : Attempt to read outside file
-902 : Attempt to write to illegal block number
-903 : Attempt to read unwritten data
-905 : Logical-to-physical map error

-910 : Open flag on (non-checkpointed system closed in an
       irregular fashion)
-911 : File not found or in incorrect format

-996 : Referencing file with invalid unit number
-997 : Device full
-998 : Read/Write error on file unit
-999 : Unexpected end of file
```

## APPENDIX C: ATTRIBUTES

This appendix discusses the suggested assignments of screen attributes to the 64 attributes available in VISTA.

The 64 attributes available in VISTA are numbered 0 to 63.

Attribute 0 is taken to be the default attribute. This would be expected to be the normal configuration the terminal powers up in (e.g. green on black, or white on black).

If an attribute is selected that has not been assigned for that particular terminal then the default attribute is assumed.

The suggested attributes are:

| Attribute number | Effect |
|---|---|
| 0 | Default |
| 1 | Reverse |
| 2 | Underline |
| 3 | High intensity |
| 4 | Reverse video |
| 5 | High intensity, reverse video and underline |
| 6 | High intensity and underline |
| 7 | High intensity and reverse video |
| 8 | Blink |
| 9 | High intensity and blink |

If a colour screen is available and it has 6 colours:
red yellow blue magenta cyan green (white) (black)

| | |
|---|---|
| 10 | Black background, red |
| 11 | Black background, green |
| 12 | Black background, yellow |
| 13 | Black background, blue |
| 14 | Black background, magenta |
| 15 | Black background, cyan |
| 16 | Black background, white |
| 17 | Black background, blinking red |
| 18 | Black background, blinking green |
| 19 | Black background, blinking yellow |
| 20 | Black background, blinking blue |
| 21 | Black background, blinking magenta |
| 22 | Black background, blinking cyan |
| 23 | Black background, blinking white |
| 24 | White background, red |
| 25 | White background, green |
| 26 | White background, yellow |
| 27 | White background, blue |
| 28 | White background, magenta |
| 29 | White background, cyan |

| | |
|---|---|
| 30 | Red background, green |
| 31 | Red background, yellow |
| 32 | Red background, blue |
| 33 | Red background, magenta |
| 34 | Red background, cyan |
| 35 | Red background, white |
| 36 | Green background, red |
| 37 | Green background, yellow |
| 38 | Green background, blue |
| 39 | Green background, magenta |
| 40 | Green background, cyan |
| 41 | Green background, white |
| 42 | Yellow background, red |
| 43 | Yellow background, green |
| 44 | Yellow background, blue |
| 45 | Yellow background, magenta |
| 46 | Yellow background, cyan |
| 47 | Yellow background, white |
| 48 | Blue background, red |
| 49 | Blue background, green |
| 50 | Blue background, yellow |
| 51 | Blue background, magenta |
| 52 | Blue background, cyan |
| 53 | Blue background, white |
| 54-63 | No suggestion |

# APPENDIX D: SKJDOK

## PURPOSE:

To print user-defined schematics in VISTA environment, also provides facility for listing procedures, modes and system schematics.

## GENERAL:

Whenever the program prompt for input from the user, <CTRL/C> will either abort the program or go back to the previous level. A <CTRL/Z> will go back to the previous prompt (or the previous level if first prompt inside the current level). A <RETURN> will generally cause the prompt to be repeated.

If the possible answers are given on the prompt-line, the default value is given in upper case.

## INVOCATION:

There are two ways of invoking SKJDOK, which will start up assuming output to be to the printer.

1) SKJDOK
2) SKJDOK <filename>

In the first case, the program will prompt with

```
Schematic file:
```

indicating that it requires the name of a schematic file. A <CR> or <CTRL/C> at this stage will abort the program and return to the operating system.

If <CTRL/Z> is pressed there will be a new prompt:

```
List device (CON:/PRT:)?
```

where console (CON:) is the default value. After this, the prompt for 'Schematic file: ' will reappear.

The program will now attempt to open the file with the given name (default extension '.VUS'). If unsuccessfull, a message to this effect will appear, and the program will yet again ask for another filename.

SKJDOK supports two kind of files. A file containing user schematics '.VUS' and a VISTA system file (generally VISETUP.VSF - but any '.VSF'-file will be regarded as a system file).

USER-SCHEMATICS:

For a '.VUS'-file, the program will read, then list all the defined groups on the console, then ask which group the user wants.

Then all schematics within the given group will be sorted and displayed. The program then prompts with:

```
<P>-print screen, <C>-continue, <S>-select, <A>-abort ?
```

A 'P' will print the display of schematics on the printer, and the prompt will reappear.
An 'A' or <CTRL/Z> or <CTRL/C> will take the program back to the previous level, ie. the display of groups.
A 'C' will cause the program to accept all the schematics displayed, then print them one at a time.
A 'S' will cause the program to enter 'S'elect-mode, where it prompts the user for a schematic mask.
The mask uses the question mark '?' as a match for any character, and an asterisk '*' will cause the remaining characters of the mask (up to max length) to be filled with '?'s.
A <CR> will match all names.

In select mode, the program will display each schematic satisfying the select mask and ask the user if he wants it printed.
A <CTRL/Z> will cause the previous schematic to be redisplayed.

Whenever all schematics have been displayed, the program will save the names of those wanted, and the process of printing the schematic is started. Afterwards, the program will display all the groups and ask for a new one.

SYSTEM-FILE:

For a system-file ('.VSF'-file), the program will prompt with:

```
<S>-system menues, <P>-procedures, <M>-modes:
```

For system schematics & modes, there is a predefined no. of entries, causing the next prompt:

```
Give mode control no. (0..49) - empty line to terminate
                        or
Give system menu no. (1..49) - empty line to terminate
```

A sequence of numbers may now be entered in completely free format, on one or more lines. The termination of input is through an empty line.
Note that a -ve value has the effect of including all entries up to and including the given absolute value.
( so that -4,8,11 will mean 1,2,3,4,8 and 11)

For procedures, the following prompt is given:

```
Procedures - all/selected (A/s)?
```

If the answer is anything but 'S', all procedures will will be buffered for printing. If selection is chosen, the procedures will be sorted and displayed one at a time time together with a Yes/No-prompt.

When entries have been selected in this manner, the printing process will start and on exit the <S>/<P>/<M>-prompt will reappear.

## Keyword index

Keyword index