

Introduction.....	16
Main Routines.....	16
Subroutines.....	16
External Variable Blocks.....	16
Compiling and executing.....	17
VisualPQL Procedures.....	17
VisualPQL Syntax.....	19
Names.....	20
<i>Note</i>	21
Variables.....	22
Control Flow.....	25
Block Structures.....	25
IF and IFNOT.....	26
File I/O.....	27
Format Specifications.....	27
Database Access.....	28
Multiple Database Access.....	28
Case Blocks.....	28
Record Blocks.....	28
Table Access.....	30
ODBC.....	31
Graphical User Interface.....	32
PQLForms.....	32
Functions.....	34
Source Commands.....	35
VisualPQL Programs and Routines.....	36
RETRIEVAL, PROGRAM, SUBROUTINE.....	37
END.....	43
EXECUTE DBMS.....	44
EXECUTE SUBROUTINE.....	45
PERFORM PROCS.....	46
PQL ESCAPE.....	47
PQL EXIT DBMS.....	48
RETURN.....	49
Variables.....	50
Explicit Variable Declarations.....	51
Variable Lists.....	52
Arrays.....	53
REDEFINE ARRAY.....	54
SORT.....	54
Implicit Variables.....	56
CAT VARS.....	57
CONTROL VARS.....	58
DATE.....	59
<i>Caution</i>	59
INTEGER.....	61

MISSING VALUES	62
OBSERVATION VARS.....	63
REAL.....	64
SCALED VARS	65
STRING.....	66
TIME.....	67
<i>Caution</i>	67
VALID VALUES	68
VALUE LABELS.....	69
VAR LABEL	70
VAR RANGES	71
Assigning Values	72
Missing Values.....	73
Expressions	74
Database Variables.....	77
AUTOSET	78
COMPUTE	79
EVALUATE	80
GET VARS	81
PRESET	82
PUT VARS	84
RECODE.....	86
SET	89
EXTERNAL VARIABLE BLOCK.....	91
INCLUDE EXTERNAL VARIABLE BLOCK.....	92
DEFINE PROCEDURE VARIABLES	93
Control Flow	94
Blocks	94
Logical Conditions.....	96
Compound Conditions	97
Precedence	97
IF, IFNOT	99
JUMP	100
AFTER.....	101
BEGIN	102
EXIT	103
FOR.....	104
IFTHEN	106
LOOP	107
NEXT.....	108
UNTIL.....	109
WAIT	110
WHILE.....	111
SUBPROCEDURE.....	112
END SUBPROCEDURE.....	113
EXIT SUBPROCEDURE.....	113

EXECUTE SUBPROCEDURE.....	114
Reading and Writing Files	115
Filenames	115
OPEN	117
CLOSE.....	118
DELETE PROCEDURE FILE MEMBER.....	118
READ.....	118
Options.....	118
I/O List - Input Specification	118
REREAD.....	118
WRITE.....	118
I/O List - Output Specification	118
Database Access.....	118
Data availability during retrieval	118
PQL CONNECT DATABASE.....	118
PQL DISCONNECT DATABASE	118
DATABASE IS.....	118
END DATABASE IS	118
Case Processing Commands	118
CASE IS.....	118
DELETE CASE	118
END CASE	118
EXIT CASE	118
NEXT CASE.....	118
PREVIOUS CASE.....	118
PROCESS CASE.....	118
RESTORE CIR	118
Record Processing Commands	118
RECORD IS.....	118
DELETE RECORD	118
END RECORD	118
EXIT RECORD	118
NEXT RECORD.....	118
PREVIOUS RECORD.....	118
PROCESS REC	118
RESTORE REC	118
BACKUP	118
Processing Database Journals	118
PROCESS JOURNAL.....	118
JOURNAL RECORD IS.....	118
EXIT JOURNAL IS.....	118
EXIT PROCESS JOURNAL.....	118
NEXT PROCESS JOURNAL	118
NEXT PROCESS HEADER	118
Concurrent VisualPQL.....	118
LOOKUP	118

Accessing Tables	118
Row Processing Commands	118
Indexes	118
Commands in ROW blocks	118
OPEN TABLE	118
CLOSE TABLE	118
PQL CONNECT TABFILE	118
PQL DISCONNECT TABFILE	118
DELETE ROW	118
END ROW	118
EXIT ROW	118
NEXT ROW	118
PREVIOUS ROW	118
PROCESS ROWS	118
ROW IS	118
ODBC Client	118
CONNECT	118
Statement	118
Example	118
Graphical User Interface	118
WINDOW	118
WINDOW TITLE	118
WINDOW STATUS	118
WINDOW OUTPUT	118
WINDOW CLEAR	118
WINDOW SAVE	118
MENU	118
MENUITEM	118
MENSEP	118
TBARITEM	118
TBARSEP	118
INITIAL	118
MESSAGE	118
ENABLE MENUITEM DISABLE MENUITEM	118
CHECK MENUITEM UNCHECK MENUITEM	118
DISPLAY POPUP LIST	118
DIALOG	118
BORDERS	118
POSTYPE	118
BUTTON	118
CHECK	118
CHOICE	118
EDIT	118
COMBO	118
SPIN	118
IMAGE	118

LABEL.....	118
LINE	118
LIST	118
RADIO.....	118
SLIDER.....	118
PROGRESS	118
TEXT	118
TREE.....	118
Dialog Message Processing	118
Other Message Types.....	118
Dialog Control Commands	118
Other GUI Commands	118
DEDIT.....	118
INSERT DCONTROL.....	118
MODIFY DCONTROL.....	118
MODIFY DCONTROL FONT.....	118
REMOVE DCONTROL.....	118
SELECT DCONTROL	118
CLEAR DCONTROL.....	118
DEDIT MESSAGE.....	118
GRID.....	118
PQLForms Overview	118
Form Structure	118
Examples.....	118
Commands	118
Specifying VisualPQL in PQLForms	118
Using PQLForms	118
Field Editing Operations	118
Moving from screen to screen.....	118
Accessing Records and Rows	118
Updating a Record	118
Deleting Records.....	118
PQLForms General Clauses.....	118
Field Elements	118
Screen co-ordinates.....	118
[NO]DATA	118
[NO]LABELS	118
[NO]PROMPT	118
FONT	118
ERROR	118
FORM	118
SCREEN	118
Clauses	118
<i>Caution</i>	118
END SCREEN.....	118
PAGE.....	118

Clauses	118
FIELD	118
Clauses	118
CALL SCREEN	118
Clauses	118
FDISPLAY	118
ABUTTON	118
FBUTTON	118
GENERATE	118
Old Forms	118
PQLForms Error Messages	118
PQLServer	118
Buffers	118
CLEAR BUFFER	118
CREATE BUFFER	118
DELETE BUFFER	118
DELETE LINE IN BUFFER	118
EDIT BUFFER	118
GET LINE FROM BUFFER	118
INSERT LINE INTO BUFFER	118
PUT LINE TO BUFFER	118
DISPLAY WDL	118
Functions	118
List of Functions by Type	118
Trigonometric Functions	118
Mathematical Functions	118
Argument List Functions	118
Across Record Functions	118
Date and Time Functions	118
Global Functions	118
String Functions	118
Concurrent Functions	118
Miscellaneous Functions	118
Session Functions	118
Schema & Database Functions	118
Tabfile & Table Functions	118
Read/Write Functions	118
Dialog & Menu Functions	118
Dialog Editor	118
Client/Server Functions	118
Client Functions to administer Master	118
Client Functions to SQLServer/ODBC	118
Client Functions to PQLServer	118
PQLServer Functions	118
CGI Functions	118
List of Functions from A to Z	118

ABS.....	118
ACOS.....	118
AINT.....	118
ALOG.....	118
ALOG10.....	118
AMOD.....	118
APPDIR.....	118
ARCOS.....	118
ARRDIMN.....	118
ARRDIMST.....	118
ARRDIMSZ.....	118
ARCOS.....	118
ARSIN.....	118
ASIN.....	118
ATAN.....	118
ATTRNAME.....	118
BINDPARM.....	118
BRANCH.....	118
BRANCHD.....	118
BRANCHN.....	118
BUFNAME.....	118
CAPITAL.....	118
CASELOCK.....	118
CATINT.....	118
CATSTR.....	118
CDATE.....	118
CENTER.....	118
CGIBUFPN.....	118
CGIBUFSV.....	118
CGIVARPN.....	118
CGIVARSV.....	118
CHAR.....	118
CIRLOCK.....	118
CLIPAPP.....	118
CLIPGET.....	118
CLIPLINE.....	118
CLIPSET.....	118
CNT.....	118
CNTR.....	118
COLCOUNT.....	118
COLLABEL.....	118
COLLEN.....	118
COLNAME.....	118
COLTYPE.....	118
COLVALN.....	118

COLVALS.....	118
COMMA.....	118
COS.....	118
CRYPTKEY.....	118
COUNT.....	118
CTIME.....	118
CURDIR.....	118
DATEC.....	118
DATEMAP.....	118
DATET.....	118
DTTOTS.....	118
DBINDN.....	118
DBINDR.....	118
DBINDS.....	118
DBINDT.....	118
DBINDU.....	118
DBINDV.....	118
DBNAME.....	118
DBTYPE.....	118
DECRYPT.....	118
DEFFAM.....	118
DEFMEM.....	118
DEFTFN.....	118
DELDIR.....	118
DELFILE.....	118
DELMCLID.....	118
DGLOBAL.....	118
DITEM.....	118
DITEMCOL.....	118
DITEMH.....	118
DITEMID.....	118
DITEMROW.....	118
DITEMS.....	118
DITEMSEL.....	118
DITEMSID.....	118
DITEMTXT.....	118
DITEMTYP.....	118
DITEMW.....	118
DSN.....	118
EDIT.....	118
EDITNAME.....	118
ENCRYPT.....	118
ERROR.....	118
EXISTS.....	118
EXTERN.....	118

EXTERNS.....	118
EXP.....	118
FAMNAME.....	118
FEQ.....	118
FILECNT.....	118
FILEIN.....	118
FILEIS.....	118
FILEN.....	118
FILEOUT.....	118
FILESTAT.....	118
FILETIME.....	118
FILL.....	118
FINDITEM.....	118
FORMAT.....	118
FST.....	118
FSTR.....	118
GETBTNH.....	118
GETCHCH.....	118
GETCHKH.....	118
GETAKL.....	118
GETDFC.....	118
GETENV.....	118
GETERR.....	118
GETFLT.....	118
GETFOCUS.....	118
GETICLK.....	118
GETIFLT.....	118
GETIINT.....	118
GETINT.....	118
GETITXT.....	118
GETLBLH.....	118
GETLTXT.....	118
GETMAXCH.....	118
GETMCADD.....	118
GETMCHK.....	118
GETMCLID.....	118
GETMCLST.....	118
GETMCON.....	118
GETMDBN.....	118
GETMSEL.....	118
GETNITEM.....	118
GETNLINE.....	118
GETNSEL.....	118
GETPOS.....	118
GETRADH.....	118

GETRSTEP.....	118
GETTXT.....	118
GETTXTH.....	118
GLOBALN.....	118
GLOBALS.....	118
GLOBNAME.....	118
HELP.....	118
ICHAR.....	118
IDSTATUS.....	118
JOUFLAG.....	118
JULC.....	118
JULN.....	118
KEYNAME.....	118
KEYORDER.....	118
LEN.....	118
LG10.....	118
LINES.....	118
LN.....	118
LOG.....	118
LOG10.....	118
LOWER.....	118
LST.....	118
LSTR.....	118
MAKEDIR.....	118
MAX.....	118
MAXR.....	118
MAXRECS.....	118
MEAN.....	118
MEANR.....	118
MEMCOUNT.....	118
MEMINFO.....	118
MEMNAME.....	118
MIN.....	118
MINR.....	118
MISNUM.....	118
MISS.....	118
MISSING.....	118
MKEYSIZE.....	118
MOD.....	118
MRECSIZE.....	118
MSGTXT.....	118
NARG.....	118
NBRANCH.....	118
NEXTROW.....	118
NGET.....	118

NGLOBAL.....	118
NKEYS.....	118
NLABELS.....	118
NMAX.....	118
NMIN.....	118
NOFCASES.....	118
NOW.....	118
NPUT.....	118
NREAD.....	118
NRECS.....	118
NSUBDIR.....	118
NUMBR.....	118
NUMCASES.....	118
NUMRECS.....	118
NVALID.....	118
NVALLAB.....	118
NVARDOC.....	118
NVARS.....	118
NVARSC.....	118
NVVAL.....	118
ODBCCOLS.....	118
ODBCTABS.....	118
OUTFNAME.....	118
PACK.....	118
PAD.....	118
PAGELEN.....	118
PAGENO.....	118
PAGEWID.....	118
PATTERN.....	118
PFORMAT.....	118
PICTURE.....	118
PROCFILE.....	118
PROCNAME.....	118
PROGRESS.....	118
RACCESS.....	118
RAND.....	118
RANF.....	118
REAL4.....	118
RECDOC.....	118
RECDOCN.....	118
RECLEVEL.....	118
RELOCK.....	118
RECNAME.....	118
RECNUM.....	118
RECSIZE.....	118

REGEXP.....	118
REGREP.....	118
REPLACE.....	118
REVERSE.....	118
RKEYSIZE.....	118
RND.....	118
RNMFILE.....	118
ROWCOUNT.....	118
RRECSEC.....	118
RVARSEC.....	118
SARG.....	118
SBST.....	118
SCROLLAT.....	118
SCROLLTO.....	118
SEEK.....	118
SERADMIN.....	118
SERADMIS.....	118
SEREXEC.....	118
SERGET.....	118
SERLINES.....	118
SERLOG.....	118
SERNOOUT.....	118
SERSEND.....	118
SERSENDB.....	118
SERTEST.....	118
SERWRITE.....	118
SETAKL.....	118
SETDFC.....	118
SETDIR.....	118
SETPOS.....	118
SETRANGE.....	118
SETRC.....	118
SGET.....	118
SGLOBAL.....	118
SIGN.....	118
SIN.....	118
SIRUSER.....	118
SMAX.....	118
SMIN.....	118
SPREAD.....	118
SPUT.....	118
SQRT.....	118
SRCH.....	118
SREAD.....	118
SRST.....	118

STATTYPE.....	118
STDEV.....	118
STDEVR.....	118
STDNAME.....	118
SUBDIR.....	118
SUBSTR.....	118
SUM.....	118
SUMR.....	118
SVVAL.....	118
SYSTEM.....	118
TABINDN.....	118
TABINDS.....	118
TABINDT.....	118
TABINDU.....	118
TABINDV.....	118
TABNAME.....	118
TABRECS.....	118
TABVARS.....	118
TABVINFN.....	118
TABVINFS.....	118
TABVNAME.....	118
TABVRANG.....	118
TABVTYPE.....	118
TABVVALI.....	118
TABVVLAB.....	118
TABVVVAL.....	118
TAN.....	118
TANH.....	118
TFACCESS.....	118
TFATTR.....	118
TFCOUNT.....	118
TFFILE.....	118
TFGRNAME.....	118
TFGRPW.....	118
TFJNNAME.....	118
TFNAME.....	118
TF'TABS.....	118
TFUSNAME.....	118
TFUSPW.....	118
TIME.....	118
TIMEC.....	118
TIMEMAP.....	118
TODAY.....	118
TRIM.....	118
TRIML.....	118

TRIMLR.....	118
TRIMR.....	118
TRUNC.....	118
TSTODT.....	118
TSTOTM.....	118
TWRITE.....	118
UPDLEVEL.....	118
UPGET.....	118
UPPER.....	118
UPSET.....	118
VALIDATE.....	118
VALLAB.....	118
VALLABSC.....	118
VALLABSN.....	118
VALLABSP.....	118
VALLABSV.....	118
VARDOCSN.....	118
VARGET.....	118
VARLAB.....	118
VARLABSC.....	118
VARNAME.....	118
VARNAMEC.....	118
VARLENG.....	118
VARPOSIT.....	118
VARPUT.....	118
VARTYPE.....	118
VFORMAT.....	118
VTYPE.....	118
WACCESS.....	118
WINCNT.....	118
WINLIN.....	118
WINMOVE.....	118
WINPOS.....	118
WINSELL.....	118
WINSELP.....	118
WRECSEC.....	118
WVARSEC.....	118
YESNO.....	118
The VisualPQL Debugger.....	118
Error Messages.....	118
Overview to the VisualPQL GUI Debugger.....	118
Source	118
Data	118
Stack.....	118
How to debug a program.....	118

Introduction

VisualPQL (Visual *Procedural Query Language*) is a structured programming and application development language that allows you to develop complete applications. You have full control of your application logic together with numerous high-level, non-procedural features and constructs for accessing data in a SIR/XS relational database.

The source of a VisualPQL program is a set of commands that are typically either a member (with a :T suffix) in the procedure file or a text file. Use a text editor in SIR/XS to create and modify programs.

Main Routines

A program has a single main routine that may optionally reference subroutines. A main routine can begin with a PROGRAM command. A main routine that accesses any database begins with a RETRIEVAL command. The main routine ends with an END PROGRAM or END RETRIEVAL command. For example, a simple program might be:

```
PROGRAM
WRITE 'Hello World'
END PROGRAM
```

In addition to programs and subroutines, VisualPQL provides a system for the creation and maintenance of data entry screens known as PQLForms. A PQLForms main routine begins with a FORM command and ends with an END FORM command.

Main routines can be re-compiled each time they are run, or can be compiled and saved as an executable member with an :E suffix. A program may use input parameters that are specified at run time.

Subroutines

A subroutine is an independent routine that is executed from the main routine or from another subroutine. Subroutines begin with the SUBROUTINE command and end with the END SUBROUTINE command.

Subroutines must be pre-compiled before they are referenced in an executing program. When a subroutine is compiled, it creates a member with an :O suffix.

A PQLForm can be saved as a subroutine.

External Variable Blocks

An external variable block is a block of variables used by several routines. An external variable block begins with the `EXTERNAL VARIABLE BLOCK` command and ends with the `END EXTERNAL VARIABLE BLOCK` command. External variable blocks must be precompiled before they are referenced in a compilation or execution of a program. When an external variable block is compiled, it creates a member with a `:V` suffix.

The five commands, `PROGRAM`, `RETRIEVAL`, `FORM`, `SUBROUTINE` and `EXTERNAL VARIABLE BLOCK` begin a routine. The corresponding `END` commands end the routine. All other VisualPQL commands must be included in one of these routines.

Compiling and executing

To compile and execute the program from the menu system, select `RUN` from the Member or File dialogs.

When a program is run, it executes, creates any files or other outputs and displays any messages or interactive output in the scrolled output window. When the run is complete, the next command is read from the input source. If there are no more commands, control is returned to the user.

Options on the `RETRIEVAL`, `PROGRAM` and `FORM` commands determine whether routines are compiled, saved or executed.

Running a program with no options on the initial command, compiles it and then executes it. The `NOEXECUTE` option compiles without executing. The `SAVE` option, together with the name of a member with an `:E` suffix, saves the executable version. Specify the `REPLACE` option to allow an existing member of the same name to be overwritten.

VisualPQL Procedures

Main Program and Retrieval routines may use one or more VisualPQL Procedures. The program creates the data for the procedure with the `PERFORM PROCS` command. The procedure specifications determine how the data is then output. Multiple procedures can be included in a single program so that one pass of the database produces multiple outputs.

Some procedures create output text files, others create files in specific formats that are directly useable by other software packages. All procedures, except the Full Report procedure, are single commands with option keywords.

The Procedure Table

The Procedure Table is the internal table that is built as the program processes the data and contains a set of data records. Each record in the table is made up of the *procedure variables* and contains a value for each variable. The default procedure variables are all

the program variables of the main routine excluding arrays. An alternate set of procedure variables can be specified with the `DEFINE PROCEDURE VARIABLES` command. Only variables available in the main routine can be included in the Procedure Table. Every time the `PERFORM PROCS` command is issued, a set of values is copied into the procedure table.

It is possible to specify a list of variables on the procedure definition itself. If this is not done, the procedure operates on all the variables in the procedure table.

VisualPQL Syntax

The syntax rules for VisualPQL are:

- Begin each new command in the first position on a line. There is no special character that indicates the end of a command.
- To continue a command on the next line, leave the first position blank and begin the continuation text in any other position. Commands can be continued for as many lines as required. There is no special character at the end of the line that indicates the command is being continued on the next line. You may split lines on blanks except where the command itself contains blanks e.g. `PROCESS CASES` - do not split command words across lines. Do not split commands, keywords, names or strings in quotes across physical lines. Otherwise the components of a command may be split as necessary. For example:
 - `WRITE ID`
 - `NAME`
 - `SALARY`
- To indent commands for readability, specify a period (.) in the first position of a line, followed by any number of spaces and then the command. For example:
 - `PROCESS CASES`
 - `. PROCESS REC EMPLOYEE`
 - `. WRITE ID NAME BIRTHDAY SALARY`
 - `. END REC`
 - `END CASE`
- If you wish to specify more than one command on a single, physical line, use a semi-colon (;) to separate the commands. Use of multiple commands on a line is not recommended since it makes the reading and modification of source more difficult.
- Any text on a line following a vertical bar (|) is treated as comments. Vertical bar comments are not continued to the next line. The `COMMENT` command specifies that the whole line is a comment.
- Commands and keywords cannot be abbreviated although there are synonyms for some commands. For example, `PROCESS REC` is a synonym for `PROCESS RECORD`; `C` is a synonym for `COMMENT`.
- A new command that does not begin with a valid command name or synonym, is taken to be an implicit `COMPUTE`. Be careful when relying on implicit compute statements and avoid using names for variables that conflict with commands. Even names that do not conflict now may conflict in subsequent releases so it is always safer to specify the command word `COMPUTE` if a program is going to continue to be run on a recurrent basis. Any variable name can be used in conjunction with the command; there are no specific reserved words. The following two statements are identical.

- ```
COMPUTE TOTAL = 10 + 15
TOTAL = 10 + 15
```

## Names

There are various types of entities in SIR/XS such as databases, records, variables, etc., each of which must have a *name*. Standard names do not begin with a number and may contain letters, numbers and the four characters \$, #, @, and \_. Standard names contain up to 32 characters and are translated to upper case.

You can also use non-standard names by enclosing the name in curly brackets ({}). A non-standard name can contain up to 30 characters and may use any character including blanks; no translations are performed on non-standard names.

When specifying commands, keywords and standard names, upper and lower case text are treated identically. For example the following two lines are identical:

```
COMPUTE A = B
compute a = b
```

The following preserves the lower case *a* for a name:

```
COMPUTE {a} = B
```

In an executing program, names are most frequently for variables. For example the expression:

```
COMPUTE A = B
```

This means take the contents of variable **B** and make these the contents of variable **A**.

When referring to other entities in a command, it may not always be as obvious. For example:

```
CLEAR BUFFER BUFNAME
```

The name `BUFNAME` could either be the name of a buffer or the name of a variable in the program that holds the name of the buffer. In fact, in the buffer manipulation commands, the name is a variable name or string expression **not** directly the buffer name. However, just as a command might be:

```
COMPUTE myname = 'Fred'
```

So a very simple string expression can be used to specify a buffer name e.g.

```
CLEAR BUFFER 'Previous Command'
```

Where a command uses expressions rather than directly naming an entity, it means that the name is not known until the program is run and, since many commands need to know names during compilation, this is not allowed everywhere. The syntax of each command specifies if this is allowed.

Some commands that normally require a name specified directly may also allow expressions where you have to enclose the expression in square brackets [] so that the

compiler can recognise that an expression is being used to derive the name. Again the syntax of each command specifies if this is allowed.

e.g.

```
EXECUTE SUBROUTINE { member_name | mem_name_exp_in_brackets }
```

So the following are identical:

```
EXECUTE SUBROUTINE OPENF
COMPUTE SUBNAME = 'OPENF'
EXECUTE SUBROUTINE [SUBNAME]
```

In particular, the `WRITE` command allows a list of variables to be written but expressions can be used by specifying them in square brackets, which can be very convenient and avoids the need for new intermediate variable names e.g.

```
WRITE [capital(name)]
```

### **Note**

*Be careful if using non-standard names in commands that allow either a variable name or a string in quotes as a name specification. If specifying a non-standard name in quotes, do not specify the curly brackets e.g.*

```
CLEAR BUFFER 'Previous Command' Not
CLEAR BUFFER '{Previous Command}'
```

*If you specify `CLEAR BUFFER {Previous Command}`, this looks for a local variable called `Previous Command` which is expected to contain the name of the buffer.*

*Similarly, be careful when manipulating non-standard names in a program. If your program is passing names to the software as strings at execution time, then it must pass the name without the curly brackets.*

*Also note that if a program gets back non-standard names from functions, they are not wrapped in curly brackets. If you are constructing commands or other processing where you would need curly brackets around any non-standard name, use the `STDNAME` function to do this.*

## Variables

Variables may be defined explicitly by command or implicitly by use. There are five types of simple local variables :

|         |                                                                                                                                                                                                                                                                      |
|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DATE    | Date variables are four byte integers. The value of a date integer is the number of days since the beginning of the Gregorian calendar. October 15, 1582 is day 1. The date format defines the input and output format. See date formats for a complete description. |
| INTEGER | Integer variables are 1, 2 or 4 byte integers. 4 bytes is the default. The value ranges are:<br>INTEGER*1 -128 to 123;<br>INTEGER*2 -32,768 to 32,763;<br>INTEGER*4 -2,147,483,648 to 2,147,483,643                                                                  |
| REAL    | Real variables are floating point numbers allowing a fractional component. REAL*4 (single precision) and REAL*8 (double precision) are allowed. Double precision is the default.                                                                                     |
| STRING  | String variables are strings of a specified length from 1 to 4094. If more characters than the declared string length are assigned to a variable, the string is truncated to the declared length.                                                                    |
| TIME    | Time variables are four byte integers. The value of a time variable is the number of seconds since midnight. The <i>time format</i> defines the input and output format. See time formats for a complete description.                                                |

To define a variable explicitly, specify the variable type followed by a list of variable names. For example:

```
INTEGER*4 month1 month2 month3
STRING*40 name1 surname
REAL*8 tot1 to tot9
DATE birthday ('DDIMMIYYYY')
TIME minutes ('MM')
```

To define a variable implicitly, assign a value to an undefined name. This creates the variable. Implicit numeric variables are REAL\*8. Implicit string variables are a default length that is normally 32 characters but this can be altered with the

- STRING LENGTH command.

### Dates and Times

If dates or times are assigned to another variable, the definition of that variable determines the value received. If the variable is numeric, it receives the numeric value; if a string, it receives the formatted date or time string. If the receiving variable is undefined, a numeric variable is implicitly created.

## Missing Values

Variables may contain Missing values. A variable has a missing value if it is undefined or allocated a value defined to be a missing value. If any variable in a computation contains missing values, then the result is missing values. (Other than those functions that specifically test the presence of missing values.)

## Declaring and using Arrays

Arrays can be defined. Each array is named and is one of the basic `INTEGER`, `REAL`, `STRING`, `DATE` or `TIME` variable types and has one or more *dimensions*. Array names cannot be the same as any of the VisualPQL function names. Specify the number of variables in each dimension. There is no limit to the number of dimensions nor the number of variables in any dimensions (other than memory or other machine limitations). An array must be explicitly declared by a command. For example:

```
INTEGER*4 ARRAY monthtot (12)
STRING*10 ARRAY sname (8)
REAL*8 ARRAY sum tsum(10,20)
DATE ARRAY fdays (12) ('DDDD')
TIME ARRAY minutes (24,60) ('MM')
```

Array dimensions normally start at 1 and proceed for the number of entries specified. An alternative start dimension can be specified where more natural or convenient using a 'from:to' syntax e.g.

```
INTEGER*4 ARRAY years (1900:2099)
```

This specifies an array with 200 entries that is referenced by values from 1900 thru to 2099.

Array dimensions can be redefined 'on the fly' with the `REDEFINE ARRAY` command. This allows you to grow, shrink or redimension any array programmatically.

Array entries can be sorted with the `SORT` command.

## Array Element Reference

In general, a subscripted array element can be used wherever an equivalent simple variable can be specified. A subscripted array element consists of the array name and the element locations for each dimension in parentheses. The subscript may be a constant or a numeric expression. For example:

```
COMPUTE MONTHTOT(12) = TOTAL
COMPUTE TOTAL = MONTHTOT(MONTH)
COMPUTE JAN01 = DAILYTOT(1,1)
```

The `SET` and `PRESET` commands can operate on whole arrays or on specific elements. For example:

```
SET MONTHTOT * (0) | whole array
SET MONTHTOT (1) (0) | specific element
```

## Control Flow

Program logic (the sequence in which commands are executed) is determined by how data matches specified logical conditions. Complex conditions can be specified by using connectors such as AND or OR. For example:

```
IFTHEN (A EQ B)
WHILE ((A EQ B) AND (C NE D))
IF (NOT E LT F)
```

## Block Structures

VisualPQL is primarily a block structured language. That is, the execution of a complete block of commands depends on the results of conditions. The various block structures are specified by a command that starts the block and an END command that ends the block. For example LOOP/END LOOP, IFTHEN/END IF.

Blocks may be nested inside other blocks. A block must be completely inside another block. Overlapping blocks are not allowed.

### Control commands in blocks

#### EXIT blocktype

An EXIT command stops execution of the block at that point and transfers control to the first command following the end of the block. An EXIT can be used in any block. A *blocktype* is normally specified on the EXIT command and this exits the innermost block of that type. An EXIT without a blocktype exits the innermost block.

#### NEXT blocktype

Many blocks are looping structures. That is, the commands within the block are executed repeatedly until some controlling condition is met. Commands such as WHILE iterate while a specific condition is true. Commands such as PROCESS REC retrieve a new record on each loop until the end of that set of records.

In looping blocks, the NEXT command transfers control to the first command in the block at the next iteration. A *blocktype* can be specified on the NEXT command and this transfers control to the innermost block of that type. A NEXT without a blocktype transfers control to the innermost looping block.

For example:

```
RETRIEVAL
PROCESS CASES ALL
. PROCESS RECORD EMPLOYEE
. IF (GENDER NE 1) NEXT RECORD
. GET VARS ALL
. PERFORM PROCS
. END PROCESS RECORD
END PROCESS CASE
REPORT
END RETRIEVAL
```

## **IF and IFNOT**

IF and IFNOT are conditional commands that are not block structured. When true, these commands execute command(s) that are specified as *continuations* of the IF, IFNOT command itself. The next new command (i.e. command starting in column 1) finishes the condition. If specifying multiple commands, separate each by a semi-colon (;).

Most commands can be specified with the IF command except:

- other IF, IFNOT commands. (Use the block structured IFTHEN if you need to nest conditions.)
- data definition commands
- block definition commands
- compiler directives

For example:

```
PROCESS CASES
. PROCESS RECORD EMPLOYEE
. IF (GENDER EQ 1) WRITE NAME
. END PROCESS RECORD
END PROCESS CASES
```

## File I/O

A program can `READ` and `WRITE` files.

Files can be opened and closed with the `OPEN` and `CLOSE` commands respectively. If a file is not opened or closed explicitly, the first occurrence of a `READ` or `WRITE` opens the file with default settings; reaching the end of the program closes the file.

### Binary Files

Normally files read or written by explicit reads and writes in VisualPQL are *text* files that contain readable characters together with end of record characters and can be viewed with a text editor. VisualPQL can also read and write *binary* files, that is files in internal non-text formats. Any file can be read as a binary file and the program is able to process the data exactly as it is on the file if the format is known. For example, a VisualPQL program could copy an image file or an executable or a library.

### Format Specifications

The `READ` command reads input from the file and assigns values read from the input to program variables. `READ` formats input data according to an input specification that is a list containing variable names and their formats. The formats can be fixed-field, free-field and can contain positional parameters.

`READ` is not a block control statement and simply executes without looping. In order to read through a complete file, it is necessary to enclose the `READ` in a looping block, typically a `WHILE` block that tests an I/O return code and finishes when the end of file is reached.

The `WRITE` command writes output formatted according to an output specification that is a list containing variable names and their formats. The formats can be fixed-field, free-field, or pictures, and can contain positional parameters. If an output format is not specified, defaults are used.

Typical input/output specifications might be:

```
write ('test.out') value1(f5.2) 2x code(A2) ',' value2(i*)
read ('test.out', iostat=status) input1(f5.4) 2x input2(i*) input3(i*)
```

## Database Access

Begin a program that accesses the database with the `RETRIEVAL` command. By default, this opens the database for read access only. Specify the `UPDATE` option on the `RETRIEVAL` command to open the database for write access.

### Multiple Database Access

The VisualPQL commands `PQL CONNECT DATABASE` and `PQL DISCONNECT DATABASE` connect and disconnect databases and set the default. A VisualPQL retrieval can reference more than one database. A retrieval can access a specified database with a `DATABASE IS` that starts a block of commands. Inside this block, all references are to variables in the new database. Any standard commands can be used in this block. When the block is exited, the original database is made current.

### Case Blocks

If the database is a Case Structured database, each case in the database has a *Common Information Record*, that is referred to as the *CIR*. The CIR contains the common variables including the *case identifier* that uniquely identifies each case.

Specify one of the *Case Processing* commands to access cases. A case processing command defines a block of commands, a *Case Block*. The block is terminated with an `END CASE` command. Within a case block, other commands may get values from or put values into common variables. As a case block is executed, a CIR is read into memory and other commands within the block use this. When the case block is exited or when a new CIR is called for, the record is replaced in the database if it has been modified and is overwritten with the new data. Each time a case is accessed with one of these commands, the CIR is available to other commands within the block.

Process cases using either the `PROCESS CASES` command that reads cases serially through the database or the `CASE IS` command that reads a specific case if it exists and can create a new case if it does not already exist. Use the `NEW CASE IS` and `OLD CASE IS` constructs to control processing depending on whether a case exists or not. `NEW CASE IS` creates a new case if one does not exist and skips the block if the case already exists. `OLD CASE IS` reads a specific case and skips the block if the case does not exist.

If a retrieval is run on a case structured database without a case processing command, an automatic `PROCESS CASES ALL` is generated.

### Record Blocks

Databases contain Record Types. Specify one of the *Record Processing* commands to access records. On case structured databases, record processing must be nested within a case block unless the record is accessed using a *secondary index*. A record processing command begins a *Record Block*. The `END RECORD` command ends a record block. Within a record block, other commands may get values from or put values into the variables in that record. As a record block is executed, a record is read into memory and other commands within the block use this. When the record block is exited or when a new record is read, the record is replaced in the database (if it has been modified) and is purged from memory.

Process records either using the `PROCESS RECORD` command that reads and selects records serially through a single case (on a case structured database), through the whole database or through a secondary index or using the `RECORD IS` command that reads a specific record if it exists and can create a record if it does not already exist. Use the `NEW RECORD IS` and `OLD RECORD IS` constructs to control processing depending on whether a record exists or not. `NEW RECORD IS` creates a new record if one does not exist and skips the block if the record already exists. `OLD RECORD IS` reads a specific record and skips the block if the record does not exist.

The record processing commands specify a record type and may specify a particular record or subset of records to retrieve. If there are no matching records, then the block of commands is skipped.

In the following example, the `WRITE` is not executed if there is no record type 2 for an employee and thus that employee does not appear in the output:

```
RETRIEVAL
PROCESS CASES ALL
OLD RECORD IS EMPLOYEE
. GET VARS ALL
. PROCESS RECORD 2
. GET VARS ALL
. WRITE ID NAME CURRPOS STARTSAL
. END PROCESS RECORD
END RECORD IS
END PROCESS CASE
END RETRIEVAL
```

## Table Access

A Table is analogous to a database record type and a *Row* is analogous to a record. These offer an alternative storage mechanism. Tables are stored on *Tabfiles*. Tables may be accessed from within either programs or retrievals. Multiple tables on multiple tabfiles may be accessed in a single program.

Table processing differs slightly from record processing as follows:

- Tables are maintained in creation sequence rather than in a key sequence.
- The only commands that deal directly with variables in a table are `GET VARS` and `PUT VARS`. This means that when retrieving a row of a table, the values of the variables must be moved into local variables with `GET VARS`. To update the values of variables in a table row, the local variables are moved into the table row with a `PUT VARS`.
- The `PROCESS ROWS` and `ROW IS` are analogous to the record commands and there are also the `OLD ROW IS` and `NEW ROW IS` constructs. Each of these commands defines a block of commands, a *row block*, that is terminated with `END ROW`.
- Tabfiles must be connected prior to the compilation of the program or subroutine either through the menu or the `CONNECT TABFILE` command. Tabfiles accessed during execution of a program or subroutine must be connected. The `PQL CONNECT TABFILE` may be used to connect tabfiles during execution.

## ODBC

Open DataBase Connectivity is a Windows based standard to allow communication between software from different vendors. Queries are done using SQL syntax. VisualPQL can set up ODBC connections, perform SQL queries, retrieve information on the results of the query and then retrieve the data.

SIR/XS allows other packages to access SIR/XS data through the SirSQLServer and VisualPQL can query this as any other ODBC source. VisualPQL can also query the SirSQLServer in a more direct fashion eliminating some of the ODBC overheads or allowing VisualPQL clients to operate on non-Windows platforms. Communication between client and server is machine-independent so allowing communication between any of the SIR/XS supported architectures providing these are networked using tcp/ip.

## Graphical User Interface

When SIR/XS starts, it invokes a main VisualPQL program that defines a main window and menu system. This program receives control when the user selects a lowest level menu item. It can deal directly with the requested function, call sub-routines, use sub-procedures or any VisualPQL construct and can call other VisualPQL programs and SIR/XS functions. The program can enable, disable, check or uncheck menu items as necessary.

The complete source code for the user interface is supplied with the system and the menus and dialogs can be used as examples for application development. You can modify the main menu program or create a customised version and run that when you start the system.

Once the system is running, any VisualPQL program can output information into the main window (such as title and status) and put text in the window using the normal `WRITE` command. Text output is scrolled and a line can be up to 4000 characters wide. Programs can also save, print or clear the main window.

VisualPQL programs can display and get information through dialogs. There are commands and functions to define a dialog and to interact with the user through the dialog.

There are commands that directly pop-up boxes that ask the user to respond, for example to display an error message or to ask for an OK or Cancel response. There are also commands that display a file browse box appropriate to the operating system when opening or saving files and commands that print files, displaying a print box to alter print specifications as necessary.

The Dialog Painter helps create VisualPQL dialogs. This gives a developer an interactive means of creating dialogs and of generating appropriate message processing blocks.

### PQLForms

PQLForms is an extension to VisualPQL that creates all the necessary logic for sets of linked, interactive dialogs for data entry, retrieval and update. A complete set of dialogs is a single VisualPQL routine known as a *Form*.

A Form can be created and maintained completely through the Forms Painter and this is the recommended way to develop forms.

There are additional commands that are only valid within a PQLForm. These define what variables are on each dialog, how they are displayed and edited, how the dialog is to look, and how dialogs are linked together. A PQLForm has built in buttons and associated

logic to allow the user to navigate through a set of records and to display, edit and insert data according to the database description. A developer can use all standard VisualPQL commands as necessary and these are executed at appropriate places in the form.

A PQLform is run in the same way as any other VisualPQL routine either directly or from a menu.

Once a form has been developed, it can be used by many people for data entry or for querying data.

### **Editor**

A program can invoke an editor for the user to enter text. Once the editor is invoked, control does not return to the program until the user exits the editor. The editor can use buffers to store data and there are VisualPQL commands to create, read and manipulate the contents of a buffer. This allows the use of buffers to enter and edit unlimited amounts of text with minimal programming. The user can choose to use a familiar standard editor or the SIR/XS internal editor (a simple gui style text dialog).

## Functions

Functions return a single numeric or string result derived from the arguments of the function. In general, the functions can appear in any string, arithmetic or logical expressions in a program. There are various types of functions such as Trigonometric, Mathematical, Date and Time, etc. For example, the function `CAPITAL (string)` capitalises the first alphabetic character of the string and the first alphabetic character following a blank. All other characters remain unedited.

```
PROGRAM
STRING * 50 NAME
NAME = 'this is the first day of the week'
NAME = CAPITAL(NAME)
WRITE NAME
END PROGRAM
```

The first character of every word in the string variable `NAME` is capitalised producing the following output:

```
This Is The First Day Of The Week
```

As another example, `FORMAT (X)` converts a number to a string in free-field format. The following gives the string '1.3':

```
XST = FORMAT(1.3)
```

There are a set of "across-records/rows" functions that compute statistics for a number of records or rows that may only appear in `PROCESS REC` or `PROCESS ROW` blocks. They use the values of a variable during the processing of a `PROCESS REC` or `PROCESS ROW` loop and produce a single value such as a total or an average. They ignore values that are missing or undefined.

## Source Commands

SIR/XS has a number of features that can assist when developing VisualPQL programs. These include features to:

- Document programs with comments;
- Include sets of code from various sources;
- Substitute Global variables;
- Generate code to compile;
- Specify conditional compilation rules for sets of code.

## VisualPQL Programs and Routines

Every VisualPQL program or subroutine starts with `RETRIEVAL`, `PROGRAM`, `SUBROUTINE`, or `FORM`. Retrievals and programs are main routines and a retrieval is allowed to access SIR/XS databases whereas a program is not.

Subroutines are independently compiled VisualPQL routines that are invoked with the `EXECUTE SUBROUTINE` command from other routines including other subroutines. Subroutines can `RETURN` to higher level routines.

PQLForms have a different structure because they contain predefined logic see PQLForms.

VisualPQL procedures can only be included in a main routine. Retrievals, programs and sub-routines use the `PERFORM PROCS` command to put data into the procedure table.

The general structure of main routines is:

```
RETRIEVAL or PROGRAM command
.....pql commands
.....pql commands
EXECUTE SUBROUTINE
.....pql commands
PERFORM PROCS
.....pql commands
PROCEDURE
PROCEDURE
.....
END RETRIEVAL
```

The general structure for a subroutine is:

```
SUBROUTINE
.....pql commands
.....pql commands
EXECUTE SUBROUTINE
.....pql commands
.....pql commands
RETURN
END SUBROUTINE
```

## RETRIEVAL, PROGRAM, SUBROUTINE

```

{RETRIEVAL |
PROGRAM |
SUBROUTINE name [(input_list)] }
 [CIRLOCK [=] lock_value]
 [CRWARN | NOCRWARN]
 [DEBUG [= name]]
 [ENDMSG | NOENDMSG]
 [EXECUTE | NOEXECUTE]
 [GET = memb_name:E]
 [LIBRARY = (family list)]
 [LOADING = num_lt_one]
 [LOADMAP]
 [LOCK [=] lock_value]
 [MISSCHAR = char]
 [NOARRAYMSG]
 [NOAUTOCASE]
 [PROGRESS]
 [RECLOCK [=] lock_value]
 [RETURNING (list)]
 [SAVE = memb_name:E [REPLACE][PUBLIC][PROCS | NOPROCS]]
 [SEED = num]
 [SHOWMISS]
 [STATIC | DYNAMIC]
 [SUMFILE = fileid]
 [TABFILE = tabfile_name]
 [TUPDATE [(list of tabfiles)]]
 [UPDATE]
 [UPSTAT | NOUPSTAT]
 [NODATABASE]
 [NOTUPDLOG]
 [VARMAP | NOVARMAP]

```

There are no required options on RETRIEVAL or PROGRAM.

The subroutine name is required on SUBROUTINE and is the name of the compiled subroutine. The name of the subroutine can be qualified with procedure file and family prefixes and passwords.

RETRIEVAL specifies the beginning of a main routine that accesses the default database. A retrieval opens the database files for read operations unless the UPDATE option is specified to open the database for write operations. All commands and procedures may be used in a retrieval. A retrieval is terminated with the END RETRIEVAL command.

PROGRAM specifies the beginning of a main routine that does not access database data. A program can use exactly the same features as a retrieval, except for the commands that

access database data. A program can access data in tabfiles and in external files. A program is terminated with the `END PROGRAM` command.

`SUBROUTINE` specifies the beginning of a subroutine that is invoked by other routines. All of the commands may be used in a subroutine but VisualPQL procedures cannot be specified. A subroutine can access data in databases, tabfiles and in external files. The code in a subroutine is logically separate from any other routine. A subroutine is terminated with the `END SUBROUTINE` command. The `RETURN` command explicitly returns control from a subroutine to the higher level routine. If a subroutine does not explicitly `RETURN`, control is passed back at the end of the routine. Subroutines may invoke other subroutines and may invoke themselves recursively.

## OPTIONS

The options on these commands specify compilation and execution conditions. Some options apply only to programs, some to retrievals and some to subroutines. Where an option does not apply to a particular type of routine, this is noted.

|                                                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|--------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>(input list)</code>                        | A subroutine may have input parameters. These are positional parameters corresponding to the <code>EXECUTE SUBROUTINE</code> list of parameters. The parameters are read-only and are local variables in the subroutine. These variables must be defined explicitly within the subroutine.                                                                                                                                                                |
| <code>CIRLOCK</code>                             | Sets the default lock type for concurrent operations for un-nested <code>PROCESS CASE</code> and <code>CASE IS</code> statements that do not explicitly specify locks. Nested <code>CASE</code> blocks inherit the lock type of the outer <code>CASE</code> block. The default lock type is <i>exclusive</i> ( <code>CIRLOCK = 6</code> ). (See Accessing the Database.)                                                                                  |
| <code>CRWARN</code>  <br><code>NOCRWARN</code>   | Causes a warning message for any variable that is created implicitly. The default is <code>NOCRWARN</code> .                                                                                                                                                                                                                                                                                                                                              |
| <code>DEBUG</code>                               | Stores information needed for the VisualPQL Debuggers with the compiled code. This includes the text of the program, pointers from the compiled code to the text line and the variable name table. The debug information is stored as a member subroutine with a default name of <code>SYSTEM.DEBUG:0</code> . This information can be stored elsewhere by specifying a member name on the <code>DEBUG</code> clause:<br><code>DEBUG [=membername]</code> |
| <code>ENDMSG</code>  <br><code>NOENDMSG</code>   | The default <code>ENDMSG</code> specifies that an 'END ASSUMED' warning message is issued for any implicit 'end of block' conditions. (See Block Structures.) <code>NOENDMSG</code> keyword suppresses warning message. You are advised NOT to specify <code>NOENDMSG</code> , as it can mask other problems in your program.                                                                                                                             |
| <code>EXECUTE</code>  <br><code>NOEXECUTE</code> | The default is <code>EXECUTE</code> , the routine begins execution when compilation is completed. <code>NOEXECUTE</code> compiles but does not execute the routine.                                                                                                                                                                                                                                                                                       |
| <code>GET</code>                                 | Loads and executes the executable member. Additional VisualPOL                                                                                                                                                                                                                                                                                                                                                                                            |

procedures can be specified for executable routines. For Example:

```
RETRIEVAL GET = WEEKLY.SALES
REPORT FILENAME = 'SALES.REP' /
 PRINT = NAME REGION NUMSALES TOTSALES
END REPORT
```

|            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LIBRARY    | Specifies a list of families that are searched when loading subroutines when the family name is not specified. The search for subroutines with unspecified family names begins in the default directory and proceeds through the list in the specified order. If the named member exists in more than one family, the first one found is used. For example:<br>RETRIEVAL LIBRARY=(STATSUBS PRNTSUBS TESTSUBS )                                                                                           |
| LOADING    | Specifies the loading factor used during a database update. The number is a percentage, expressed as a decimal number (e.g., .15 is 15%).                                                                                                                                                                                                                                                                                                                                                                |
| LOADMAP    | Specifies that a description (map) of routines loaded prior to execution is produced.                                                                                                                                                                                                                                                                                                                                                                                                                    |
| LOCK       | Defines the lock value for both CIR and records for concurrent operations. Use in place of defining both CIRLOCK and RECLOCK when these have the same value.                                                                                                                                                                                                                                                                                                                                             |
| MISSCHAR   | Specifies the character used when printing missing values. The default is asterisk (*). For example, to specify that a question mark is printed when the value of a variable is missing.<br>RETRIEVAL MISSCHAR = ?<br>To specify that a blank is used, specify:<br>RETRIEVAL MISSCHAR = /<br>Note: that the slash is necessary here to indicate a blank.                                                                                                                                                 |
| NOARRAYMSG | Suppresses the output of warning messages normally produced at compile time by references to array subscripts that do not correspond to the array definition. Specify when using REDEFINE ARRAY and references are expected that do not match the initial definition.                                                                                                                                                                                                                                    |
| NOAUTOCASE | In RETRIEVAL routines, this suppresses the generation of a PROCESS CASE command. If this is not specified, a PROCESS CASE is generated before the first executable command in the retrieval. If another case block is found later, the automatic one is removed. The compiler interprets all commands between the automatic PROCESS CASE and the first real case block as if they happened inside a case block.<br>NOAUTOCASE suppresses generation of an automatic PROCESS CASE command in a retrieval. |

NOAUTOCASE in a subroutine, allows a record block without a CASE block. If this is specified and the subroutine is not called from within a CASE block, execution of a RECORD block causes an execution error and the program terminates. Any references to variables are treated as if they are in a CASE block.

See Accessing the Database.

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| NODATABASE         | When compiling a subroutine, the compiler assumes that the database is accessed and the subroutine is referenced from a RETRIEVAL. The NODATABASE keyword specifies subroutines that may be used by a PROGRAM.                                                                                                                                                                                                                                                                                                                                                                                                          |
| PROGRESS           | When a retrieval is running, gives a visual indication of progress so far through the database. The system keeps track of progress by any PROCESS command. It takes the total number of cases or record type and increments a percentage as appropriate. This means that the displayed percentage may fluctuate when these commands are nested but a general indication of progress still applies.                                                                                                                                                                                                                      |
| RECLOCK            | Sets the default lock type for PROCESS REC and RECORD IS statements that do not explicitly specify locks for concurrent operations. The default record lock is <i>exclusive</i> ( RECLOCK = 6).                                                                                                                                                                                                                                                                                                                                                                                                                         |
| RETURNING ( list ) | A SUBROUTINE can return values to the EXECUTE SUBROUTINE command. At the time the subroutine returns control, these output variables are mapped positionally to the RETURNING ( list ) variables on the EXECUTE SUBROUTINE command. These variables must be defined explicitly within the subroutine and cannot be the same variables input to the subroutine.                                                                                                                                                                                                                                                          |
| SAVE               | Saves an executable (compiled) version of the program as a member. Using stored executables saves the overhead of repeated compilations. The member saved with this keyword is given a ':E', for 'executable', suffix. For example:<br>RETRIEVAL SAVE = WEEKLY.SALES:E                                                                                                                                                                                                                                                                                                                                                  |
| PROCS  <br>NOPROCS | Used with SAVE. PROCS is the default and specifies that any procedures (e.g. REPORT, SAS SAVE FILE, etc.) are saved along with the executable program. SIR SAVE FILE and WRITE RECORDS procedures cannot be saved.<br>NOPROCS specifies that the procedure specifications are not saved as part of the stored executable. This allows you to save an executable version of a program that builds a procedure table and to specify the procedures at run time. See the GET option. The following example stores an executable retrieval without the procedures.<br><br>RETRIEVAL SAVE = WEEKLY.SALES:E NOPROCS NOEXECUTE |
| PUBLIC             | Used with SAVE. Specifies that anyone may execute the saved member, but only those with family or member passwords may alter it. The following example saves an executable program as a member protected with passwords and makes it publicly available.<br>PROGRAM SAVE = APPLIC/MOON.MENUSYS:E/STARS PUBLIC                                                                                                                                                                                                                                                                                                           |
| REPLACE            | Used with SAVE and with SUBROUTINES (that are saved by default). Specifies that the member being saved replaces a member of the same name if it exists.<br>PROGRAM SAVE = WEEKLY.SALES:E REPLACE                                                                                                                                                                                                                                                                                                                                                                                                                        |

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SEED                | <p>SEED defines the seed value used by the random number generator for any sampling done by procedures, for any sampling done by <code>PROCESS CASE</code> or <code>PROCESS ROW</code> commands that do not specify a seed and for the <code>RAND</code> function if this does not reset the seed.</p> <p>This value is not saved on any saved executable. If you wish to use a non-standard seed, specify it on the command you use to execute the saved program or retrieval.</p> <p>The random number generator is initialised at the start of the execution and any sampling that generates a call or calls to it by the <code>RAND</code> function proceed through a set sequence of 'random' numbers depending on the seed. If the seed is reset, subsequent calls to the generator proceed through the new sequence of numbers.</p> |
| SHOWMISS            | <p>SHOWMISS specifies that a variable's original missing values are used when printing missing values. The default is asterisk (*) or the character specified by <code>MISSCHAR</code>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| STATIC  <br>DYNAMIC | <p>STATIC is the default subroutine loading mode. If static, then subroutines are loaded when the main routine is first executed and remain in memory. If local variables are altered in a statically loaded subroutine, the values are preserved from one invocation to the next. DYNAMIC specifies that subroutines are loaded each time they are executed and are unloaded when their execution completes.</p> <p>Note that this keyword applies to the default subroutine loading mode of the <i>main</i> routine. This can be overridden by the <code>EXECUTE SUBROUTINE</code> command. Specifying <code>STATIC   DYNAMIC</code> on a subroutine compilation does nothing.</p>                                                                                                                                                       |
| SUMFILE             | <p>Specifies the file where any database and tabfile update logs are written. (Specify <code>UPSTAT</code> to produce update logs.) If <code>SUMFILE</code> is not specified, any update logs are written to the standard output. <code>SUMFILE</code> affects only the update logs, other output is not affected.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| TABFILE             | <p>Specifies the default tabfile used on any <code>SAVE TABLE</code> procedures.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| TUPDATE             | <p>Specifies tabfiles opened in <code>WRITE</code> mode for update by the program. If this parameter is specified without any tabfile names, all referenced tabfiles are available for write (update). If specific tabfiles are listed, only those tabfiles are made available for update and any tabfile not in this list is opened as read only.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| UPDATE              | <p>Specifies that the database is attached for write (update). This keyword must be used to add, modify or delete from the default database. If the routine uses multiple databases, the <code>DATABASE IS</code> command specifies the update status for each database.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |

UPDATE can be specified for a subroutine. This enables the creation of a

self-contained RETRIEVAL UPDATE component.

|                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| UPSTAT               | Specifies that an update log is produced for database or tabfiles that are updated. NOUPSTAT is the default.                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| NOTUPDLOG            | Suppresses the tabfile part of the update log produced when UPSTAT is specified.                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| NOUPDLOG             | Suppresses the database part of the update log produced when UPSTAT is specified.                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| VARMAP  <br>NOVARMAP | <p>VARMAP specifies that the program variables are listed after compilation. The listing includes the routine name (main, subroutine or variable block name), variable names and data type.</p> <p><i>Proc Var</i> indicates that this variable is included in a summary table (see PERFORM PROCS).</p> <p>The VARMAP listing indicates variables explicitly declared before the first executable command. These variables are not affected by the AUTOSET command.</p> <p>NOVARMAP specifies that the listing is not produced and is the default.</p> |

## END

```
END RETRIEVAL |
END PROGRAM |
END SUBROUTINE
```

Indicates the end of a particular routine. These commands are synonyms so it is not strictly necessary to match the routine type and the type of `END` command, although it is good practice.

If there isn't an explicit `END` command, the end of a routine is indicated by the end of the input source or an `END TASK` or a new task indicated by a `TASK NAME` command or the start of another retrieval, program or subroutine.

## EXECUTE DBMS

```
EXECUTE DBMS string_exp
```

Suspends execution of this program and executes the specified SIR/XS command. This may call sets of commands and execute other programs or retrievals. When the input has finished processing, control is returned to this program at the following command.

Cannot be used inside a block that is accessing a database or tabfile e.g. a CASE, RECORD or ROW block but can be used in a PQLForms screen.

```
.....
EXECUTE DBMS 'RUN MYPROGS.REPORT'

COMPUTE COMSTR = 'RUN MYPROGS.REPORT'
EXECUTE DBMS COMSTR

FBUTTON ACTION (EXECUTE DBMS 'RUN MYPROGS.REPORT')
 PROMPT 'Run Report'
```

## EXECUTE SUBROUTINE

```
EXECUTE SUBROUTINE { member_name | mem_name_exp_in_brackets }
 [(list of expressions)]
 [RETURNING (list of variables)]
 [STATIC | DYNAMIC]
```

Executes the specified previously compiled subroutine, loading it if necessary. Specify either the explicit subroutine name or the name of a variable in square brackets that contains the subroutine name, for example:

```
EXECUTE SUBROUTINE [SUBNAME]
```

Specify an optional list of values to pass to the subroutine. This list may contain constants and expressions including variables. Variables referenced in this list must be defined in the calling routine. While individual array elements may be referenced and passed in this manner, a whole array cannot be passed to a subroutine. To pass a whole array, declare it as an external variable.

A subroutine may be executed at any point within another routine. Recursive executions are allowed and each copy maintains separate local subroutine variables.

**RETURNING**      The variables specified on the **RETURNING** clause are updated on return from the subroutine.

**STATIC** |      In the default **STATIC** mode, the subroutine is loaded into memory  
**DYNAMIC**      either when the calling routine is loaded or, if the subroutine name is specified with an expression, when the **EXECUTE SUBROUTINE** is first executed. The subroutine remains in memory until the program ends. Subroutine specific variables maintain their values from one invocation of the subroutine to the next, i.e. the variables are not automatically re-initialised with each execution of the subroutine.

In **DYNAMIC** mode the subroutine is loaded each time the **EXECUTE SUBROUTINE** is executed and unloaded when the **RETURN** statement is executed, releasing the memory used by the subroutine. If a subroutine is called dynamically, any subroutine called from within it is also dynamic unless it has previously been loaded statically.

## PERFORM PROCS

### PERFORM PROCS

The `PERFORM PROCS` command builds a set of data for the VisualPQL Procedures. A VisualPQL program that specifies one or more VisualPQL Procedures consists of two parts. The first part of the program retrieves data and puts it into the Procedure Table using the `PERFORM PROCS` command. The second part consists of the procedure specifications and executes after the first part has completed. Each procedure specifies how the data in the procedure table is output.

The procedure contains a set of data records. Each record in the table is made up of the procedure variables and contains a value for each variable. By default, the procedure variables are all the program variables in the main routine. The procedure variables can be specified with the `DEFINE PROCEDURE VARIABLES` command.

Each time a `PERFORM PROCS` command is executed a record with the current values of the procedure variables is added to the procedure table. A `PERFORM PROCS` command can appear in both the main routine and in subroutines. If a VisualPQL Procedure is specified and the `PERFORM PROCS` command is omitted from the main routine, a compilation error occurs.

## PQL ESCAPE

```
PQL ESCAPE string_exp [WAIT num_exp] [MINIMISE|MINIMIZE num_exp]
[RETURNING num_var]
```

Stops execution of this program and creates a sub-process that executes the operating system command specified in the string expression. The string expression is required and must immediately follow the command.

Specify the `WAIT` keyword followed by a numeric expression to control waiting for the sub-process to complete. If the expression is missing or resolves to a positive value, VisualPQL processing waits for the sub-process to complete; if the expression resolves to zero or a negative value, the VisualPQL processing continues without waiting.

Specify the keyword `MINIMISE` (optionally `MINIMIZE`) followed by a numeric expression to control visibility of the sub-process. If the expression resolves to a positive value, the sub-process runs minimised; if the expression is missing or resolves to zero or a negative value, the sub-process runs visibly. Note that if the sub-process is minimised and waits for completion, the SIR window is not refreshed until processing continues.

Specify the `RETURNING` keyword followed by a numeric variable name to get a return code. If the sub-process runs without waiting, zero is returned; if the sub-process fails to start, -1 is returned. Otherwise the termination status of the sub-process is returned (normally zero equates to success).

By default (no keywords), the command runs visibly and waits for the sub-process to complete before returning.

## **PQL EXIT DBMS**

PQL EXIT DBMS

Terminates the SIR/XS session. Use this command to exit completely without requiring further action from the user.

## RETURN

```
RETURN [NLEVELS n | TO subroutine_name]
```

Exits the current subroutine and is only allowed within a subroutine. Execution control is passed to the first statement following the `EXECUTE SUBROUTINE` command that called the current subroutine.

If execution reaches the end of the subroutine, control is returned automatically.

If `NLEVELS` is specified, the return goes back through `n` levels of sub-routine calls; if `TO` is specified the return goes back to the named subroutine. **Warning:** Using either the `NLEVELS` or `TO` options means that the subroutine is not independent and relies on knowledge as to how it is called and so these are *not* recommended practices.

# Variables

VisualPQL allows you to declare variables and define their characteristics; assign values to variables; create and use an External Variable block to pass data to subroutines; define the data passed to any VisualPQL Procedures.

Variables for use within a routine are referred to as *program*, *local* or *summary* variables as opposed to *database*, *table* or *external* variables.

Local variables can be explicitly defined with specific data declaration commands. If a command assigns a value to an undeclared variable, the variable is implicitly defined. Arrays can be defined and referenced using subscripts.

The `VARMAP` option prints a list of program variables.

Every variable has a name and a data type. Variables may have extended definitions such as value labels and missing values.

All of the definitions that can be given to database variables in schema definition may be given to variables in routines. The extended variable definitions can be explicitly defined or copied from the dictionary schema with the `GET VARS` command.

Variable declarations and extended definitions typically appear at the beginning of a routine. The declaration of a variable must precede any extended variable definition commands. Variable definitions must precede any reference to the variable whether the declaration is implicit or explicit. The code that defines the variable must physically precede the lines of code that reference the variable. Only define variables in subprocedures if the variable is only referenced in the subprocedure.

## Explicit Variable Declarations

Variables are defined explicitly with commands.

### Simple Variables

There are five types of simple local variables, `DATE`, `INTEGER`, `REAL`, `STRING`, `TIME`. The type can be followed by a length and a format for date and time. For example:

```
INTEGER*1 gender
REAL*8 total totall
STRING*25 name
DATE curdate ('DDIMMIYYYY')
```

### Extended Variable Definitions

Each variable may contain extended definitions for data validation and for default labels. The extended definitions include:

`VALUE LABELS` that defines descriptive labels for individual values of a variable.

`VAR LABEL` that defines a 78 character label for the variable that can be used in place of the variable name.

`MISSING VALUES` that defines specific values that are treated as missing in computations and statistical procedures.

`VALID VALUES` and `VAR RANGES` that defines values or ranges of values that are valid for this variable.

`SCALED VARS` that defines a scaling factor for an integer variable. The scaling factor is a power of ten, negative values specify decimal places, positive values specify tens, hundreds, etc.

## Variable Lists

Specify a list of variables with the `TO` keyword and use the same method to reference the variable. The order of local variables is determined by the order they are declared in the program. The order of database variables is determined by the order they are defined in the schema. Typically, programs declare variables whose names indicate a position within the list but this is not necessary. For example

```
INTEGER*1 VAR1 TO VAR10
SET VAR1 TO VAR10(2,4,6,8,10,12,14,16,18,20)
```

Variables can be referenced in a `TO` list format. A `TO` list specifies a beginning and ending variable. The following example declares seven variables and assigns a value to `NAME`, `ADDRESS`, `CITY`, `STATE` and `COUNTRY`. `REGION` and `ZIPCODE` are not affected by the `SET` command because they are not part of the implied list of variables.

```
STRING*25 REGION NAME ADDRESS
STRING*10 CITY STATE COUNTRY ZIPCODE
SET NAME TO COUNTRY ('Unknown')
```

An individual variable can be referenced by specifying an index value after the `TO` list that specifies the position of the variable in the list. Specify the index value immediate following the `TO` list specification, enclosed in parentheses. For example:

```
INTEGER*1 NUMA NUMB NUMC NUMD | declare variables
SET NUMA TO NUMD (11,12,13,14) | assign values to variables
COMPUTE NUMX = NUMA TO NUMD(3) | put 3rd var into NUMX
```

Variable list references may appear anywhere an expression may appear. The index value may be any numeric expression, including variable names, array references and more complex expressions. For example:

```
COMPUTE NUMA TO NUMD(3) = 32
IF(NUMA TO NUMD(3) EQ 32) WRITE 'O.K.'
```

*Note:* The variable reference is resolved and the variable moved to a temporary string or numeric variable before further computations are done. This means that variables such as categorical, date and time variables always return their numeric value when referenced in a `TO` list.

## Arrays

An array is a set of variables all of the same type. It has one or more *dimensions* that define the number of variables in the array. There is no internal limit to the number of dimensions nor the number of variables in any dimension, though the machine must be able to refer to enough memory for the array. You must explicitly declare arrays before use in another command. The general syntax to declare arrays is:

```
type [* size] ARRAY array_list (dimension [...]) [(format)]
```

|            |                                                                                                                                                                                                                                                                                                                                                                                  |
|------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| type       | Variable type: INTEGER, REAL, STRING, DATE or TIME                                                                                                                                                                                                                                                                                                                               |
| size       | Size of the integer, real or string variables.                                                                                                                                                                                                                                                                                                                                   |
| ARRAY      | Keyword specifying that arrays are being declared                                                                                                                                                                                                                                                                                                                                |
| array list | A list of the names of the arrays. Do not use VisualPQL function names. Do not use names of other variables referenced in the routine.                                                                                                                                                                                                                                           |
| dimension  | The number of occurrences in a dimension. You can specify a dimension either as a single number that is the number of entries in the array with references starting at 1, or as a start and finish pair of references separated by a colon ':' where the number of entries is the difference between these values plus 1. The number of entries must be a positive integer. e.g. |

```
INTEGER*4 ARRAY monthtot (12)
INTEGER*4 ARRAY yeartot (1990:2009)
```

You can declare arrays for any of the basic data types. Following is the syntax for each data type:

```
INTEGER [* {1 | 2 | 4}] ARRAY name_list (dimension [...])
STRING [* num_le_254] ARRAY name_list (dimension [...])
REAL [* {4 | 8 }] ARRAY name_list (dimension [...])
DATE ARRAY name_list (dimension [...]) ('date
format')
TIME ARRAY name_list (dimension [...]) ('time
format')
```

Some commands, such as SET and PRESET, can operate on whole arrays, in which case reference the array by name plus an asterisk \*. The extended variable commands refer to whole arrays. Most other commands operate on individual array elements.

Reference array elements by the array name and the element location within the array in parentheses, commonly called the array *subscript*. The subscript may be a numeric expression or constant. Specify a value for each dimension: e.g.

```
COMPUTE MONTHTOT(12) = TOTAL
COMPUTE TOTAL = MONTHTOT(MONTH)
COMPUTE JAN01 = DAILYTOT(1,1)
COMPUTE DEC31 = DAILYTOT(12,31)
```

## REDEFINE ARRAY

```
REDEFINE ARRAY array_name_exp (dim1, dim2,...)
```

The `REDEFINE ARRAY` command alters the dimensions of a locally defined array; arrays defined in `EXTERNAL VARIABLE` blocks cannot be redefined. The number of dimensions can be altered as well as the value of any dimension. The array can grow or shrink and existing values are mapped to the new dimensions. Any new values are set to missing.

Note that the array name is an expression, that is a string variable, expression or constant. To specify the name of the array to be redefined directly, simply enclose the name in quotation marks.

The VisualPQL compiler checks array subscript references where possible and warns if these do not match the array definition. If arrays are redefined, this checking may result in unwanted warnings. These can be suppressed with the `NOARRAYMSG` option. For example:

```
PROGRAM NOARRAYMSG
INTEGER*4 ARRAY NUM1 (50)
FOR I = 1,50
. COMPUTE NUM1 (I) = I
ROF
WRITE "Before redefine"
WRITE 'NUM1 (1) (50) Should be 1 50 ' NUM1(1) NUM1(50)
REDEFINE ARRAY 'NUM1' (50,2)
WRITE "After redefine of NUM1 to (50,2)"
WRITE 'NUM1 (1,1) (50,1) Should be 1 50 ' NUM1(1,1) NUM1(50,1)
WRITE 'NUM1 (1,2) (50,2) Should be * * ' NUM1(1,2) NUM1(50,2)
END PROGRAM
```

## SORT

```
SORT array_name
[BY key_array_varname]
[(n)]
[DESCENDING]
```

The `sort` command sorts the entries in an array. By default, all entries are sorted according to their values into ascending sequence. Multiple dimensions are sequenced as a single extended dimension e.g. If an array has two dimensions then entry (1,1) is first, (2,1) is second through to (n,1) that is followed by (1,2) etc. Note that the names of arrays specified in this command are specified directly, they are not expressions.

|                                |                                                                                                                                                                                                  |
|--------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>BY</code>                | One array can be sorted according to the values in a second array.                                                                                                                               |
| <code>key_array_varname</code> | The system matches the two arrays positionally and then sorts the original array according to the values in the named key array. If the arrays are different in size, the smaller value is used. |
| <code>(n)</code>               | The sort can be restricted to the first N entries.                                                                                                                                               |
| <code>DESCENDING</code>        | The sort can be into descending sequence.                                                                                                                                                        |

## Implicit Variables

Variables are implicitly defined in VisualPQL in two ways:

### Declaration by Assignment

If a value is assigned to a variable that does not exist, the variable is created. The data type of the new variable is taken from the context in which it is used. For example, assuming variable B already exists, the following implicitly defines variable A:

```
COMPUTE A = B
```

Numeric variables are declared as `REAL*w`. String variables are `STRING*w` where `w` is the current value of `STRING LENGTH` - default 32. Assigning dates and times creates numeric variables. Variables defined by assignment have no definitions other than the variable name, type and length.

The `CRWARN` option on the routine definition command issues a warning message during compilation whenever a variable is created by assignment.

### GET VARS

The `GET VARS` command implicitly declares new program variables copying the type and format, value labels and missing values from the schema definition of a database or table variable. `GET VARS` can copy individual variables or can use the keyword `ALL` that implicitly declares new program variables for all variables from a given record or table. A routine may access record variables directly in an appropriate block structure or may copy the data into an internal variable for further processing. For example:

```
PROCESS REC EMPLOYEE
GET VARS NAME
```

This creates a new implicit program variable `NAME`. This program variable is available outside the `PROCESS REC` block. The `GET VARS` command can copy database or table variables into explicitly defined local variables, in which case the definition of the variable is not affected.

## CAT VARS

```
CAT VARS varname ('value') varname ('value')
```

Specifies string variables that are held as categorical integers and defines the set of string values that can be held for the variable. The variables must first be explicitly defined as string variables and these cannot be arrays.

The values in the value list are each enclosed in single quote marks (') and the list for a variable is enclosed in parentheses. Specifications for multiple variables may be separated with a slash (/) for readability.

Internally, categorical variables are held as integers that are the position of the string in the value list. The variable may be treated either as a string or as a number depending on context. If the categorical variable is assigned to an undeclared variable, a numeric variable is created. If it is written without a format specification, the string is written. For example:

```
program varmap
string*1 str1 str2
cat vars str1 ('a','b','c')
compute str1 = 'a'
compute x = str1
compute str2 = str1
write str1 str2 x
end program
Variable list for Main Program
Variable Name Proc Type
STR1 Y CI*3
STR2 Y S*1

AUTOSET Variables
X Y R*8
Start program execution
a a 1
End program execution
```

## CONTROL VARS

```
CONTROL VARS varlist
```

Declares a list of variables or arrays that are *Control* variables for the `TABULATE` procedure.

The variables and arrays named on the command must be numeric and must have a `VAR RANGES` defined.

By default, variables that have `VALID VALUES` or `VALUE LABELS` are automatically control variables.

All other numeric variables are observation variables, that is variables with continuous values.

```
PROGRAM
INTEGER*2 var1
VAR RANGES var1 (1,30)
CONTROL VARS var1
....
PERFORM PROCS
....
TABULATE var1
```

## DATE

```
DATE varlist ('date format') [...]
```

Date variables are four byte integers. Dates are held as the number of days between the date and the start of the Gregorian calendar where October 15, 1582 is day 1. Dates can be represented as formatted strings and translated according to the date format.

When a date is assigned to another variable either the integer value or the equivalent formatted value is moved. If the assigned variable is numeric or undefined, the integer value is assigned. If the assigned variable is a string variable, the formatted string value is assigned.

The *date format* defines the default format. That is the format that is expected on input, is written on output and is assigned to string variables. See date formats for a complete description.

### **Caution**

When comparing dates and strings remember that the date is converted to a string using its default format then compared with the string. For example (assuming the date format for birthday is DDIMMIYYYY):

```
IF (BIRTHDAY lt '01 01 2007') SET AGEGROUP (2)
```

This is a comparison of strings does not classify dates correctly as any date string that has days of the month greater than 1 (i.e. '02 mm yyyy') is greater than the string '01 01 2007'.

If the date format for birthday is YYYYIMMIDD then:

```
IF (BIRTHDAY lt '2007/01/31') SET AGEGROUP (2)
```

This compares strings like '2007 12 31' with '2007/01/31'. Again this gives rise to errors because the former is less than the later because the character ' '(blank) has a lower ASCII value to the slash.

It is recommended that all date comparisons and processing are done with the numeric values:

```
IF (BIRTHDAY lt CDATE('2007/01/31','YYYY/MM/DD')) SET AGEGROUP (2)
```

For example, the following program declares and uses date variables. The program expects a string such as 'Jan 30, 2007' as input for BIRTHDAY and a string like '01-30-07' for VISDATE. The input strings '01 30, 2007' and 'JA/30/07' are also valid. Note that on

output, the default separator characters are spaces not slashes or dashes. Use the format options on the write to output other characters.

```
PROGRAM
DATE BIRTHDAY ('MMMiDDiiYYYY') /
 VISITDAT ('MMiDDiYY')
COMPUTE BIRTHDAY = 'Feb 26, 1970'
COMPUTE VISITDAT = '07/13/05'
WRITE 'Born on ' BIRTHDAY ('WWW DD/MMM/YYYY')
 'and visited on ' VISITDAT ('DD/MM/YYYY')
END PROGRAM
```

The M, D and Y strings cannot be split. The following is not allowed:

```
DATE BIRTHDAY ('YiMMiY')
```

## INTEGER

```
INTEGER [* { 1 | 2 | 4 }] varlist
```

The `INTEGER` command declares the listed variables as integers. Optionally specify the size of the variables as 1, 2 or 4 byte integers. If a size is not specified, the variables are 4 byte integers.

- `INTEGER*1` has values from -128 to 123
- `INTEGER*2` has values from -32,768 to 32,763
- `INTEGER*4` has values from -2,147,483,648 to 2,147,483,643

Example:

```
PROGRAM
INTEGER*1 SCORE1 TO SCORE5 SEX
INTEGER*2 MONTHSAL
INTEGER*4 YEARSAL
SET SCORE1 TO SCORE5 (0)
SET SEX (1)
SET MONTHSAL (2500)
COMPUTE YEARSAL = MONTHSAL * 12
WRITE MONTHSAL ('99,999') 2X YEARSAL ('999,999')
END PROGRAM
```

## MISSING VALUES

```
MISSING VALUES varlist (value [,value [,value]]) [/...]
```

MISSING VALUES specifies up to three values for a variable that are treated as *missing*. Missing values are excluded from statistical procedures and functions. A missing value is, by definition, a valid value for the variable and need not be re-specified.

The missing values can be constants or the keyword BLANK. If BLANK is not a missing value for a numeric variable, then blanks are stored as 0 (zero).

Missing values can be specified for string variables. Missing values for string, date and time variables are specified as strings. If the specified missing value matches the leftmost input characters, missing values are recorded.

Missing values can be specified for an array. Specify the array name in the command, not specific array elements.

For example, the following declares several variables and defines missing values for them. If the date 01/01/01 is assigned to TESTDATE, the value is treated as missing. If either a blank or the letters ZZ are assigned to STATE, they are considered missing. For the numeric array and numeric variables, the value 9 is treated as missing. If blanks are input with a READ command, they are treated as missing.

```
DATE TESTDATE ('MMiDDiYY')
STRING*2 STATE
INTEGER*1 ARRAY QUESTION (25)
INTEGER*1 MATHTEST READTEST
MISSING VALUES STATE (BLANK , 'ZZ') /
QUESTION MATHTEST READTEST (BLANK , 9) /
TESTDATE ('01/01/01')
```

## OBSERVATION VARS

OBSERVATION VARS varlist

Specifies variables and arrays that the `TABULATE` procedure use as Observation Variables. By default, variables that have Valid Values or Value Labels are Control Variables. `OBSERVATION VARS` changes these to Observation Variables.

## REAL

```
REAL [* { 4 | 8 }] varlist
```

The `REAL` command declares the listed variables as double precision, real, floating point numbers. `REAL*4` (single precision) and `REAL*8` (double precision) are also allowed.

When assigning a value to real variables, integers can be used without a decimal point.

## SCALED VARS

SCALED VARS varname (n)

SCALED declares the integer variables are scaled to power  $n$ .  $N$  is a positive or negative number representing the power of ten to which the variable is scaled.

If the variable has not been defined previously, this defines an `INTEGER*4` variable. To create a different length integer, define the variable before declaring the scaling factor. The full, unscaled number, including any decimal point, is used wherever this number is referenced.

## STRING

```
STRING [* number] varlist
```

`STRING` declares the listed variables as string of maximum length *number*. The maximum length of a string variable is **4094**. If a length is not specified, the default is the current setting of `STRING LENGTH`, that by default is thirty two characters. If more characters than the declared string length are assigned to a variable, the string is truncated to the declared length.

## TIME

```
TIME varlist ('time format') [...]
```

Time variables are four byte integers. Times are held as the number of seconds between the time and the previous midnight. Times can be represented as formatted strings and translated according to the time format.

When a time is assigned to another variable either the integer value or the equivalent formatted value is moved. If the assigned variable is numeric or undefined, the integer value is assigned. If the assigned variable is a string variable, the formatted string value is assigned.

The *time format* defines the default format. That is the format that is expected on input, is written on output and is assigned to string variables. See time formats for a complete description.

### Caution

When comparing times and strings remember that the time is converted to a string using its default format then compared with the string.

e.g. (assuming the time format for START is HH MM):

```
IF (START gt '09:00') SET LATE (1)
```

is a comparison of strings and the string '09 59' is less than the string '09:00' because the character ' '(blank) has a lower ASCII value to the colon.

In these cases it is best to convert the string to a number for the comparison:

```
IF (START gt CTIME('09:00','HH:MM')) SET LATE (1)
```

**H** A number of hours greater than 24 or minutes/seconds greater than 60 sets the variable to undefined. If hours, minutes or seconds are not input, they default to zero.

The following program declares and uses time variables:

```
PROGRAM
TIME STARTIME ENDTIME ('HHiMM')
COMPUTE STARTIME = SREAD('Enter Starting Time (HH:MM)')
COMPUTE ENDTIME = SREAD('Enter Quitting Time (HH:MM)')
COMPUTE TTIME = ENDTIME - STARTIME
WRITE 'You worked ' TTIME(TIME 'HH')
 ' hours and ' TTIME(TIME 'MM') ' minutes.'
END PROGRAM
```

## VALID VALUES

```
VALID VALUES varlist (value_list) [...]
```

Specifies the set of specific valid values a numeric variable can assume. If both `VAR RANGES` and `VALID VALUES` are defined for a variable, both specifications must be satisfied. Attempting to store a value in the variable that is not either a valid Missing Value or a Valid Value results in undefined. When a variable is updated during the running of a program, data validation takes place in the following order:

1. Missing Values
2. Valid Values
3. Variable Ranges

Examples:

```
INTEGER * 1 VAR1 TO VAR5 SCOREA SCOREB SCOREC
VALID VALUES VAR1 TO VAR5 (1 , 2) /
 SCOREA TO SCOREC (1 , 2 , 3 , 88 , 99)
```

## VALUE LABELS

```
VALUE LABELS varlist (value1) 'label text'
 [(value2) 'label text_2' [...]]
 [...]
```

Defines descriptive labels for individual values of a variable. Each label may be up to 78 characters long. Enclose labels in quotes. The keywords `UNDEFINED` and `BLANK` can be used as values and assign labels to undefined or blank missing values.

Specify value labels for multiple values of a single variable as one continuous command. If a number of variables have the same value labels, you can specify a list of variables, followed by the values and labels. If specifying value labels for an array, specify the array name not individual array elements. You can specify value labels for several variables on the same command.

For example, to declare a string variable, an integer variable and a 25 element array and define value labels for each:

```
PROGRAM
STRING*3 STATE
INTEGER*1 REGION
INTEGER*1 ARRAY QUESTION (25)
VALUE LABELS QUESTION (1) 'Yes'
 (2) 'No'
 REGION (1) 'North'
 (2) 'South'
 (3) 'East'
 (4) 'West'
 STATE ('NSW') 'New South Wales'
 ('QLD') 'Queensland'
 ('VIC') 'Victoria'
SET STATE REGION ('NSW',1)
SET QUESTION * (1)
COMPUTE STATEV = VALLAB(STATE)
COMPUTE REGIONV = VALLAB(REGION)
COMPUTE QUESTV = VALLAB(QUESTION(1))
WRITE STATEV REGIONV QUESTV
END PROGRAM
```

## VAR LABEL

```
VAR LABEL {variable | array } 'var label text'
```

VAR LABEL specifies a descriptive label for a variable. A variable label may be up to 78 characters in length and may be enclosed in quotes. Labels for multiple variables may be specified on a single command. The variable label can be retrieved during program execution with the VARLAB function.

Several VisualPQL Procedures automatically use a variable label if one is defined.

### Examples:

```
STRING*3 STATE
INTEGER*1 REGION
INTEGER*1 ARRAY QUESTION (25)
VAR LABEL STATE 'State of Residence'
 REGION 'Region of the State'
 QUESTION 'Survey Question'
```

## VAR RANGES

```
VAR RANGES {variable | array } (min_value , max_value) [/ . . .]
```

Specifies the range of values that a variable can have. Input values outside the specified range are set to undefined. If specific `VALID VALUES` are defined for a variable, do not specify `VAR RANGES`. If both are specified, the value must satisfy both specifications. When a variable is updated during the running of a VisualPQL program, data validation takes place in the following order:

1. Missing Values
2. Valid Values
3. Variable Ranges

### Examples:

```
INTEGER*1 YRSEDUC YRSWORK YRSPLAY
INTEGER*4 INCOME
DATE LASTDATE ('MMiDDiYY')
VAR RANGES YRSEDUC TO YRSPLAY (0,99) /
 INCOME (10000 , 90000) /
 LASTDATE ('01/01/2004' , '12/31/2005')
```

## Assigning Values

Values assigned to variables are specified as expressions. A variable may also be undefined or have a missing value. The commands that assign values explicitly to variables are:

`AUTOSET` resets implicitly defined local variables to *undefined*. It is typically used to ensure that values from a `GET VARS` in a `RECORD/ROW` block are not carried forward accidentally when the block is not executed due to a non-occurrence of that record for this particular instance. It also resets any variable explicitly declared after the start of the routine (the first executable command). It resets the values each time the command is executed.

`COMPUTE` sets a variable to a specified constant or expression value.

`EVALUATE` compiles small VisualPQL expressions during execution, allowing programs to accept expressions 'on the fly'.

`GET VARS` copies the definition and the value of a database or table variable to a local variable.

`PRESET` sets the initial value of variables at compilation time. Pre-compiled subroutines and stored executable programs save any preset values as part of the executable image that is loaded and executed at run time.

`PUT VARS` writes local data back into table or record variables.

`SET` sets variables to given constant values at execution time. It resets the values each time the command is executed.

`RECODE` recodes the value of a variable into itself or another variable.

The initial values of program variables are set to undefined unless `PRESET` is specified.

## Missing Values

Until a variable has been assigned a real value, its value is *undefined*, which is a system assigned *missing value*.

Some specific values of a variable may be treated as *missing*. A variable SEX might have valid value of 1 and 2 for Male and Female, and a value 3, for Unknown, that is treated as missing.

There are functions and procedures to get and use the actual value of the variable. In general, operations that result from evaluating a missing or undefined value yield an undefined value (e.g. adding a number to an undefined value yields an undefined value). Functions that calculate statistics on a set of values ignore undefined values.

The numeric value 0 (zero) is a normal numeric value and is different from undefined. A zero length string (a string with no characters) is also a valid value that is different from an undefined string.

Logical tests evaluate to true or false. When specifying logical tests remember that a missing value or undefined in a logical test always evaluates to false.

## Expressions

Expressions evaluate to a single value. For example:

```
COMPUTE REGION = 'Western ' + 'Canada'
COMPUTE TOTAL = 10 + 17
```

Expressions have two main elements; other *expressions* and *operators*. Operators are a symbol that specifies an operation between two expressions. Parentheses () may be used to specify the precedence (order) of operations.

Simple expressions are:

- Variables
- String and Numeric Constants
- Functions

### Variables

Variables have names and during program execution contain a value. A reference to a variable resolves itself to the value held by the variable. In general, wherever a variable may be referenced a subscripted array reference may be used.

### String Constants

String constants are expressed as characters enclosed in quotation marks (either the single or the double quotation mark ). If one type of quotation mark is used to start a string, the same type of quotation mark finishes the string. For example, in the `EVALUATE` command it is possible to specify a string inside another string by using both types of quotation mark:

```
EVALUATE X = 'NUMBR ("20")' + ' + 22'
```

### Numeric Constants

Numeric constants are numbers. A numeric constant may contain:

- a number
- one decimal character (the period)
- a leading plus or minus sign (the + or - )
- a trailing letter **E** to indicate exponentiation followed by a number (that may be signed)

Following are valid examples of the `SET` command using several forms of expressing numeric constants.

```
SET TESTNUMB (22)
SET TESTNUMB (+3.1)
SET TESTNUMB (-3.1)
SET TESTNUMB (4.5E-2)
```

## Functions

Functions are named routines that perform an operation based on values passed to the function and return a single value. Functions are specified with a function name followed by a list of values enclosed in parentheses. The values passed to functions may be constants, variables, functions and expressions. There are around 360 functions that perform various operations including string manipulation, mathematical calculations, statistics, setting and getting information from dialogs and getting information about a database or tabfile.

## Operators

### String Operators

There is one string operator, the *concatenation* operator, represented by the + sign. String concatenation appends one string value expression to another. Operations in string expressions are left to right. When string values are computed into a variable, if the string is longer than the declared length of the variable the result is truncated. Concatenating undefined or missing values result in an undefined value. For example:

```
PROGRAM
STRING*40 ADDRESS
INTEGER ZIPCODE
COMPUTE CITY = 'Chicago'
COMPUTE STATE = 'Illinois'
COMPUTE ZIPCODE = 60614
COMPUTE ADDRESS = CITY + ', ' + STATE + ' ' + FORMAT(ZIPCODE)
WRITE ADDRESS
END PROGRAM
```

In this example, ADDRESS is computed from three types of simple value expressions; string constants in quotes, variable names and the FORMAT function that converts a number to a string.

### Arithmetic Operators

There are five arithmetic operators:

- + the plus sign performs addition
- - the minus sign performs subtraction
- \* the asterisk performs multiplication
- / the slash performs division

- **\*\*** the double asterisk performs exponentiation, a number raised to a power. A number raised to a reciprocal power yields the root

Enclose signed constants that follow an arithmetic operator in parentheses. For example:

```

COMPUTE NUM1 = 10 + 20 + 33
COMPUTE NUM2 = 100 - NUM1
COMPUTE NUM2 = NUM1 * 5
COMPUTE NUM1 = 2 / 3
COMPUTE NUM1 = 4**3 | 4 cubed
COMPUTE NUM2 = NUM1**(1/3) | cube root
COMPUTE NUM = 13 * (-2)

```

In arithmetic expressions, operations of equal precedence are done from left to right. The precedence of operations is:

1. expressions within parentheses
2. functions
3. exponentiation
4. multiplication and division
5. addition and subtraction

An arithmetic operation that involves an undefined or missing value returns an undefined value. A number divided by zero yields an undefined value.

Examples:

```

COMPUTE NUM = 6 + 3 / 3 | NUM is 7
COMPUTE NUM = (6 + 3) / 3 | NUM is 3
COMPUTE NUM = 16**1 / 2 | NUM is 8, 16 divided by 2
COMPUTE NUM = 16**(1/2) | NUM is 4, square root of 16
MISSING VALUES NUM (1)
COMPUTE NUM = 1 | NUM is missing
COMPUTE NUM2 = NUM + 3 | NUM2 is undefined
COMPUTE NUM3 = 1 / 0 | NUM3 is undefined

```

## Database Variables

Commands outside a case, record or row block only access local variables. Within a block, a command can access case or record variables in addition to all local variables. The `GET VARS` and `PUT VARS` commands access case, record or row variables specifically.

It is possible, even likely, that a local variable has the same name as a variable in the record. When a retrieval references one of these variables in a case or record block, VisualPQL determines which variable is used.

- In a row block, the `GET VARS` and `PUT VARS` commands access `ROW` variables directly. All other commands access local variables.
- In a case block, a command has access to common variables. The common variable is used rather than a local variable of the same name.
- In a record block, a command has access to all record variables. In a case structured database, a command has access to both common and record variables. A common or record variable is used rather than a local variable of the same name.
- The local variable is always used outside case and record blocks.

Assigning a value to a database variable is only allowed if this is a retrieval update. If a value is assigned to a database variable, the database is updated when the record or case block is exited.

For example, the first program updates the salary on every employee record as well as listing the records. (Without the `RETRIEVAL UPDATE` command, this would not compile). The second program does not update the database, it simply produces a list of new salaries:

```
RETRIEVAL UPDATE
PROCESS REC EMPLOYEE
. COMPUTE SALARY = SALARY* 1.1
. WRITE NAME SALARY
END PROCESS REC
END RETRIEVAL

RETRIEVAL
PROCESS REC EMPLOYEE
. GET VARS NEWSALARY = SALARY
. COMPUTE NEWSALARY = NEWSALARY * 1.1
. WRITE NAME NEWSALARY
END PROCESS REC
END RETRIEVAL
```

## AUTOSET

```
AUTOSET [varlist (value_list)]
```

AUTOSET sets all implicitly declared variables and any variables not declared before the first executable command. An executable command is any command except variable declaration, variable definition and PRESET commands. AUTOSET is typically used to initialise local implicitly defined variables defined with GET VARS. AUTOSET sets variables to UNDEFINED unless a variable list and value list is specified. If such a list is specified, all AUTOSET variables are set to undefined and then the listed variables are set to the values specified in the parenthesised value list. If fewer values are specified than variables, the value list is cycled through as many times as needed to assign a value to each of the variables in the list.

In the following retrieval, AUTOSET is used to make sure that values from a previous record type 3 record aren't accidentally carried over to another case if that case happens not to have a record type 3 record.

```
RETRIEVAL
PROCESS CASES | for every case
AUTOSET | initialise variables
. PROCESS REC 1 | step thru rectype 1 recs
. GET VARS ALL | move all vars to summary rec
. PROCESS REC 3 REVERSE | step thru rectype 3, backwards
. GET VARS ALL | move all vars to summary rec
. EXIT REC | we only want this one, get out
. END REC
. PERFORM PROCS | copy summary rec to summary table
. END REC
END CASE
SAS SAVE FILE FILENAME = 'SAS.SYS' | create SAS file
 VARIABLES = ALL
END RETRIEVAL
```

## COMPUTE

```
COMPUTE varname = expression
```

Assigns the value determined by the expressions to a variable or array element. `COMPUTE` cannot be used to set a whole array. (Use `SET`)

The computed variable may be a local variable, an array element or a database variable.

The data type of the computed variable or array element must be compatible with the type implied by the expression. You must declare arrays before use with `COMPUTE`. If the computed variable has not been declared, an implicit local variable is created as either a string or real number, depending on the type implied by the computation expression.

## EVALUATE

```
EVALUATE varname = string_expression
```

The `EVALUATE` command compiles and then evaluates a VisualPQL expression during program execution. The expression that is evaluated is re-compiled and re-evaluated every time that it is traversed that is an expensive process to perform at run time. This is typically used when a user is asked to type in some condition at execution time.

If the expression is a logical expression, the command returns a 0 (zero) or a 1 (one) depending on whether the expression is true or false. If the expression is a numeric calculation, the result is returned. If the expression is a string operation, the result is a string. The left hand side variable determines the type expected from the right hand side expression. If this variable is not explicitly declared, it is implicitly declared as real.

The following retrieval allows the user to specify a condition for retrieving records.

```
RETRIEVAL
LOOP
. COMPUTE EXPRESS = SREAD('Enter search condition (CR to quit)')
. IF (LEN(TRIM(EXPRESS)) = 0) STOP
. PROCESS CASES
. PROCESS REC 1
. EVALUATE TRUE = EXPRESS
. IF (TRUE) WRITE ID NAME TO CURRDATE
. END REC
. END CASE
END LOOP
END RETRIEVAL
```

The expression to the right of the equal sign is a string expression and therefore enclosed in quotes. The syntax of the command may also require a string expression enclosed in quotes. Use a mixture of single and double quote marks. Each matching pair denotes a string. For example:

```
EVALUATE X = 'NUMBR ("20")' + ' + 22'
```

This passes a valid VisualPQL expression `NUMBR ("20") + 22` to the compiler that then produces the result 42 in X.

## GET VARS

GET VARS transfers values of database or table variables to local variables. If the referenced local variables are not explicitly declared, this command implicitly declares them with all the schema definitions of the database or table variables, including Data Type, Value Labels, Missing Values, Valid Values and Ranges. The command is only allowed inside a case, record or table block. It takes three forms:

|                                             |                                                                                                                                                                                                                                                                                                                                                                                                |
|---------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| GET VARS<br>local_var_list<br>= db_var_list | The values of local variables are assigned the values of the database or table variables. The two lists must be of equal length and the value assignments are performed listwise.                                                                                                                                                                                                              |
| GET VARS<br>db_varlist                      | There is an assumed left hand side list of local variables with the same names as the database or table variable list.                                                                                                                                                                                                                                                                         |
| GET VARS ALL                                | The keyword ALL specifies all record or table variables are assigned to local variables of the same name.                                                                                                                                                                                                                                                                                      |
| PREFIX   SUFFIX<br>'text'                   | The keywords PREFIX and SUFFIX followed by text in quotes, specify text to append to the record or table variable names to create local variables with modified names. The text is used exactly as specified so ensure the correct case (upper/lower) is used. If the modified name exceeds to maximum length for names (32 characters), a warning is printed and the unmodified name is used. |

This is the only command that accesses table variables for input.

For example: the GET VARS command is used three times to retrieve database and table data and copy it into local variables.

```
RETRIEVAL
PROCESS CASES ALL
. GET VARS ID
. PROCESS RECORD EMPLOYEE
. GET VARS NAME GENDER PREFIX 'EMPLOYEE_'
. PROCESS ROWS OCTAB INDEXED BY OCCINDEX VIA (ID)
. GET VARS POS START = CURRENT_POSITION START_DATE
. PERFORM PROCS
. END PROCESS ROWS
. END PROCESS REC
END PROCESS CASE
REPORT FILENAME = TEST.LIS
PRINT = ALL
END RETRIEVAL
```

## PRESET

```
PRESET varlist (value_list) ...
```

Assigns constants to variables and array elements during compilation. `PRESET` statements must precede the first executable command within a routine. `PRESET` may also be used in an `EXTERNAL VARIABLE BLOCK`. The preset values are the initial values when program execution begins. The syntax is identical to the `SET` command. `PRESET` happens once at compilation; `SET` happens during execution whenever the `SET` is encountered.

Values in the value list are assigned in list order to the variables in the variable list. If the value list is shorter than the variable list, the value list is cycled until a value has been assigned to each variable. If the value list is longer than the variable list, the excess values are ignored.

### Value Keywords for Undefined Values

The value list may contain value constants and the keywords `MISSING`, `NMISSING` and `SMISSING`. `NMISSING` assigns a numeric undefined value, `SMISSING` assigns a string undefined value and `MISSING` assigns the appropriate type of undefined value depending on the type of the variable being set. If `MISSING` is specified for an undeclared variable, it is implicitly declared as `REAL`.

### Repeat Values

A shorthand syntax for repeating a value is the asterisk symbol. The syntax is:

```
PRESET varlist (repeat_value * value [value_list])
```

In the following example, the first four variables are set to 2, the next three are set to 12 and the last three are set to 7, 8 and 9 respectively.

```
PRESET VAR1 TO VAR10 (4*2, 3*12, 7, 8, 9)
```

### Setting Array Elements

Specific array elements may be preset. All elements in an array may be preset by specifying the array name followed by an asterisk. Values in the value list are assigned column wise by dimension. For example:

```
INTEGER*1 ARRAY A (3,2) | declare two dimensional array A
PRESET A * (0) | preset all elements to 0
PRESET A * (1,2,3,4,5,6) | set each element to unique value
```

```
PRESET A(1,1) A(2,1) A(3,1) A(1,2) A(2,2) A(3,2)
 (1,2,3,4,5,6) | Equivalent to previous command
```

## PUT VARS

`PUT VARS` transfers values of local variables into database or table variables. This command must be used to update the values in table variables, whereas database variables are automatically updated by assignment within a record or case block. `PUT VARS` takes three forms:

`PUT VARS`  
`db_varlist =`  
`local_var_list`      The values of the database or table variables are assigned the values currently held by the local variables on the right side of the equals sign. The lists of variables must be of equal length. The value assignments are performed list wise; the first right side value is assigned to the first left hand variable, the second right to the second left, and so forth.

`PUT VARS`  
`db_varlist`      There is an assumed right side list that is identical to the database or table variable list. The referenced database or table variables must have the same name as local variables.

Note that the `PUT VARS` takes local variables as the source and sets database variables to be the same as the local variables. As these have the same name, there is an opportunity for confusion if the variable values were set inside the database block. e.g.

```
PROCESS REC EMPLOYEE
GET VARS SALARY
COMPUTE SALARY = SALARY * 1.1
PUT VARS SALARY
END PROCESS REC
```

Because this is in a record processing block, the *database* variable `SALARY` is updated by the compute, not the local variable with the same name. Then the `PUT VARS` would restore the original value of salary because that is the current value in the local variable. If database variables are updated inside a record block, the `PUT VARS` is unnecessary.

`PUT VARS ALL`      Any record or table variables with the same names as local variables are updated.

Values of keyfields in records may not be updated. Values of keyfields of the index being used on table rows may not be updated.

For example: The program retrieves data from the database and creates a new row on a table, if one does not already exist for that employee in that position. The two forms of `PUT VARS` are used, one to move a local variable to a row variable of the same name, the

other to set a row variable of a different name. Note that the index variables are set by the reference on the `ROW IS` and are not referenced by a `PUT VARS`.

```
RETRIEVAL TUPDATE
PROCESS CASES ALL
. GET VARS ID
. PROCESS RECORD EMPLOYEE
. GET VARS NAME CURRPOS SALARY CURRDATE
. NEW ROW IS OCCTAB INDEXED BY OCCINDEX (ID , CURRPOS)
. PUT VARS START_DATE = CURRDATE
. PUT VARS SALARY
. END ROW IS
. END PROCESS REC
END PROCESS CASE
END RETRIEVAL
```

## RECODE

```
RECODE [update_var =] recode_var
 [(value_list = recode_value) [...]]
```

The `RECODE` command computes a value according to the specifications and assigns the value to a named variable. The computed value may be re-assigned to the original variable or assigned to a different variable leaving the original value unchanged.

The values in the *value list* and the *recode value* are either constants or one of the *value keywords* documented below. (Expressions and variable names are not allowed as values.)

|                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| recode variable | The variable or array element with the initial value to recode. If an update variable is not specified, this variable is updated with the recoded value.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| update variable | The variable or array element to receive the recoded value if the original recode variable is not updated. The update variable must be a data type compatible with the recode value.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| recode value    | The values in the value list are converted to this single value. This value must be the same type as the update variable. Specify the keyword <code>UNDEFINED</code> to recode values to undefined.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| value list      | Specify the list of original values to recode. Specify a value list for each single recode value. Any value in the list is recoded into the single recode value. If the value of the variable is not found in a value list, the value is not recoded and is assigned unaltered to the update variable. Specify a separate parenthesised recode value list for each recode value. In the following examples: In the first, if <code>RVAR</code> is 1, 2 or 3, <code>UVAR</code> is recoded to 0. If <code>RVAR</code> has any other value, <code>RVAR</code> is copied to <code>UVAR</code> . In the second example, <code>UVAR</code> is 0 if <code>RVAR</code> is 1, 2 or 3 and 1 (one) if <code>RVAR</code> is 4, 5 or 6 : |

```
RECODE UVAR = RVAR(1,2,3 = 0)
RECODE UVAR = RVAR(1,2,3 = 0)(4,5,6 = 1)
```

You may use a number of keywords in the value list.

`THRU` Specifies an inclusive range of values. For example:

```
RECODE UVAR = RVAR (1 THRU 3 = 0)(4 THRU 6 = 1)
```

The value lists can overlap avoiding the possibility that a value (such as 3.5) falls between two value lists and is not recoded. The first match determines the recode used. Multiple ranges and multiple values can be specified in a value list. For example:

```
RECODE UVAR = RVAR
```

```
(1 THRU 3,7 THRU 99 = 0)(3 THRU 7 = 1)
```

LOWEST,LO

Specifies the lowest possible value. For example:

```
RECODE UVAR = RVAR (LO THRU 3 = 0)(3 THRU 6 = 1)
```

HIGHEST,HI

Specifies the highest possible value. For example:

```
RECODE UVAR = RVAR (LO THRU 3 = 0)(3 THRU HI = 1)
```

UNDEFINED

Specifies an undefined value. For example:

```
RECODE UVAR = RVAR (UNDEFINED,LO THRU 3 = 0)(3 THRU 6 = 1)
```

UNDEFINED may also be used as the recode value. For example:

```
RECODE UVAR = RVAR (LO THRU 3 = 0)(3 THRU HI = UNDEFINED)
```

MISSING(0)

Specifies missing values. MISSING(0) is a synonym for UNDEFINED.

MISSING(1)

MISSING(1) refers to the first missing value, MISSING(2) to the

MISSING(2)

second, and MISSING(3) to the third. For example:

MISSING(3)

```
PROGRAM
INTEGER*1 RVAR
MISSING VALUES RVAR (7,8,9)
SET RVAR (9)
RECODE UVAR = RVAR
 (MISSING(1)=4)
 (MISSING(2)=5)
 (MISSING(3)=6)
WRITE UVAR
END PROGRAM
```

BLANK

Specifies that the *blank* missing value is recoded. For example:

```
RECODE UVAR = RVAR (BLANK,7 THRU HI = 0)
```

ELSE

Specifies a recode for all values not included in any previously defined value list. If ELSE is specified, no other values may be specified in the value list. This must be the last recode specification of a set. For example:

```
RECODE UVAR = RVAR (1,2,3 = 1)(4,5,6 = 2)(ELSE = 0)
```

### Mixed Data Type Recodes

A variable of one type may be recoded into a variable of another type. In the following example, a string variable is recoded into a numeric variable.

```
PROGRAM
INTEGER*1 NUMVAR
STRING*1 STRVAR
SET STRVAR ('A')
RECODE NUMVAR = STRVAR('A' = 1)('B' = 2)(ELSE = 0)
WRITE STRVAR NUMVAR
```

END PROGRAM

If the recode variable in a mixed data type recode has a value not referenced in a recode value list, the update variable is set to undefined.

## SET

```
SET varlist (value_list) ...
```

Assigns explicit values to variables and array elements during execution. Values in the value list are assigned in list order to the variables in the variable list. If the value list is shorter than the variable list, VisualPQL cycles through the value list until a value has been assigned to each variable. If the value list is longer than the variable list, the excess values are ignored.

### Value Keywords for Undefined Values

The value list may contain value constants and the keywords `BLANK` `MISSING` `NMISSING` `SMISSING`. `BLANK` assigns blanks to a variable. This can be used to assign a blank missing value to a numeric variable. `NMISSING` assigns a numeric undefined value, `SMISSING` assigns a string undefined value and `MISSING` assigns the appropriate type of undefined value depending on the type of the variable being set. If `MISSING` is specified for an undeclared variable, it is implicitly declared as type `REAL`.

### Repeat Values

The asterisk is a symbol for repeating a value. The syntax is:

```
SET varlist (repeat_value * value [value_list])
```

In the following example, the first four variables are set to 2, the next three are set to 12 and the last three are set to 7,8 and 9 respectively.

```
SET VAR1 TO VAR10 (4*2,3*12,7,8,9)
```

### Setting Array Elements

Specific array elements may be included in the variable list. All elements in an array may be set by specifying the array name followed by the asterisk. Values in the value list are assigned column wise by dimension. For example:

```
PROGRAM
INTEGER*1 ARRAY A (3,2) | declare two dimensional array A
SET A * (0) | set all elements to 0
SET A * (1,2,3,4,5,6) | set each element to unique value
WRITE A(1,1)
END PROGRAM
```

The second `SET` statement in the above example is equivalent to:

SET A(1,1) A(2,1) A(3,1) A(1,2) A(2,2) A(3,2) ( 1,2,3,4,5,6 )

## EXTERNAL VARIABLE BLOCK

```
EXTERNAL VARIABLE BLOCK member[:V]
 [NOSAVE] [REPLACE] [PUBLIC] [VARMAP]
...
variable definitions ...
...
END EXTERNAL VARIABLE BLOCK
```

An `EXTERNAL VARIABLE BLOCK` declares a set of variables and arrays that may be shared between routines. The external variable block contains variable declarations and definitions and the `PRESET` command. No other commands are allowed in an external variable block. The block is ended with the `END EXTERNAL VARIABLE BLOCK` command.

The external variable block is compiled separately (by running it) and is stored in its compiled form in the specified member. This member is given the `:V` (for **V**ariables) suffix. This set of variables is made available to routines by specifying the `INCLUDE EXTERNAL VARIABLE BLOCK` command within a routine.

The external variable block provides a common data area that can be used by a VisualPQL program and its subroutines as an alternative to passing values between subroutines with argument lists on the `EXECUTE SUBROUTINE` command. External variables that are updated in one routine are available to all other routines that include the external variable block during VisualPQL execution.

|                      |                                                                                                                                                                                                                          |
|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>member</code>  | The member name of the compiled variable definitions. It is given the <code>:V</code> (for <b>V</b> ariables) suffix if not specified. The member name can contain complete profile, family and password specifications. |
| <code>NOSAVE</code>  | Compiles the external variable block without saving it, simply checking the code for errors.                                                                                                                             |
| <code>REPLACE</code> | Gives permission to overwrite an existing member of the same name. If such a member does not exist, the option has no effect.                                                                                            |
| <code>PUBLIC</code>  | Makes the compiled external variable block available to all users without need for passwords. These users may reference the member, but not modify or delete it.                                                         |
| <code>VARMAP</code>  | Lists the variables and their data types.                                                                                                                                                                                |

## INCLUDE EXTERNAL VARIABLE BLOCK

```
INCLUDE EXTERNAL VARIABLE BLOCK member[:V]
```

Includes the variables as local variables in the routine. Do not declare variables from the included block in the routine that includes the block.

`member`                Names a member with the `:V` suffix that is a previously compiled and stored set of variable declarations.

Use this command anywhere that variable declaration or definition commands are legal (except within another `EXTERNAL VARIABLE BLOCK`). External variables that are updated in one routine are accessible in other routines that have included the block. External variables provide an alternative mechanism to passing values on the `EXECUTE SUBROUTINE` command.

## DEFINE PROCEDURE VARIABLES

```

DEFINE PROCEDURE VARIABLES [{ INCLUDE | EXCLUDE } (varlist)]
 [NOARRAYS | ARRAYS]
 [NOEXTERNALS | EXTERNALS [
 (external_block_list)]]
 [NOSIMPLE | SIMPLE]

```

Controls the variables that are copied to the Procedure Table with the `PERFORM PROCS` command. If this command is not used, by default all local simple variables from the main routine are passed to the Procedure Table; arrays and external variables are not. If arrays or external variables are needed for the procedures, this command must be used. The options on the command are:

|                            |                                                                                                                                                                                                                                                                                                                                                                                                        |
|----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| INCLUDE                    | Specifies a list of variables included in the Procedure Table. This list may include simple program variable names, array names and external variable names. These variables must be available in the main program or retrieval.                                                                                                                                                                       |
| EXCLUDE                    | Specifies a list of variables and arrays that are excluded from the Procedure Table. All main routine program variables, external variables and arrays not mentioned in this list become part of the Procedure Table.                                                                                                                                                                                  |
| ARRAYS  <br>NOARRAYS       | Specifies that all arrays declared in the main routine are included in the Procedure Table. <code>NOARRAYS</code> is the default.                                                                                                                                                                                                                                                                      |
| EXTERNALS  <br>NOEXTERNALS | Specifies that any external variables included with the <code>INCLUDE EXTERNAL VARIABLE BLOCK</code> command in the main program or retrieval are included in the Procedure Table. You may specify the external variable blocks to include, in which case all variables in other, unspecified external variable blocks are excluded from the Procedure Table. <code>NOEXTERNALS</code> is the default. |
| SIMPLE  <br>NOSIMPLE       | Specifies that all simple variables (not arrays, not external variables) explicitly or implicitly declared in the main program or retrieval are included in the Procedure Table. <code>NOSIMPLE</code> excludes all simple variables from the Procedure Table. <code>SIMPLE</code> is the default.                                                                                                     |

## Control Flow

There are a number of commands that deal with the flow of control within a program depending on particular logical conditions.

The simple `IF,IFNOT` commands test a logical condition and execute one or more commands immediately if the condition is satisfied.

The `JUMP` command transfers control to a specific point in the program identified by a *statement label*.

Subprocedures are parts of a routine that can be executed from any point within that routine, returning control to the next statement. Subprocedures are defined at the end of the routine and share data with the routine.

### Blocks

The major control structures are *blocks* and other commands occur in blocks that are treated as a unit. Blocks are defined by commands that specify the type of block and are bounded by an `END` command. Block structures are used for all database and table access.

Blocks consist of a command that starts the block and an end command that ends the block (e.g. `LOOP / END LOOP`). Blocks may be nested within other blocks and an inner block must be completely within an outer block. Blocks may not overlap. The block structures are:

`AFTER`

A block of commands executed after all other processing. (There is no `END AFTER`; this block is delimited by the end of the program or retrieval.)

`BEGIN`

A block of commands executed at this point.

`FOR`

A block of commands executed a specified number of times.

`IFTHEN`

A block of commands executed when a condition is true. `IFNOTTHEN` is a variant that executes the block when a condition is false.

`CASE IS, RECORD IS and ROW IS.`

These blocks relate to one occurrence of a particular database record or row. Specify the

key to identify the record on the command. The commands have variants using the keywords `NEW` and `OLD`. These specify that the block is only executed if the record created/already exists.

The `JOURNAL RECORD IS` creates a block that processes data from a journal entry that matches the specified record type. This can only be used inside a `PROCESS JOURNAL` block.

#### LOOP

A block of commands executed repeatedly until specifically ended (typically by an `EXIT LOOP`).

`PROCESS CASE`, `PROCESS RECORD` and `PROCESS ROW`.

A block of commands executed once for each occurrence of a record in a database or row in a table.

`PROCESS JOURNAL` specifies a block of commands that are executed once for each occurrence of matching records on a database journal.

#### UNTIL

A block of commands executed until a specified condition is met.

#### WHILE

A block of commands executed as long as a specified condition is met.

The `EXIT` and `NEXT` commands further control processing within the block. `EXIT` passes control to the first command after the end of the block; `NEXT` goes to the next iteration of a looping block.

## Logical Conditions

A logical condition can be composed of a number of elements. The basic element is a logical expression that uses a *Relational Operator* to specify a comparison between two values. This returns either true or false. The values must be of the same type, either string or numeric. Constants, variables, functions and expressions using arithmetic, functions and other computations may be compared. When expressions are compared, all string and arithmetic operations are completed before the relational operations. When strings are compared, they are case sensitive (upper or lower) and tests for greater than/less than use the standard sort sequence: 'B' is greater than 'A' and 'BOY' is greater than 'BOX'. The relational operators are:

|                   |                                                                     |
|-------------------|---------------------------------------------------------------------|
| EQ or =           | True when values are equal.                                         |
| NE or ><          | True when values are not equal.                                     |
| GT or >           | True when the first value is greater than the second value.         |
| GE or >= or<br>=> | True when first value is greater than or equal to the second value. |
| LT or <           | True when the first value is less than the second value.            |
| LE or <= or<br>=< | True when first value is less than or equal to the second value.    |

The logical condition can also contain the *NOT* operator:

|     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|-----|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| NOT | NOT operates on a logical expression and negates its logical value. A true (non-zero, non-missing) expression is made false and a false expression is made true. NOT operates on the expression that follows up through the next relational operator or the next un-matched right parenthesis. NOT is evaluated and resolved after the entire expression to which it applies has been resolved and before other logical operators at the same level of nesting are evaluated. For example:<br>IF( NOT 1 GT 2 ) WRITE 'TRUE' |
|-----|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## Compound Conditions

Compound conditions test two logical expressions and resolve to either true or false. The relationship between the two expressions is specified using the logical operators:

|     |                                                                 |
|-----|-----------------------------------------------------------------|
| AND | Intersection. True if both expressions are true.                |
| OR  | Inclusive Union. True if either expression is true.             |
| XOR | Exclusive Union. True if one but not both expressions are true. |

These logical expressions resolve to either true or false. For example,

```
IF(1 LT 2 AND 'A' LT 'B') WRITE 'Both statements are true'
```

### Compound Logical Operations on the same variable.

When the same variable is repeatedly tested for different values, the variable name (and test if that repeats) do not have to be repeated. For example:

```
IF(ID EQ 1 OR 3 OR 12 OR 16) WRITE ID
```

The phrase 'ID EQ' is assumed to follow each 'OR' making this equivalent to:

```
IF(ID EQ 1 OR ID EQ 3 OR ID EQ 12 OR ID EQ 16) WRITE ID
```

The following two statements are equivalent:

```
IF (ID EQ 1 OR ID GT 10) WRITE 'TRUE'
IF (ID EQ 1 OR GT 10) WRITE 'TRUE'
```

The following two statements are also equivalent:

```
IF (ID EQ A OR B AND NOT C) WRITE 'TRUE'
IF (ID EQ A OR ID EQ B AND NOT ID EQ C) WRITE 'TRUE'
```

## Precedence

Parentheses may be used to establish precedence, in which case more deeply nested operations are resolved earlier. Within any expression, the various elements are resolved in the order listed below. Elements at an equal level of precedence are resolved left to right.

1. parenthesised expressions
2. functions
3. string concatenation
4. exponentiation ( \*\* )
5. multiplication and division ( \* and / )
6. addition and subtraction ( + and - )
7. relational operators EQ, NE, GT, GE, LT, LE
8. logical operator NOT
9. compound logical operators AND, OR, XOR

### Logical Values

Expressions that contain relational or logical operators resolve into true or false logical values equivalent to a 1 (one) or a 0 (zero). It is valid to compute a numeric variable to hold the result of a logical expression and to use logical expressions as arguments in functions such as SUM.

IF commands can directly test expressions that resolve to numeric values. Zero tests to false, missing tests to false and all other values test to true. In the following example, LOGICVAR contains the value 0 (zero) if the value of A does not equal the value of B, or 1 if it does:

```
COMPUTE LOGICVAR = (A EQ B)
IF (LOGICVAR) WRITE 'TRUE'
```

Because of the simple syntax for multiple tests on a single variable, compound conditions on logical variables must be specified with care. Consider the following two statements,

```
IF (LOGVAR1 AND AGE GT 21) WRITE 'TRUE'
IF (AGE GT 21 AND LOGVAR1) WRITE 'TRUE'
```

The first statement test LOGVAR1 to be 1 (true) and AGE greater than 21. However the second statement is expanded to:

```
IF (AGE GT 21 AND AGE GT LOGVAR1) WRITE 'TRUE'
```

## IF, IFNOT

```
IF (logical expression) command(s)
IFNOT (logical expression) command(s)
```

IF executes the specified command or commands when the logical expression is true. When the logical expression is false, program execution continues with the next executable command. IFNOT executes the specified command or commands when the logical expression is false. When the logical expression is true, program execution continues with the next executable command.

Any VisualPQL command may be specified as the executable result of an IF/IFNOT except:

- other logical test commands
- data definition commands
- block definition commands
- compiler commands

Separate multiple commands with semi-colons (;). For example:

```
IF (1 EQ 1) SET X(1); WRITE 'OK'
IF (X EQ 1) SET X(0);
 WRITE 'OK 2';
 EXIT RETRIEVAL
PROCESS REC EMPLOYEE
. IFNOT(SALARY GE 2000) NEXT REC
. WRITE NAME SALARY
END REC
```

The example goes to the next record when salary is not greater than or equal to 2000 and also when salary is missing (because the expression is treated as false).

## JUMP

```
JUMP statement label
JUMP (statement label , statement label, ...) variable
```

`JUMP` transfers control to the point in the program identified by the specified *statement label*. Execution continues serially from that point. It is possible to jump to a statement label or to *fall through* to it from the normal flow of control. A `JUMP` command must be at an equal or higher level of nesting and within the same nested set as the statement label that it references. `JUMP` can jump out of a block, but not into the middle of a more deeply nested block. `JUMP` can not jump out of or into a `SUBROUTINE` or `SUBPROCEDURE`.

Specify a statement labels as the first element on a line. A statement label is a name followed by a colon (:). When referenced in the `JUMP` command, do not specify the colon. If you need to use a non-standard name (enclosed in curly brackets { }) as a label in PQL, there is a possible conflict with the syntax for command labels used to control command processing. Avoid this conflict by indenting the label and specifying a full stop in column 1.

The first form of `JUMP` specifies a single statement label and transfers control to the command after the label.

The second form of the command, the computed `JUMP`, transfers control to the *n*th label in a specified list of labels, where the numeric variable contains the value *N*.

## AFTER

AFTER PROGRAM or AFTER RETRIEVAL

Specifies a block of commands executed when the program or retrieval is complete. The block is typically used to print report ending information and for actions that are taken just before ending the program.

Table, case or record processing commands are not allowed. Reference may not be made to any data in the database or table files. Local variables are available for output.

This command is typically used with the Full Report Procedure.

AFTER PROGRAM is identical to AFTER RETRIEVAL, except that it is used in programs rather than retrievals.

## BEGIN

BEGIN

Specifies the beginning of a block of commands. The commands in a BEGIN block are executed when control reaches this point.

END BEGIN

Delimits the BEGIN block.

EXIT BEGIN

Transfers program control to the statement following the END BEGIN command. It is usually used conditionally to terminate processing of the block.

BEGIN can be used to anywhere in a program to group a set of commands that have some common purpose. For example, initialisation at the start of the program.

```
PROGRAM
BEGIN
. COMPUTE TOTAL = 0
. other initialisation commands
END BEGIN
PROCESS ROWS SCHOOLS.STUDENTS
. COMPUTE TOTAL = TOTAL + 1
. other commands performed for every row
END ROW
END PROGRAM
```

BEGIN is often used to create a better structure in a program with complex logical conditions and to avoid the use of JUMP or very complex IFTHEN / ELSEIF constructs. For example:

```
BEGIN
. IF (logical condition) EXIT BEGIN
. visualpql code
. IF (logical condition) EXIT BEGIN
. IF (logical condition) EXIT BEGIN
. visualpql code
END BEGIN
```

## EXIT

EXIT [ blocktype ]

EXIT blocktype stops execution of the block at that point and transfers control to the first command following the END blocktype command. An EXIT can be used in any block. If the blocktype is specified, the command exits the innermost block of that type. If the blocktype is not specified, the command exits the innermost block. It is good practice to specify blocktype. An EXIT command may be specified in the following blocks:

|                         |                                                                                                                           |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------|
| EXIT BEGIN              | Exits the current BEGIN block.                                                                                            |
| EXIT CASE               | Exits the current case processing block.                                                                                  |
| EXIT FOR                | Exits the current FOR block.                                                                                              |
| EXIT IF                 | Exits the current IFTHEN or IFNOTTHEN block.                                                                              |
| EXIT JOURNAL            | Exits the current JOURNAL RECORD IS block.                                                                                |
| EXIT LOOP               | Exits the current LOOP block.                                                                                             |
| EXIT PROCESS<br>JOURNAL | Exits the current PROCESS JOURNAL loop.                                                                                   |
| EXIT PROCESS<br>RECORD  | Exits the current PROCESS RECORD loop.                                                                                    |
| EXIT PROCESS<br>ROW     | Exits the current PROCESS ROW loop.                                                                                       |
| EXIT PROGRAM            | Terminates the PROGRAM after executing any commands in the AFTER PROGRAM block. STOP is a synonym for EXIT PROGRAM.       |
| EXIT RECORD             | Exits the current record processing block.                                                                                |
| EXIT<br>RETRIEVAL       | Terminates the RETRIEVAL after executing any commands in the AFTER RETRIEVAL block. STOP is a synonym for EXIT RETRIEVAL. |
| EXIT ROW                | Exits the current row processing block.                                                                                   |
| EXIT UNTIL              | Exits the current UNTIL block.                                                                                            |
| EXIT WHILE              | Exits the current WHILE block.                                                                                            |

## FOR

```
FOR control_var = startvalue, endvalue [,increment]
```

The `FOR` command specifies the number of times the commands in a `FOR` block are executed. `FOR` assigns a new value to the control variable each time the block is repeated.

|                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|--------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>control_var</code> | The control variable must be a numeric variable and may not be an array element. It is incremented by the value of <i>increment</i> during each pass through the block. When the value of this variable exceeds the <i>endvalue</i> , the block is exited and control is transferred to the first statement following <code>END FOR</code> . If the control variable is not explicitly declared, it is declared implicitly as a <code>REAL*8</code> local variable. |
| <code>startvalue</code>  | The control variable is set to this value when the block is first executed. The value may be a program variable, array element reference, expression or a numeric constant.                                                                                                                                                                                                                                                                                         |
| <code>endvalue</code>    | The end value determines the final pass through the <code>FOR</code> block. The value may be a program variable, array element reference, expression or a numeric constant.                                                                                                                                                                                                                                                                                         |
| <code>increment</code>   | The value by which the control variable is incremented during each pass through the block. The value may be a program variable, array element reference, expression or a numeric constant. The default increment value is <b>1</b> (one).                                                                                                                                                                                                                           |
| <code>END FOR</code>     | Delimits the <code>FOR</code> block.                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <code>EXIT FOR</code>    | Terminates processing of the <code>FOR</code> block and transfers program control to the first statement following <code>END FOR</code> . This command is usually executed conditionally with an <code>IF</code> command or from within a conditionally executed block of commands.                                                                                                                                                                                 |
| <code>NEXT FOR</code>    | Terminates processing of the current pass through the <code>FOR</code> block and passes control to the start of the next <code>FOR</code> loop after incrementing and checking the control variable.                                                                                                                                                                                                                                                                |

The following sequence is executed in a `FOR` block:

- The control variable is set equal to the starting value, all commands in the block are executed, then the increment is added to the control variable.
- If the control variable is greater than the end value or if the control variable becomes a missing value, execution of the block is terminated. Otherwise the block is executed again.
- If either start value or end value is missing, the `FOR` block is skipped.

- If the control variable is a database variable (the `FOR` block is within a record or case block), the database variable is updated on each pass through the block.
- When the increment is positive, the `FOR` block is not performed when the start value is larger than the end value. When increment is negative the `FOR` block is not performed when the start value is smaller than the end value.

## IFTHEN

```
IFTHEN (logical_expression)
IFNOTTHEN (logical_expression)
```

The `IFTHEN` command defines a block of commands that are executed conditionally.

In its simplest form, `IFTHEN` defines a block of commands that are executed when a logical expression is true. An `IFNOTTHEN` block of commands is executed when the logical expression is false. Any command may be included within an `IFTHEN` or `IFNOTTHEN` block, including other complete blocks of commands and other logical test commands.

Other conditional blocks within the condition, can be defined with the `ELSE`, `ELSEIF`, and `ELSEIFNOT` commands. Each conditional block is terminated with another `ELSEIF`, `ELSE` or the `END IF`. An `IFTHEN` block may contain multiple `ELSEIF` and `ELSEIFNOT` blocks. It may only contain one `ELSE` block.

|                                               |                                                                                                                                                                                                                                                                                                                                                               |
|-----------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>END IF</code>                           | Delimits the <code>IFTHEN</code> block.                                                                                                                                                                                                                                                                                                                       |
| <code>EXIT IF</code>                          | Directs the flow of the program to the statement following the <code>END IF</code> of the block in which it appears.                                                                                                                                                                                                                                          |
| <code>ELSE</code>                             | Specifies a block of commands that are executed when no other commands are executed in the <code>IFTHEN</code> block. The block of commands is delimited by the <code>END IF</code> .                                                                                                                                                                         |
| <code>ELSEIF</code><br><code>ELSEIFNOT</code> | Specifies a block of commands that are executed when the conditions specified on the <code>IFTHEN</code> and any other previous <code>ELSEIF</code> commands are not satisfied and the condition specified on this command is satisfied. The block of commands is delimited by a further <code>ELSEIF</code> , <code>ELSE</code> or the <code>END IF</code> . |

```
IFTHEN (logical expression)
. commands executed when above expression is TRUE
. ELSEIF (logical expression)
. commands executed when above expression is TRUE
. ELSEIFNOT (logical expression)
. commands executed when above expression is FALSE
. ELSE
. commands executed when no other blocks are executed
END IF
```

## LOOP

### LOOP

LOOP repeatedly executes a block of commands. By definition, a LOOP block is an infinite loop and a command such as EXIT or JUMP must be used to terminate looping. The compiler does not check for the existence of these commands within a LOOP.

|           |                                                                                                                                                                                              |
|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| END LOOP  | Delimits the LOOP block.                                                                                                                                                                     |
| EXIT LOOP | Terminates looping, directing program execution to the statement following the END LOOP. This command is frequently executed conditionally, as with an IF command, to terminate the looping. |
| NEXT LOOP | Terminates processing of the current pass through the block and passes control to the next iteration of the current LOOP block (control is passed to the statement following LOOP).          |

## NEXT

```
NEXT [blocktype]
```

Some blocks (e.g. `WHILE`) are looping structures and execute repeatedly until some controlling condition is met. In looping blocks, the `NEXT blocktype` command transfers control to the first command in the block at the next iteration.

If the `blocktype` is specified, the command transfers control to the next iteration of the innermost block of that type. If `blocktype` is not specified, control is transferred to the next iteration of the innermost looping block. It is good practice to specify `blocktype`. `NEXT` may be specified in the following looping blocks:

|                         |                                                                                                                                                        |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>NEXT CASE</code>  | Transfers control to the next iteration of <code>PROCESS CASES</code> if another case exists.                                                          |
| <code>NEXT FOR</code>   | Transfers control to the next iteration of the <code>FOR</code> , if the end condition specified on the <code>FOR</code> has not yet been reached.     |
| <code>NEXT LOOP</code>  | Transfers control to the next iteration of the <code>LOOP</code> .                                                                                     |
| <code>NEXT REC</code>   | Transfers control to the next iteration of <code>PROCESS RECORDS</code> if another record exists.                                                      |
| <code>NEXT ROW</code>   | Transfers control to the next iteration of <code>PROCESS ROWS</code> if another row exists.                                                            |
| <code>NEXT UNTIL</code> | Transfers control to the next iteration of the <code>UNTIL</code> , if the end condition specified on the <code>UNTIL</code> has not yet been reached. |
| <code>NEXT WHILE</code> | Transfers control to the next iteration of the <code>WHILE</code> , if the end condition specified on the <code>WHILE</code> has not yet been reached. |

For example:

```
PROCESS REC EMPLOYEE
. IF (GENDER = 1) NEXT PROCESS REC
...
END PROCESS REC
```

## UNTIL

UNTIL (logical expression)

UNTIL repeatedly executes a block of commands until the logical expression becomes true. When the expression becomes true, the block is exited and program control is transferred to the statement following the `END UNTIL` command.

The condition of the logical expression is tested at the start of each pass through the UNTIL block. If the condition is initially true, the block is skipped entirely.

|                         |                                                                                                                                                                                                                                                                                                              |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>END UNTIL</code>  | Delimits the UNTIL block                                                                                                                                                                                                                                                                                     |
| <code>NEXT UNTIL</code> | Terminates the current pass through the block and transfers control to the next iteration of the UNTIL command. This command is usually executed conditionally, as the result of a logical test (e.g. with the <code>IF</code> command). This command may only be used within an UNTIL block.                |
| <code>EXIT UNTIL</code> | Terminates processing of the UNTIL block and transfers program control to the statement following the <code>END UNTIL</code> . This command is usually executed conditionally, as the result of a logical test (e.g. with the <code>IF</code> command). This command may only be used within an UNTIL block. |

## WAIT

WAIT num\_exp

Pauses execution of this program for the specified number of tenths of a second.

```
PROCESS RECORD PATIENT LOCK = 4 | get the patient record
. LOOP
. IF(SYSTEM(36) = 1) EXIT LOOP | exit if we get the record
. WRITE 'Waiting for locked record'
. WAIT 5 | wait half a second
. RETRY RECORD | try to get the record
. END LOOP
...
END PROCESS RECORD
```

## WHILE

`WHILE (logical expression)`

`WHILE` repeatedly executes a block of commands while the expression is true. When the expression becomes false, execution of the block is terminated and program control is transferred to the statement following the `END WHILE`. The condition of the expression is tested at the start of each pass through the `WHILE` block. If the condition is initially false, the block is skipped entirely.

|                         |                                                                                                                                                                                                                                                                      |
|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>END WHILE</code>  | Delimits the <code>WHILE</code> block                                                                                                                                                                                                                                |
| <code>NEXT WHILE</code> | Terminates the processing of the current pass through the block and passes control to the next iteration of the block. This command is usually executed conditionally, as the result of a logical test (e.g. with the <code>IF</code> command).                      |
| <code>EXIT WHILE</code> | Terminates processing of the <code>WHILE</code> block and transfers program control to the statement following the <code>END WHILE</code> . This command is usually executed conditionally, as the result of a logical test (e.g. with the <code>IF</code> command). |

## SUBPROCEDURE

`SUBPROCEDURE name [ NOAUTOCASE ]`

A subprocedure is a named, structurally complete, block of commands that may be executed from any point within a routine with the `EXECUTE SUBPROCEDURE` command. Subprocedure names must be unique within the routine. A subprocedure is compiled with, and belongs to, the routine it is defined in. A subprocedure shares variables with the routine and other subprocedures and has access to all variables available to the routine in which it is defined. Declare any variables used by the routine before they are referenced. (i.e. do not declare new variables in the subprocedure that are referenced by the routine.)

Define subprocedures at the end of the routine ahead of any VisualPQL Procedures or an `AFTER RETRIEVAL` block. There is an implicit `STOP` (or `RETURN` if the routine is a subroutine) just before the first subprocedure.

The `SUBPROCEDURE` command begins a subprocedure definition. The subprocedure definition ends with the `END SUBPROCEDURE` command.

If you define a subprocedure with the name `RT_ERROR`, this procedure is given control if a run time error occurs. It is then this subprocedure's responsibility to take any appropriate action necessary. If the subprocedure does not exit the routine, control is passed back to the next command following the command that caused the run time error.

`NOAUTOCASE` Specifies that the subprocedure is called from within a `CASE` block, allowing specification of a `RECORD` block within the subprocedure and/or references to `CIR` variables. If the subprocedure is executed outside a `CASE` block, the execution of a `RECORD` block causes an error and the main routine terminates. Any references to variables with names matching `CIR` variable names reference `CIR` variables. If the subprocedure is executed outside a `CASE` block, references to `CIR` variables return undefined values.

References to other variables in the subprocedure are to local variables, unless within a `RECORD` block physically specified in the subprocedure itself, even if the subprocedure is executed from within a record block. However, if the subprocedure *is* executed from within a record block, this alters the behaviour of certain VisualPQL functions that allow an expression as a variable name, such as `VARGET` and `VARPUT`. Because these functions resolve their variable references at execution time and are checked against any `CIR` variables, then any active record variables ahead of any local variables. execution of these functions from within a

record block references any matching record variable regardless as to whether the functions are in a subprocedure.

**END SUBPROCEDURE**

**END SUBPROCEDURE** ends the block of subprocedure code. Control is passed back to the first command following the **EXECUTE SUBPROCEDURE** command that invoked the subprocedure.

**EXIT SUBPROCEDURE**

**EXIT SUBPROCEDURE** exits the subprocedure and control is passed to the first statement following the **EXECUTE SUBPROCEDURE** command that invoked the subprocedure.

## EXECUTE SUBPROCEDURE

```
EXECUTE SUBPROCEDURE name
```

Transfers control to the first line of code in the named subprocedure. When the subprocedure is exited, control returns to the first statement following the EXECUTE SUBPROCEDURE. This command may appear anywhere in a routine including within a subprocedure.

Subprocedure Example:

```
PROGRAM
TIME NOON NOWTIME ('HHiMM') | declare time variables
COMPUTE NOON = '12:00' | set the value of midday
COMPUTE NOWTIME = NOW(0) | get the current time
IFTHEN (NOWTIME LT NOON)
. EXECUTE SUBPROCEDURE MORNING | execute MORNING
. ELSE
. EXECUTE SUBPROCEDURE AFTERN | execute AFTERN
ENDIF
SUBPROCEDURE MORNING
. WRITE 'Good Morning'
END SUBPROCEDURE
SUBPROCEDURE AFTERN
. WRITE 'Good Afternoon'
END SUBPROCEDURE
END PROGRAM
```

## Reading and Writing Files

There are VisualPQL commands that read and write files.

Programs can read from and write to named files. These may be text or *binary* files and output files can be newly created or opened in *append* mode, which adds to the end of any existing file of the same name.

An interactive program can `READ` without naming a file. This displays a text box for the user to enter data. If a `WRITE` does not specify a file, output is to the default output file. In an interactive session, the default output file is the scrolled output window, but can be assigned to a file. Batch runs always have output assigned to a file.

The `OPEN` command is optional. If a file is referenced on a `READ` command and the longest record in that file is 80 characters or less, it is automatically `OPENED`. If a file is referenced on a `WRITE` command, it is automatically `OPENED` with the same record length as the current default output. By default, this is 120 characters. To open a file with any other characteristics, use the `OPEN` command.

By default, output files are new files. Use the `APPEND` option on the `OPEN` command to append to existing files of the same name.

### Filenames

Files have an external name that may be fully qualified with path or subdirectory specifications. Specify filenames in quotes if they contain characters such as slash, comma or blank that have meaning in VisualPQL. For example:

```
READ ('D:\SIRDB\EMPDB\DATA\REC1.DAT') DATALINE (A20)
```

When VisualPQL accesses a file, it passes the name directly to the operating system without checking it.

SIR/XS uses a short, internal, name or Attribute that is mapped to the long, fully qualified filename. If a filename is a valid SIR name not in quotes, it is checked against the current set of attributes. If a match is found, the full filename that the attribute refers to is used. Temporary attributes with internal names are automatically created for long names and specific, named attributes are created by various VisualPQL and general SIR/XS commands including `OPEN` and `SET ATTRIBUTE`.

## Binary Files

Normally files read or written by explicit reads and writes in VisualPQL are *text* files; that is they contain readable characters together with end of record characters and can be viewed with a text editor. VisualPQL can also read and write *binary* files using a combination of the `BINARY` keyword on the `OPEN` command and the `Hex`, `IB` and `RB` formats on the `READ` and `WRITE` commands. Integers and real numbers in a binary file are in internal computer format and a binary file does not have end of record markers. Reading and writing binary files means that exactly the number of bytes specified in the read or write variable list are transferred between the file and the program. It is the programmer's responsibility to ensure that file and variable specifications match.

There are three formats that specify that binary data is being transferred. The `HEXw` format is for generic strings of binary data and these are not altered in any way by the read/write process. The `w` specifies the length of field and can be up to 4094. Declare the variable being used as a normal VisualPQL string. If strings are written as text using the `A` format as opposed to Hex format, if the string contains a hex character '00', it is taken as the end of any text output line and the line is truncated to that point. The `IBw` and `RBw` formats allow the transfer of numeric data. `IB` is for Integers in Binary format and `RB` is for Reals in Binary format. The `w` specifies the length of field and is 1,2 or 4 for integers, 4 or 8 for reals. For example, the following program copies any file:

```

program
integer*1 ibyte1
string*256 a256
string*250 oldfl newfl
real*8 cnt fsize
cnt=0
c ** Change these names to required filenames
compute oldfl = 'splash.bmp'
compute newfl = 'copy.bmp'
compute fsize=filestat(oldfl,10)
open inf dsnvar=oldfl lrecl=256 binary
open outf dsnvar=newfl lrecl=256 binary write
loop
cnt=cnt+256
ifthen(fsize-cnt ge 256)
. read (inf,error=end) a256(HEX256)
. write (outf) a256(HEX256)
else
c Less than 256 bytes left so go 1 byte at a time
. loop
. read(inf, err=end) ibyte1(ib1)
. write(outf) ibyte1(ib1)
. end loop
endif
pool
end:
end program

```

## OPEN

```

OPEN fileid
 [BINARY]
 [DELETE]
 [DSN = 'file_name' | DSNVAR = str_varname |
 | | LDIVAR = str_varname]
 [DYNAMIC]
 [ERROR = statement_label]
 [IOSTAT = num_varname]
 [LRECL = max_rec_length]
 [MEMBER [REPLACE]]
 [READ | WRITE [APPEND]]

```

Opens the specified file or member for READ or WRITE access, READ is the default. The READ, WRITE and CLOSE commands may use an opened file.

Files are accessed by the READ and WRITE commands.

**fileid**            The internal name or attribute of the file is the name referenced by any READ, WRITE or CLOSE command. If the external filename is exactly the same as the attribute name and the file is in the default directory, the DSN may be omitted. Otherwise the DSN clause must be specified.

If a member is being opened, this must be the member name. If the family name is not specified, the current default family name is used. If the profile name is not specified, the current default profile is used.

**APPEND**            Specifies that the opened file is added to the end of any existing file with the same name. If the file does not exist, it is just created.

**BINARY**            Specifies that the opened file is treated as a binary file. Data is read or written exactly as given and no translation to text takes place. There are no end of line or end of record markers.

To illustrate the differences between writing text and binary, suppose a write statement references an integer with a value of say 100. In the text file (with no format), this results in the characters 100 (in hexadecimal a character 0 is 30 and character 1 is 31 so this is 313030) but in a binary write (with format IB4) of an integer\*4, this results in the hexadecimal 4 bytes value 64000000 (This is true on a PC but different byte ordering applies on other machines that gives different results). Similarly, if a binary file is read, the internal fields on the read

must match the type of data being read. You must know what you are doing to use binary files!

|                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|--------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DELETE                   | Specifies that the opened file (or member) is deleted after it is closed. Files are closed when a <code>CLOSE</code> command is executed or at termination of the program. The <code>CLOSE</code> command also has a delete option.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| DSN   DSNVAR<br>  LDIVAR | <p>DSN specifies a fully qualified external filename enclosed in single quotes (').</p> <p>DSNVAR specifies the name of a string variable that contains the fully qualified filename. A value of asterisk (*) specifies that the default input or output files are used that may be useful during program development and debugging.</p> <p>LDIVAR specifies the name of a string variable that contains the attribute name of the file. A value of asterisk (*) specifies that the default input or output files are used that may be useful during program development and debugging.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| DYNAMIC                  | Specifies that file attribute entry is not stored with the subroutine object code.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| ERROR                    | Specifies a statement label where control is transferred if an error occurs in opening the file. If the <code>ERROR</code> clause is not specified and an error in opening the file occurs, an error message is displayed and program execution terminates.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| IOSTAT                   | <p>Specifies a numeric variable to hold a return code. If <code>IOSTAT</code> is specified and an error occurs, a value is assigned to the specified variable and control transfers to the <code>ERROR</code> clause statement label. If there is no <code>ERROR</code> clause, execution continues with the next statement. If <code>IOSTAT</code> is not specified and an error occurs a message is printed and the program is terminated. The codes for normal files are:</p> <pre> 0 Successful Open - 5 File locked (in use) - 6 File not found - 8 Access problem - 9 Miscellaneous problems </pre> <p>The codes that apply to members are standard error message numbers and are:</p> <pre> 439 Cannot form member name. 440 Member must be type text. 441 Family password mismatch. 442 Member password mismatch. 443 Cannot open procedure file. 444 Family not found. 445 Member found but replace mode not specified. 446 Member not found. 447 Cannot open scratch file to process member. 449 Procedure file already open in this retrieval. </pre> |
| LRECL                    | Specifies the longest record length (in bytes) on the file being opened.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |

If `LRECL` is not specified, 80 bytes for `READ` and the current default output width (that itself defaults to 120) for `WRITE` are used. If a record longer than the specified length is encountered, the record is truncated and a warning message is issued.

`MEMBER`

Specifies that the fileid is a procedure file member. Once opened, a member can be read from or written to. Only one member of the procedure file may be open at any given time. If a member is not explicitly closed with the `CLOSE` command, it is closed at the termination of the program.

`REPLACE`

`REPLACE` gives permission to overwrite an existing member. If the specified member does not exist, this keyword has no effect. If an attempt is made to write to an existing member and this keyword is not specified, the member is not overwritten and an advisory message is issued. A family cannot be created with an `OPEN` command.

`READ` | `WRITE`

`READ`, the default, opens the file or member for read access.  
`WRITE` opens the specified file or member for write access.

## CLOSE

```
CLOSE fileid [DELETE]
```

Closes the specified file. `CLOSE` may be used to close a file early in a program in order to reopen it later during the same program. Closing and then opening a file allows the program to re-read the file from the beginning.

`DELETE`            The file or member is deleted when it is closed.

## DELETE PROCEDURE FILE MEMBER

DELETE PROCEDURE FILE MEMBER name

Removes the named procedure file member at execution time. The name may be a string constant, a string variable or [string\_expression]. The type must be specified. Any member type may be deleted. (The delete option on the OPEN/CLOSE commands can only be used with text members.)

The format of the name is the normal member name format i.e.:

[procfile.][family[/pword].]member[/pword]:type

## READ

### Interactive READ

```
READ ['prompt_text'] I/O_list
 ['prompt_text'] I/O_list ...
```

### File READ

```
READ (fileid, [ERROR=label] [IOSTAT=VARNAME] [MEMBER]) I/O_list
```

The interactive `READ` reads input from the user, popping up a box with the specified prompt (a question mark `?` is displayed if no prompt text), space for data entry and response buttons that allows entry of data. The program waits for the user to enter data. Regardless of the number of fields on the input specification, a single box is popped. (If multiple boxes are wanted, specify multiple prompts and i/o lists.) Input fields are delimited by spaces. The user must enclose a string input field in quotes if it contains spaces.

It is recommended that this interactive read is not used. Use `DISPLAY TEXTBOX` instead.

The file `READ` reads from a file. Records are read from first to last sequentially, a new record being read each time a `READ` of the file is executed.

Input is read according to the i/o list and values read from the input are assigned to program variables.

If the file is a normal text file then fields from the file are translated into appropriate internal formats. Each read reads a single record.

If the file is a binary file, input fields must match the type of the fields on the file in order to process data correctly and just those fields specified are read.

If explicit `OPEN` and `CLOSE` commands are not used, the first time the `READ` is executed the file is opened and program termination closes the file. If the file cannot be opened successfully, an error message is displayed and the program stops executing.

A `READ` command is not a block control statement and simply executes without looping. In order to read through a complete file, it is necessary to enclose the `READ` in a looping block, typically a `WHILE (IOSTAT =0)`.

An `IOSTAT = varname` may be specified as a return code. An `ERROR = label` may be specified that is gone to when an error condition is encountered on the `READ`. Return codes with any value other than zero are errors. When the program reaches end of file, this results in an error return code (-4) and programs normally treat any non-zero return code as end of file.

```
PROGRAM
STRING * 80 LINE
INTEGER*1 STAT
STAT = 0
WHILE (STAT = 0)
. READ(INPUT.TXT,IOSTAT = STAT)LINE(A80)
. WRITE LINE(A80) | display what we read
END WHILE
END PROGRAM
```

## Options

|               |                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 'prompt text' | The text displayed on the screen to request input. Specifying prompt text or omitting a filename indicates that interactive input is expected. The text box is displayed with a single line for input. The maximum input size is 80 characters. Multiple fields may be read at once depending on the format specification. A second prompt may be specified on the single command that is essentially identical to repeating the command. |
| fileid        | Specifies the name of the input file or member. This may be a filename or an attribute name.                                                                                                                                                                                                                                                                                                                                              |
| ERROR         | Specifies a statement label in the program. Control is passed to that point if an error or end-of-file condition occurs.                                                                                                                                                                                                                                                                                                                  |
| IOSTAT        | Specifies an integer variable for a return code. Return codes are:<br>0 Success<br>-1 File not Open<br>-4 End of File                                                                                                                                                                                                                                                                                                                     |
| MEMBER        | Specifies that the file being read is a member from the procedure file. This is unnecessary if already specified on an explicit <code>OPEN</code> .                                                                                                                                                                                                                                                                                       |

## I/O List - Input Specification

I/O lists contain variable names and their formats. The formats can be fixed-field, free-field or pictures, and can contain positional specifiers.

### Fixed-field formats

Iw Integer, w digits wide

Fw.d Real single precision, w digits wide, d decimal places

Ew.d Real, scientific notation

Dw.d Real double precision, w digits wide, d decimal places

Aw String, w characters wide

Bw String, characters reversed (**backwards**)

DATE 'format' Date in specified date format.

TIME 'format' Time in specified time format.

### Free-field formats

\* Any free-field variable

I\* Integer

F\* Real

E\* Real, scientific notation

D\* Real, double precision

A\* String

### Positional specifiers

nX Skip the next **n** columns.

nT Move (**T**ab) to column **n** before reading the next variable. Column **n** can be to the left or right of the current position.

## Binary Formats

HEXw Binary string, w digits(up to 4094)

IBw Binary integer, w digits(1,2 or 4)

RBw Binary real, w digits (4 or 8)

## Delimited Input

The free field input formats (represented by \*) must establish what represents a field on the input. If the field is in quotes, then the string in the quotes is used (start and end quotes stripped off). In unquoted strings, blanks, tabs and commas are delimiters. Multiple blanks or tabs are treated as a single delimiter. Multiple commas are taken as multiple fields i.e. each comma corresponds to one field on the read input format. Two commas together in the record result in blank input to that field.

## Picture clauses

Instead of a format, a picture clause enclosed in quotes can be specified. Aside from the following specific picture characters, any other characters that appear in the picture must appear "as is" in the input record. The input field must conform to the specified picture:

- d any digit
- s digit, decimal point, plus or minus
- a any letter
- u any uppercase letter
- l any lowercase letter
- c any character

## REREAD

```
REREAD (fileid, [ERROR= label/] [IOSTAT= varname/]) I/O_list
```

Rereads the last record read from the specified sequential file. The syntax for the command is the same as the READ command.

The REREAD may specify a different input specification from the previous READ. For example:

```
PROGRAM
STRING*80 TEXTLINE
STRING*20 STUDENT
INTEGER*1 RECTYPE NAME SEX ETH_CODE RANK
READ (DATAFILE.DAT) RECTYPE(I1) NAME(A20) 3X
 SEX(I1) ETH_CODE(I1) RANK(I1)
IFTHEN (RECTYPE = 2)
. REREAD (DATAFILE.DAT) RECTYPE (I1)TEXTLINE(A80)
ENDIF
...pql commands
END PROGRAM
```

## WRITE

### Default WRITE

```
WRITE I/O_List
```

### File/Member WRITE

```
WRITE (fileid [ERROR=label] [IOSTAT=varname] [MEMBER [REPLACE]]
[NOEOL]) I/O_List
```

The default `WRITE` writes output to the scrolled output or assigned output file. There is no paging on interactive output. Pages are maintained when standard output is assigned to a file.

The text file or member `WRITE` creates a new file, appends to an existing file (with the `APPEND` option on the `OPEN`) or creates a new member and writes records to the end of the specified file each time the `WRITE` command is executed. Output is formatted according to the i/o list and assigns values from the program variables to the output.

The binary file `WRITE` creates a new file or appends to an existing file (with the `APPEND` option on the `OPEN`) and writes fields to the end of the specified file each time the `WRITE` command is executed. The exact values from the program variables are written to the output in the internal format of those variables if they are strings or if the binary formats are used: `IB1`, `IB2` or `IB4`, `IR4` or `IR8`, (Use `IB4` for dates or times).

`fileid` Specifies the name of the output file or member. This may be an attribute or filename.

The filename `STDOUT` can be used when options such as `NOEOL` are required but output is still directed to the default output.

The filename `CGI` specifies that, if the program is being used in CGI mode from a webserver, output is returned to the server (which means that it appears on the user's web page). If the program is run in normal mode, i.e. not from a webserver, then a file called `sircgi.htm` is created. The `NOEOL` option can be used with `CGI`

`ERROR = label` Specifies a statement label in the program. Control is passed to this label if an error condition occurs on the command. If the `ERROR` or

|         |                                                                                                                                                                                                                                                                                                                                            |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|         | <p>IOSTAT clause is not specified and an error occurs, an error message is written and the program is terminated.</p>                                                                                                                                                                                                                      |
| IOSTAT  | <p>Specifies the name of a numeric variable that returns any error code. The value zero indicates no error, a negative number indicates the type of error that occurred.</p>                                                                                                                                                               |
| MEMBER  | <p>Specifies that the file being written is a member from the procedure file. This is unnecessary if already specified on the explicit OPEN.</p>                                                                                                                                                                                           |
| REPLACE | <p>Gives permission to overwrite an existing member.</p>                                                                                                                                                                                                                                                                                   |
| NOEOL   | <p>Specifies that the output is written without an end of line. The next WRITE simply continues putting data to the file and no new record is created. This can be used to build a complete file in a particular format without any end of line characters or can be used interspersed with WRITE commands that do create end of line.</p> |

## I/O List - Output Specification

I/O lists contain variable names or expressions and formats. The formats can be fixed-field, free-field, or pictures, and can contain positional parameters. If a slash (/) appears in the output specification, the write is positioned to the next line. The backslash (\) is not allowed in an output specification.

To specify an expression in an output specification, enclose it in square brackets. The value of the expression is calculated and output according to the output format at execution time.

Format specifications for date, time and categorical variables may be numeric or string. The data is automatically converted to a proper output string, if string specifications are used.

If an output format is not specified, the first format encountered is taken to apply to all previous fields without a format.

If no formats or a free-field (\*) format is specified, the following defaults are used:

- Integers use an "Iw" format, where w is the minimum required to print the value.
- Floating point either use an "Fw.d" format, where "w" and "d" are adjusted to maintain significance up to 8 decimal places or the "E" (scientific) notation with 5 significant decimal digits, depending on the value written.
- Strings use an "A\*" format that writes the number of characters in the string followed by a blank.
- Date, time and categorical variables are written as strings.

Do not specify an output format for string constants (characters enclosed in quotes); these are written out as specified. If a format is specified, an error message is issued.

### Picture clauses

A picture can be specified for numeric fields instead of a format. A picture is a string of characters, enclosed in quotes. Within the picture certain characters have special meanings:

Each digit can be represented by a "9", a "z", a "\*" or a "\$". "9" specifies that leading zeros are replaced by blank; "z" specifies that leading zeros are written; "\*" specifies that leading zeros are replaced by "\*"; "\$" after an initial "\$" character, represents a floating dollar sign where leading zeros are suppressed. If the field has a value of zero, a picture of all "9"s results in blanks and all "\$"s results in a single "\$" since all leading zeros are suppressed; if a single zero is wanted, specify a single "z" as the last character of the

picture.

A period represents the decimal point and separates the specification into characters before and after the decimal point. There can only be one decimal point (period) in the picture. If there are insufficient digits to display the integer portion of the field (including any minus sign when negative and \$ when specified), the field is written as all 'X's. The decimal component is rounded to match the number of decimal digits specified. If there are no decimal digits in the picture, the field is rounded to the integer value.

Specify comma (,) to insert this character. If leading zeros are suppressed (by blanks or a floating dollar), any leading commas are suppressed. If a single dollar sign is specified, it is output in that position. If multiple dollar signs are specified, these suppress leading zeros and result in a floating dollar sign that is output in front of the first significant digit. After the decimal point, the special characters "9", 'Z', "\$" and "\*" are all equivalent and specify a digit. Any other characters are treated as any other special character.

Negative numbers, by default, are output with a minus sign ahead of the first significant character. If an explicit minus sign is included as the last character in the picture, and the number is negative, the minus is written at that point. Any other characters are output at the position specified in the picture.

The same picture specifications apply to the PFORMAT function. For example:

```
PROGRAM
c The following formats produce following output
WRITE [123.4] ('$ZZ,ZZZ.99-') | $00,123.40
WRITE [123456789] ('ZZZ-ZZZ-ZZZ') | 123-456-789
WRITE [-123.4] ('99,999.99') | -123.40
WRITE [-123.4] ('$99,99Z.99') | $ -123.40
WRITE [-123.4] ('$$, $$Z.99') | -$123.40
WRITE [-123.4] ('ZZ,ZZZ.99') | -0,123.40
WRITE [-123.4] ('$ZZ,ZZZ.99') | $-0,123.40
WRITE [-123.4] ('99,999.99-') | 123.40-
WRITE [-123.4] ('$99,99Z.99-') | $ 123.40-
WRITE [-123.4] ('$$, $$Z.99-') | $123.40-
WRITE [-123.4] ('ZZ,ZZZ.99-') | 00,123.40-
WRITE [-123.4] ('$ZZ,ZZZ.99-') | $00,123.40-
WRITE [1234.56] ('$*****.**') | $***1234.56
WRITE [1234.56] ('Z Z Z Z . Z Z') | 1 2 3 4 . 5 6
WRITE [1234.56] ('ZZZZ') | 1235
END PROGRAM
```

## Fixed-field formats

Iw Integer, w digits wide

Fw.d Real single precision, w digits wide, d decimal places

Ew.d Real, scientific notation

Dw.d Real double precision, w digits wide, d decimal places

`Aw` String, *w* characters wide

`Bw` String, characters reversed (**backwards**)

`DATE 'format'` See date formats for a complete description. For example:  
`WRITE XBEG (DATE 'MMM DD, YYYY')`

`TIME 'format'` See time formats for a complete description. For example:  
`WRITE ALARM (TIME 'HH:MM:SS')`

### Free-field formats

\* Any free-field variable

`I*` Integer

`F*` Real

`E*` Real, scientific notation

`D*` Real, double precision

`A*` String

### Positional specifiers

`nX` Skip the next *n* columns before writing the next variable.

`nT` Move (**T**ab) to column *n* before writing the next variable. *n* must be 1 or greater.

### Array Element Printing

`nE` Print the next *n* elements of the previous array. *n* must be 1 or greater. Multi-dimensional arrays are printed so that entry (1,1) is first, (2,1) is second through to (n,1) that is followed by (1,2) etc.

## Database Access

There are VisualPQL commands that access and update data stored in a Database.

Any VisualPQL program that uses `CASE OF RECORD` processing commands must begin with the `RETRIEVAL` command. This implicitly opens the current default database for access by the retrieval. By default, the database is opened for read, meaning that the retrieval can get data from the database but cannot add, delete or modify data. Retrievals that create, modify or delete database data must use the `UPDATE` option on the `RETRIEVAL` command.

By default, the last database connected is the default.

If a VisualPQL program references a database, it must be connected when the retrieval is compiled and must be connected when the retrieval is executed.

There are commands to connect and disconnect databases. A VisualPQL retrieval can access a specified database and then all references are to variables in that database.

A SIR/XS session can be started with an `MST=` parameter, in which case any database access by VisualPQL programs is through the concurrent `MASTER` process. A session can logon and logoff to Master as necessary. VisualPQL database access is either local or through Master depending on the current status of the master setting.

When operating in concurrent mode, locks on individual records may be specified. If a retrieval does not specify locks, defaults are used. If a retrieval specifies locks and does not run through master, any locking is ignored. An identical VisualPQL retrieval can run concurrently and independently. Even if there are processes accessing the database through `MASTER`, a retrieval can be run in a SIR/XS stand-alone session and use read only mode against the same database.

### Data availability during retrieval

During the execution of a retrieval, data can be in local variables, case or common variables and record variables. The same name may be used for local, case and record variables and the actual variable referenced depends on the placement of the command and its scope.

### Local Variables

Local variables include the variables and arrays declared explicitly or implicitly in the routine and any variables included with an `INCLUDE EXTERNAL VARIABLE BLOCK`.

### Case variables

If the database is a case structured database, each case in the database has one CIR or *Common Information Record*. The CIR contains common variables including the *Case Identifier* variable that uniquely identifies each case. The common variables are defined when the database schema is created.

During execution, a retrieval accesses a CIR with one of the Case Processing commands. A case processing command defines a block of commands, a *Case Block*. The common variables are available at any point in the case block, including within record processing blocks. When a case block is executed the case variables are read and other commands within the block can use the common variables. When the case block is exited or when another case is read, if the CIR has been modified it is replaced in the database.

### Record variables

Databases are made up of multiple Record Types. Each record type contains a set of variables defined during database schema definition. Some of these variables may be key fields that, in combination with the case identifier variable, uniquely identify an individual record. The structure of the record type cannot be altered through a retrieval.

During execution, the retrieval accesses records with one of the Record Processing commands. A record processing command defines a block of commands, a *Record Block*. Within a record block other commands may get values from or put values into the record variables. When a record block is executed the record variables are read and other commands within the block can use these variables. When the record block is exited or when a new record is read, if the record has been modified it is replaced in the database.

If a record block is nested within another record block, the variables for the outer record are restored when the inner block is exited.

### Priority of Access to Data

At any given point during retrieval execution, a retrieval potentially has access to one set of common variables, one set of record variables and the local variables. It is possible, even likely, that a local variable has the same name as a common or record variable in the database. If the referenced variable exists both as a database variable and a local variable, VisualPQL uses the following rules to decide which variable to use:

- If the reference is outside a case or record block, the local variable is used.
- If the reference is within a case block, the case variable is used rather than a local variable of the same name.
- If the reference is within a record block, the record variable is used rather than a local variable of the same name.

- If the reference is within a record block and the CIR and the record both contain a variable of the same name, the record variable is used. If the variable is updated, both the record variable and the CIR variable are updated.

### Skipping blocks

Commands specify a particular record or record type to retrieve. If there are no matching records, then the block of commands is skipped completely. When developing a retrieval, this must be taken into account. For example:

```
RETRIEVAL
PROCESS CASES ALL
OLD REC IS EMPLOYEE
. GET VARS ALL
. PROCESS REC 2
. GET VARS ALL
. WRITE ID NAME SALARY
. END PROCESS REC
END REC IS
END PROCESS CASE
END RETRIEVAL
```

The `WRITE` command is not executed if there are no record type 2 for an employee and thus that employee does not appear on the output. Any variables that are updated within the block, are not reset. The `AUTOSET` command can be used to reset variables in this instance.

## PQL CONNECT DATABASE

```
PQL CONNECT DATABASE database_name_exp
[PREFIX prefix_exp]
[SECURITY exp,exp,exp]
[IOSTAT varname]
```

Connects the specified database at execution time. Sets this as the default database. Does not automatically run any `SYSTEM` procedures.

All the parameters, except the `IOSTAT` varname are expressions; enclose any name constants in quotes.

There is a table of connected databases, one of which may be the current default database. By default, the last database connected is the default and is the first database referenced by a `RETRIEVAL`.

If a VisualPQL retrieval references a database, it must be connected when the program is compiled and must be connected before it is executed. This means that the `PQL CONNECT DATABASE` command cannot be used to connect and compile or connect and execute within one VisualPQL process.

A database can also be connected with the `SIR/XS` command `CONNECT DATABASE`.

`SECURITY` Three expressions separated with commas. Specify the database password, then the read password then the write password.

`IOSTAT` A numeric variable that returns the database connection number if successful or a negative number (in the range -2001 to -2058) if there is a problem with the connection. See error messages.

## PQL DISCONNECT DATABASE

```
PQL DISCONNECT DATABASE database_name_exp
[IOSTAT varname]
```

Disconnects the named database. If PQL DISCONNECT is executed on the default database, the default is set to zero and SYSPROC is set as the profile.

**IOSTAT** A numeric variable that returns a negative number (in the range -2001 to -2058) if there is a problem with the connection. See error messages.

## DATABASE IS

```
DATABASE [IS] dbname [UPDATE | NOUPDATE]
```

Starts a block that accesses a specified database. May only be used in a `RETRIEVAL`.

Inside this block, all references are to variables in the new database. Any standard VisualPQL commands can be used in this block. (This is not a looping block so `NEXT` cannot be specified.)

Within a `RETRIEVAL`, the initial database is the default database.

Note the database name in this command is a constant e.g. `DATABASE IS COMPANY` not an expression as the name is required at compile time as well as during execution.

## END DATABASE IS

```
END DATABASE [IS]
```

Ends definition of a database block. References outside this block are to the original database. When the block is exited, if there was an original database, it is made current.

## Case Processing Commands

The case processing commands define a block of commands that is delimited with the `END CASE` command. These commands are not valid for caseless databases. Each time a case is accessed with one of these commands, the common CIR variables are available to other commands within the block. There are two commands that process cases:

- `PROCESS CASES` retrieves a specific set of cases and updates these cases if required.
- `CASE IS` (and the variants `NEW CASE IS` and `OLD CASE IS`) retrieves or creates a single case with a specified case id.

All updates to the database, including the creation of a new case, require the `UPDATE` keyword on the `RETRIEVAL` command. New cases are created with the `CASE IS` or `NEW CASE IS` block. Existing cases may be accessed with the other types of case blocks.

### Commands in CASE Blocks

All commands(except `AFTER RETRIEVAL`), including other case block commands, may be used within a case block. The following commands may only be used within a case block:

- `DELETE CASE`
- `EXIT CASE`
- `NEXT CASE`
- `RESTORE CIR`
- `RETRY CASE`
- `BACKUP`

Be aware of how commands transfer values from the CIR to local variables and vice-versa:

- Any VisualPQL command in a case block that assigns a value to a variable assigns the value to a common variable if a common variable of the specified name exists.
- The value of the case identifier variable can never be modified from within a case block.

`COMPUTE` can be used within case blocks to update database variables. If the computed variable is a CIR variable, the value of the expression is assigned to it and the database value is modified. For example, in a database that has a common variable called `COMMVAR`, the following retrieval allows the user to modify its value.

```

RETRIEVAL UPDATE
CASE IS 5
WRITE 'Current Value of COMMVAR is ' COMMVAR
COMPUTE COMMVAR = SREAD('Enter New Value for COMMVAR')
END CASE
END RETRIEVAL

```

GET VARS transfers the value of a CIR variable to a local variable. When a CIR variable is referenced within a case block, the value of the CIR variable is used (even if a local variable of the same name exists).

GET VARS can implicitly define a local variable with the *definition* of the database variable as well transferring the value, whereas COMPUTE simply assigns the value.

PUT VARS transfers values of local variables into database variables. PUT VARS may only be used in update mode.

The following example assigns the value of a CIR variable to a local variable that is accessed later from outside the case block.

```

RETRIEVAL
. PROCESS CASES REVERSE COUNT = 1 | find the last case
. GET VARS COMMVAR | get value of COMMVAR
. END CASE
WRITE COMMVAR | display value of COMMVAR
END RETRIEVAL

```

### Case Functions

- COUNT(rt\_num) Returns the number of records of the specified record type belonging to the current case. If the specified record type is not defined, an undefined value is returned. (Use in the case block.)
- SYSTEM(14) Returns a 1 if the last CASE IS, NEW CASE IS, or OLD CASE IS block was executed. It returns 0 (zero) if the last block was not executed.
- SYSTEM(15) Returns a 1 if the last CASE IS or NEW CASE IS block created a new case. It returns 0 (zero) if the block did not create a new case. (Use the SYSTEM functions after the case block.)

## CASE IS

```
[NEW | OLD] CASE IS caseid [LOCK = num]
```

`CASE IS` defines a block that accesses the single case specified by the case id. The case id value may be a constant or local variable, including an array reference.

|                          |                                                                                                                                                                                                                                                          |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>CASE IS</code>     | Accesses a single case in the database. If the case does not exist, then in <code>UPDATE</code> mode, a new case is created; if the retrieval is not in <code>UPDATE</code> mode and the case does not exist, the <code>CASE IS</code> block is skipped. |
| <code>OLD CASE IS</code> | Accesses an existing case. If the case does not exist, the <code>CASE IS</code> block is skipped.                                                                                                                                                        |
| <code>NEW CASE IS</code> | This command is only allowed in a <code>RETRIEVAL UPDATE</code> and creates a new case with the specified case identifier value. If the specified case already exists, the <code>NEW CASE IS</code> block is skipped and no new case is created.         |
| <code>LOCK</code>        | Specifies case level locking for concurrent operations.                                                                                                                                                                                                  |

## DELETE CASE

```
DELETE CASE [KEEP CIR]
```

Deletes the current case (CIR) and all records belonging to the case. This command is only allowed in UPDATE mode. Only users with the highest read and write security passwords for the database may delete cases and records.

KEEP CIR           Deletes all records belonging to the case but does not delete the Common Information Record.

The following example deletes all cases that do not have any record type 1 records.

```
RETRIEVAL UPDATE
PROCESS CASES
IF(COUNT(1) EQ 0) DELETE CASE
END CASE
END RETRIEVAL
```

## END CASE

END CASE

Terminates CASE IS and PROCESS CASE blocks.

END CASE IS terminates CASE IS blocks only.

END PROCESS CASE terminates PROCESS CASE blocks only.

## **EXIT CASE**

EXIT CASE

Terminates processing of the current case block and transfers control to the first statement following the `END CASE`.

## NEXT CASE

NEXT CASE

Terminates processing of the current case and retrieves the next case if there is one that meets the `PROCESS CASE` specification.

## PREVIOUS CASE

### PREVIOUS CASE

Terminates processing of the current case and retrieves the previous case if there is one that meets the `PROCESS CASE` specification. Use of this with `SAMPLE` or `COUNT` yields unpredictable results.

## PROCESS CASE

```
PROCESS CASES [ALL]
 [COUNT = total [,inc [,start]]]
 [LIST = caseid list]
 [LOCK = num]
 [REVERSE]
 [SAMPLE = fraction [,seed]]
```

Defines the beginning of a case processing block that is delimited by the `END CASE` command. `PROCESS CASE` and `PROCESS CIR` are synonyms.

The options on `PROCESS CASES` define the set of cases that are stepped through. The commands in this block are executed once for each case within the specified set.

If there is no `PROCESS CASE` command in the retrieval and the database has a case structure, a `PROCESS CASES ALL` command is generated before the first executable command in the retrieval. The `NOAUTOCASE` option on the `RETRIEVAL` command suppresses this.

**ALL** Processes all cases in the database. This is the default.

**COUNT** Specifies the number of cases to process. The values for *total*, *increment* and *start* are variables or expressions which should resolve to positive integer values. A single integer number is a valid expression.

*total*  
Specifies the maximum number of cases to retrieve. If more cases are requested than are available, the retrieval reads as many as exist. For example, to process the first 5 cases in the database:

```
PROCESS CASES COUNT = 5
```

*increment*  
Specifies the "skipping factor" for retrieving cases. An increment of 3 produces every third case. For example, to access a total of 5 cases, retrieving every tenth case:

```
PROCESS CASES COUNT = 5 , 10
```

*start*

- Specifies the ordinal of the first case processed. For example, 3 starts retrieving at the third case.
- LIST** Specifies a list of case identifier values of the cases to process. The list may be composed of constants or variables. The **THRU** keyword specifies an inclusive range of case id values. For example:
- ```
PROCESS CASES LIST = 1,2,8,17
WRITE IDNUM
END CASE

PROCESS CASES LIST = 1,5 THRU 10,18,20
WRITE IDNUM
END CASE

SET FVAR LVAR (5,10)
PROCESS CASES LIST = FVAR THRU LVAR
WRITE IDNUM
END CASE
```
- LOCK** Specifies case level locking for concurrent operations.
- REVERSE** Specifies that the cases are processed in reverse order. Note that if you specify a list of specific cases, the list order is the order of processing regardless of this setting.
- SAMPLE** Retrieves a random sample of cases from the database. The specified values for fraction and seed are variables or expressions which should resolve to positive numbers. A specific decimal number or positive integer is a valid expression.
- fraction*
- Specify a number between 0 (zero) and 1 (one). The retrieval generates a random number between 0 and 1 for each case. If the number falls between 0 and the specified number, the case is retrieved, otherwise processing goes on to the next case. Since each case is evaluated for inclusion independently, the actual sample may not be exactly the requested size particularly for databases with a limited number of cases. For example, to process 25% of the cases:
- ```
PROCESS CASES SAMPLE = .25
```
- seed*
- Specify the starting seed for the random number generator. A given seed guarantees that the same set of random numbers is generated. Note that the PQLProcedures may also use samples and the **SAMPLE** option here, resets the seed for any sampling. It is recommended that the seed is set using the **SEED** option on the retrieval command and that all subsequent sampling in the retrieval uses the random numbers generated from that. If a seed is not specified, the random number generator is not reset. For example
- ```
PROCESS CASES SAMPLE = .25 , 13579
```

RESTORE CIR

RESTORE CIR

Re-reads the CIR from the database. When a case block is first executed, a CIR is read. Updates to common variables are performed in memory. The modified record is re-written when the case block is exited, another CIR is accessed or when a `BACKUP` command forces a write. A `RESTORE CIR` before the data is re-written cancels all modifications.

Record Processing Commands

Record processing commands access a specific record type. These commands define a block of commands that is delimited with the `END RECORD` command. When a record is accessed, the record variables are available to other VisualPQL commands within the record block. On case structured databases, record blocks must be nested within a case block. There are two commands that process records:

- `PROCESS RECORD` retrieves a specific set of records and updates these records if required.
- `RECORD IS` (and the variants `NEW RECORD IS`, `OLD RECORD IS`) retrieves or creates a single record with a specified record key.

To perform any updates to the database, including the creation of new records, the retrieval must be in update mode. New records are created with a `RECORD IS` or `NEW RECORD IS` block. Existing records may be accessed with the other types of record blocks.

Record Functions

<code>RECLEVEL(0)</code>	Returns the update level at which this record was last written to the database. Can only be used in a record block.
<code>SYSTEM(3)</code>	Returns the update level at which a record was last written to the database. The record referred to is the record from the last record block executed.
<code>SYSTEM(16)</code>	Returns a 1 if the last <code>REC IS</code> , <code>NEW REC IS</code> , or <code>OLD REC IS</code> block was executed. It returns 0 (zero) if the last block was not executed.
<code>SYSTEM(17)</code>	Returns a 1 if the last <code>REC IS</code> or <code>NEW REC IS</code> block created a new record. It returns 0 (zero) if the block did not create a new record. (Use the <code>SYSTEM</code> functions after the record block.)

Commands in RECORD Blocks

All commands, including other record block commands, may be used within a record block. The following commands may only be used within a record block:

```
DELETE RECORD
EXIT RECORD
NEXT RECORD
RESTORE RECORD
RETRY RECORD
BACKUP.
```

Be aware of how commands transfer values from the record to local variables and vice-versa:

- Any command in a record block that assigns a value to a variable assign the value to a database record variable if a record variable of the specified name exists.
- The values of the case identifier variable and record type keyfield variables can never be modified from within a record block.

`COMPUTE` or `PUT VARS` can be used in record blocks to update database variables. The database variables can only be updated in a retrieval in update mode. If the computed variable is a common or record variable, the value of the expression is assigned to it and the database value is modified.

`COMPUTE` or `GET VARS` can be used to transfer the value of a database variable to a local variable. `GET VARS` implicitly defines a local variable with the same definition as the database variable as well transferring the value, whereas `COMPUTE` simply assigns the value. When a record variable is referred to in an expression, the record variable is used even if a local variable of the same name exists.

RECORD IS

```
[ OLD | NEW ] RECORD IS {name | number} (value list)
```

Defines a record block that accesses a single record. The value list must specify a valid value for every keyfield of the record type. If any keyfield is missing or undefined, the block is skipped. `RECORD` and `REC` are synonyms. In a case structured database, record blocks occur within case blocks and the records accessed belong to the current case.

<code>RECORD IS</code>	Accesses a record if it exists. If it does not exist and the retrieval is not in <code>UPDATE</code> mode, the <code>RECORD IS</code> block is skipped. In <code>UPDATE</code> mode, a new record is created.
<code>OLD RECORD IS</code>	Accesses an existing record. If the specified record does not exist, the <code>OLD REC IS</code> block is skipped.
<code>NEW RECORD IS</code>	Creates a new record with the specified key values. Only allowed in a <code>RETRIEVAL UPDATE</code> . If the specified record exists, the <code>NEW REC IS</code> block is skipped.
<code>name number</code>	The record name or number. This must be specified.
<code>value list</code>	A list of values expressed as constants, variable names or array references. Each element in the list represents a value for a keyfield. Specify the values in sequence to match keyfields in the order defined in the schema. Specify a valid value for every keyfield of the record type. If any keyfield is missing or undefined, the record block is skipped. If the record type being accessed has no keyfields, specify the command without a value list.
<code>LOCK</code>	Specifies record level locking for concurrent operations.

DELETE RECORD

DELETE RECORD Deletes the current record.

A record can only be deleted in UPDATE mode. Deleting a record requires write security at an equal or higher level to the record write security level. It also requires write security at an equal or higher level to the highest write security level of any variable in the record. For example, the following deletes all record type 3 records that were updated at update level 47:

```
RETRIEVAL UPDATE
PROCESS CASES
. PROCESS REC 3
. IF(RECLEVEL(0) EQ 47 )DELETE REC
. END REC
END CASE
END RETRIEVAL
```

END RECORD

```
END RECORD [ IS ]  
END PROCESS RECORD  
END PROCESS JOURNAL  
END JOURNAL RECORD
```

Terminates RECORD IS, PROCESS RECORD **blocks and** PROCESS JOURNAL, JOURNAL RECORD **blocks**.

END RECORD IS **terminates** RECORD IS **blocks**.

END PROCESS RECORD **terminates** PROCESS RECORD **blocks**.

END PROCESS JOURNAL **terminates** PROCESS JOURNAL **blocks**.

END JOURNAL RECORD **terminates** JOURNAL RECORD **blocks**.

REC is a synonym for RECORD.

EXIT RECORD

EXIT RECORD

Terminates processing of the current record block and transfers control to the first statement following the `END RECORD`.

`REC` is a synonym for `RECORD`.

NEXT RECORD

NEXT RECORD

Terminates processing of the current record and retrieves the next record if it exists. REC is a synonym for RECORD. The following example processes only the males in the database.

```
RETRIEVAL
PROCESS CASES
. PROCESS RECORD EMPLOYEE
.   IFNOT (GENDER = 1) NEXT REC  |- go to next rec if not male
.   WRITE NAME SSN BIRTHDAY
. END REC
END CASE
END RETRIEVAL
```

PREVIOUS RECORD

PREVIOUS RECORD

Terminates processing of the current record and retrieves the previous record if it exists. REC is a synonym for RECORD. Use of this with SAMPLE or COUNT yields unpredictable results.

PROCESS REC

```

PROCESS RECORD name | num
               [ LOCK = num ]
               [ INDEXED BY index_name ]
               [ ONETIME ]
               [ REVERSE ]
               [ AFTER (value list) ]
               [ AFTER (value list) THRU (value list) ]
               [ AFTER (value list) UNTIL(value list) ]
               [ FROM (value list) ]
               [ FROM (value list) THRU (value list) ]
               [ FROM (value list) UNTIL(value list) ]
               [ THRU (value list) ]
               [ UNTIL (value list) ]
               [ VIA (value list) ]

```

PROCESS RECORD, (PROCESS REC is a synonym), defines a block of commands that are executed repeatedly, once for each record of the specified type within the specified range. If the command does not use the INDEXED BY construct, then, in a case structured database, the command must be inside a case block and the records accessed are those belonging to the current case.

Note: Specifying a record selection clause (e.g. AFTER, FROM, THRU, etc.) on the PROCESS REC locates records through the database index, which is an efficient way to process subsets of records. Use record selection clauses whenever possible.

name num	The name or number of the record type to retrieve. This is required.
LOCK	Specifies record level locking for concurrent operations.
INDEXED BY	Specifies the name of the index to use to retrieve records. All record selection clauses can be used in conjunction with INDEXED BY. When an index is used, the key values are those values used for the index.
ONETIME	By default, when no records exist within the specified range, the block is skipped. ONETIME forces the block to be entered with the values of the record variables set to undefined, when there are no matching records.
REVERSE	Processes the records in reverse order. If used with a record selection clause, processes the selected subset in reverse order. If specifying a range of record keys to select, specify these in the normal way (i.e. the FROM key has a lower value than the UNTIL key).
AFTER	Selects records whose key value is greater than that specified by the

value list. `AFTER` can be used in combination with `THRU` or `UNTIL` to specify a range of keys.

<code>FROM</code>	Selects records whose key value is greater than or equal to the key specified by the value list. <code>FROM</code> can be used in combination with <code>THRU</code> or <code>UNTIL</code> to specify a range of keys.
<code>THRU</code>	Specifies the key value to process to and include in the retrieved subset. Use <code>AFTER</code> or <code>FROM</code> to specify a beginning record for processing.
<code>UNTIL</code>	Specifies the key value to process up to but not include in the retrieved subset. Use <code>AFTER</code> or <code>FROM</code> to specify a beginning record for processing.
<code>VIA</code>	Selects records whose key value matches (equals) the key specified by the value list. If a partial key value list is specified, all records matching the partial list are selected. <code>WITH</code> is a synonym for <code>VIA</code> .
value list	A list of values for keyfields. These may be expressed as constants, variable names or array references. The list is matched with values of keyfields in the order defined in the schema or the order defined in the index. The value list need not list values for the entire set of keyfields. Low level keys may be omitted, but not higher levels. For example, if <code>A</code> , <code>B</code> , and <code>C</code> represent a record's keyfields, then:

```
VIA (A, B, C)    legal
VIA (A)         legal
VIA (A, B)     legal
VIA (, ,C)     invalid, needs A and B
VIA (A, ,C)    invalid, needs B
```

During execution, if a value is undefined or missing, the value list is treated as if it were terminated with the value previous to the undefined value.

Note: In earlier versions of the software, undefined values caused an execution error.

RESTORE REC

RESTORE REC

Re-reads the current record from the database. When a record block is first executed, a data record is read into memory. Updates to record variables are performed in memory. The modified record is re-written to the database when the block is exited. A `RESTORE REC` cancels all modifications when done before the record is re-written.

`RETRY RECORD` is a synonym.

BACKUP

BACKUP

Writes modified database record or CIR to the database.

During a Retrieval Update, updates are performed in memory. The modified CIR or record is copied to the database when the processing block is exited or before another CIR or record occurrence is accessed.

BACKUP forces a write and is very seldom needed.

This is only allowed in retrievals in update mode.

Processing Database Journals

A database journal is a record of updated records on the database. (A database unload file is in identical format and can also be processed in VisualPQL with these commands.)

The journal file consists of a linked set of entries, one entry per update run. Each entry consists of a set of images of updated records in that run. The images consist of before and after images of updated records.

The `PROCESS JOURNAL` command allows you to get information about the various entries on the journal and to select one or more entries to process. When processing through an entry, data records are read in sequence from the earliest to the latest. Within the `PROCESS JOURNAL` block, a `JOURNAL RECORD IS record_type` names the record that is of interest. This block is given control when a record of that type is read. Within this block, you can use normal VisualPQL to access the data from the journaled record using the record variable names. e.g.

```
PROCESS JOURNAL
. JOURNAL RECORD IS record_type
.   PQL access to record variables
. END JOURNAL RECORD IS
END PROCESS JOURNAL
```

You can specify a `PROCESS JOURNAL` in a program and it can run with no database attached to examine the headers on the file. However a database schema is needed to interpret the data and to compile any `JOURNAL RECORD IS record_type` commands and so the `JOURNAL RECORD IS record_type` can only appear in a retrieval (you may want to specify `NOAUTOCASE`). In a retrieval, the file being processed must match the current database both at compile time and at execution time.

If you are processing a journal for a case structured database, note that the journal entries for individual records do not have any non-key common variables; these are on a separate journal for the CIR. The common vars can only be referred to in a `JOURNAL RECORD IS CIR` block and not within the individual record blocks. Further, the journal holds a sequential series of records which is written as the records are updated. If records in a case are updated but no common vars are updated, then there will not be a journal entry for the CIR. If some common vars are updated, the CIR journal entry follows the individual record journal entries. If only common vars are updated, then there will not be a journal entry for the individual record type.

Do not specify a `JOURNAL RECORD IS record_type` block inside another `JOURNAL RECORD IS record_type` block. Since the block is only entered for the specified record type, the inner block with a different record type is never executed.

You can compile `JOURNAL RECORD IS record_type` blocks which are not physically in a `PROCESS JOURNAL` block so they might be in a sub-routine or sub-procedure. If a `JOURNAL RECORD IS record_type` block is executed that is not in an executing `PROCESS JOURNAL` block, it is simply skipped.

PROCESS JOURNAL

```

PROCESS JOURNAL
  [FILENAME= fname_expression ] (sr5 is the default)
  [FROM = updlevel | START = date [,time] ]
  [THRU = updlevel | END   = date [,time] ]
  [REVERSE]
Return Data
  [DATE = varname] [ENDDATE = varname]
  [TIME = varname] [ENDTIME = varname]
  [LEVEL = varname]
  [RECORD = varname]
  [TYPE = varname]
  [USER = varname]

```

`PROCESS JOURNAL`, defines a block of commands that are executed repeatedly, once for each journal record within the specified range. Some records are *headers* that identify the update run and some are data records that contain information about a particular record type that was updated in the run.

The `PROCESS JOURNAL` command has two sets of keyword specifications. The first set specify a filename, which journal entries to process and whether to go from earliest to latest or in reverse. Selecting entries to process can be on the basis of update levels or date and time and can specify either start points, end points or both. All of these specifications are expressions which evaluate to the value to use. Typically these are specified as a variable name which holds the value.

The second set of specifications name a number of variables that are then used by the process to return information to the program. If you are selecting multiple entries, then information may be needed about which entry is being processed. When processing the potentially multiple records within an entry, information may be needed about the individual record image that is above and beyond the actual data in the record.

The named file is processed until a header matches the `PROCESS JOURNAL` specification. Control is then passed to the VisualPQL inside the block for each record until a new header is reached that does not match the specification and the block is exited.

<pre>FILENAME = fname_expression</pre>	<p>The journal file to process. If not specified, the default is the current journal file for the default database (the .sr5 file). Specify the name as an expression, that is a string variable or other string expression. If you are specifying a known filename, you can simply enclose it in quotes e.g.</p>
--	---

	FILENAME = 'COMPANY.UNL'	
FROM = updlevel		The first update level to start processing journal entries. If not specified, processing starts at the first journal entry on the file.
THRU = updlevel		The last update level to process. If not specified, processing stops after processing the last journal entry on the file.
START = date [,time]		The date and, optionally the time, of the first journal entry to start processing. Date is an expression that resolves to a date in format MMIDDYY. Time, if specified, is an expression that resolves to a time in format HHIMISS. Use either (or neither) a start time or a from update level, do not specify both.
END = date [,time]		The date, and optionally the time, of the last journal entry to process. Date is an expression that resolves to a date in format MMIDDYY. Time, if specified, is an expression that resolves to a time in format HHIMISS. Use either (or neither) an end time or a thru update level, do not specify both.
REVERSE		Specifies that the journal is processed in reverse sequence. This only effects the sequence of entries not the sequence of records presented within entries. i.e. If the journal holds entries relating to updates that took the database from update level 5 to 6, 6 to 7 and 7 to 8, REVERSE presents 7 to 8, then 6 to 7 then 5 to 6. This also affects the way that you specify selection - specify the level/date/time to start that is higher/later than the one to finish.
DATE = varname		Specify a variable name. If specified, this contains the start date of the journal entry currently being processed.
ENDDATE = varname		Specify a variable name. If specified, this contains the end date of the journal entry currently being processed.
TIME = varname		Specify a variable name. If specified, this contains the start time of the journal entry currently being processed.
ENDTIME = varname		Specify a variable name. If specified, this contains the end time of the journal entry currently being processed.
LEVEL = varname		Specify a numeric variable name. If specified, this contains the update level of the journal entry currently being processed.
RECORD = varname		Specify a numeric variable name. If specified, this contains the record type of the journal data record currently being processed.
TYPE = varname		Specify a numeric variable name. If specified, this contains the type of the journal record currently being processed. The journal type is a positive number for data records and a negative number for journal headers. Types are: 1 New record written (This is the type of all data on an unload file.) 2 Before existing record updated 3 After existing record updated. Note that these before and after records are a pair and are written together.

- 4 Before Record deleted
- 1 Journal Data header
- 2 Unload Schema header
- 3 Unload Data header
- 4 Journal Schema header
- 5 User header

`USER = varname` Specify a character variable name capable of holding a 32 byte name. If specified, returns the name of the user responsible for the update. This is taken from the `SIRUSER` if it is specified on start up, or from the system environment variables (from `sir.ini`) `USERNAME` or `USER`. The username can be set during a session by the `SIRUSER PQL` function.

See Processing Journals for more details.

JOURNAL RECORD IS

JOURNAL RECORD IS {name | number}

Defines a record block that is entered when the journal records being processed match the specified record type. If there is a JOURNAL RECORD IS *record* inside the PROCESS JOURNAL block and the journal record being processed matches the record type specified then, when that block is reached, the block is processed. If the journal record does not match a JOURNAL RECORD IS *record* that block is skipped.

Specify either a record number or a record name. To process the CIR specify either 0 or CIR.

Within the block, you can use normal record variable names to process the data from the journal record. You can use these for reports or can use them for other database manipulation. You can nest other blocks e.g. database access, if required.

See Processing Journals for more details.

EXIT JOURNAL IS

EXIT JOURNAL IS

Terminates processing of the current journal record and exits the journal record is block.

EXIT PROCESS JOURNAL

EXIT PROCESS JOURNAL

Terminates processing of the current journal record and exits the process journal block.

NEXT PROCESS JOURNAL

NEXT PROCESS JOURNAL

Terminates processing of the current journal record and retrieves the next journal record. This may be a data record or may be a new header.

NEXT PROCESS HEADER

NEXT PROCESS HEADER

Terminates processing of the current journal set of records and retrieves the next journal header. If processing a large journal update or an unload, this is much more efficient than processing through every data record looking for the next header.

Concurrent VisualPQL

Use the `MST=` parameter when starting a SIR/XS session, or use the `SET MASTER` command to use Master and thus use concurrent VisualPQL. VisualPQL programs run concurrently may update the database concurrently with other products using Master.

MASTER must be running when a client tries to use it and all retrievals then run through MASTER until the use of Master is turned off with a `CLEAR MASTER` command.

Read only retrievals run *much* faster when run in stand alone mode rather than through MASTER. Retrievals execute properly in either mode.

Utilities ignore Master settings and may require exclusive access to the database.

Locking

The `LOCK =` keyword on the database access commands, apply a lock to the case or record being accessed for concurrent operations. The lock type is a numeric value and may be specified as a constant or as an integer variable. Lock values are:

0 - Null. The lock is not specified and takes the default (exclusive in updates, concurrent read in retrievals). Same as not specifying a lock clause.

1 - Exclusive. Same as 6.

2 - Concurrent Read. Anyone else may read or write this record. This process intends to read this record. This is the default in retrievals.

3 - Concurrent Write. Anyone else may read or write this record. This process intends to write this record.

4 - Protected Read. Anyone else may read this record. No-one may write this record. This process intends to read this record.

5 - Protected Write. Anyone else may read this record. No-one may write this record. This process intends to write this record.

6 - Exclusive. No-one else may read or write this record. This is the default in updates.

Changing Locks

Once a record or case has been retrieved, it is possible to alter the locktype held on that record with the `CASELOCK` and `RECLOCK` functions. Specify the new locktype on the function. If the change is successful, the record is written to the database and re-retrieved

with the new locktype and the function returns a 1. The function returns a zero if the change could not be made because of other locks on the record.

Lock Conflicts

During concurrent execution, the retrieval may encounter a record that is locked by another user.

Current Lock	Requested Lock					
	Null	1	2	3	4	5
EX	Locked	Locked	Locked	Locked	Locked	Locked
CR	Read	Locked	Write	Write	Write	Write
CW	Read	Locked	Write	Write	Locked	Locked
PR	Read	Locked	Write	Locked	Write	Locked
PW	Read	Locked	Write	Locked	Locked	Locked

If the case/record is locked (see table above) then:

- the current case/record variables are all set to undefined;
- a flag is set that can be tested with two functions: `SYSTEM(36)` for records and `SYSTEM(37)` for cases ;
- control is passed to the first statement in the block.

If the record is not available, the retrieval could wait to try accessing it again by using the `RETRY RECORD OR RESTORE REC` command. e.g.

```

PROCESS RECORD PATIENT LOCK = 4      | get the patient record
. LOOP
. IF(SYSTEM(36) = 1) EXIT LOOP      | exit if we get the record
. WRITE 'Waiting for locked record' at 24,5
. WAIT 5                            | wait half a second
. RETRY RECORD                       | try to get the record
. END LOOP
...
END PROCESS RECORD

```

LOOKUP

```
LOOKUP {RECORD dbname.recname | TABLE tabfile.table}
      [FORWARD | BACKWARD]
      [GET VARS { ALL|
                target_varlist|
                local_varlist = target_varlist}]
      [INDEXED BY indexname]
      [RESULT num_varname]
      [USING caseid,keylist | VIA keylist]
      [WHERE (condition)]
```

LOOKUP accesses a single database record or table row if one exists that matches keys and/or conditions and returns data as specified. The RECORD or TABLE clause must be specified. Unless further clauses are specified, the command does not achieve anything. The command may be specified in a PROGRAM, RETRIEVAL or SUBROUTINE at any point. It does not affect other database or table access processes.

RECORD [dbname.]recname TABLE [tabfile.]table	Specify either a record or table to use for the lookup. Specify the database or tabfile containing the record or row unless the default. The database or tabfile must be connected both at compile time and at execution time.
FORWARD BACKWARD	Specify either FORWARD or BACKWARD to control the direction of search. FORWARD is the default.
GET VARS ALL target_varlist local_varlist = target_varlist	Specify GET VARS clause to pass back values if found. The keyword ALL specifies all the matching record or table variables are assigned to local variables of the same name. A single list of variables creates a set of local variables with the same names as the database or table variable list. Note that ALL or a single list can only be used where table variables have valid local variable names. A list of local variables can be equated to a list of variables from the target record or table and the local variables are assigned the values of the database or table variables. The two lists must be of equal length and the value assignments are performed listwise.
INDEXED BY	Specify an index to use if necessary.

indexname

RESULT
num_varname

Specify a `RESULT` numeric variable to return positive for record found, negative for not found. A negative number is an error code and associated text can be retrieved with the `MSGTXT` function.

USING | VIA

Specify either `USING` or `VIA` to lookup using particular key values. On a case structured database, not using an index, `USING` specifies the *case* key first, then record keys. `VIA` specifies keys in sequence either from an index or from the current case. Where a key is specified, it is matched exactly. If all keys are not specified, the subset of records identified by the partial key is used. The values specified may be constants or expressions. If expressions use database record variables, these are from the current context not from the record being looked for.

WHERE
(condition)

Specify a `WHERE` condition to test prospective records (either the record that matched specified keys exactly or the subset identified by partial keys). The first to satisfy the condition is returned. Variables used in the condition clause may either be local or from the looked up record.

Accessing Tables

There are commands that create table rows, that update table variables and that access data stored in tables.

The structure and contents of tables and tabfiles is discussed in Tabfiles and Tables.

Tables in tabfiles may be accessed in any VisualPQL routine (program, retrieval or subroutine). Options on these routine commands affect tabfile and table processing.

Before a routine can be compiled or executed, the tabfile must be connected.

A program can connect a tabfile at execution time with the `PQL CONNECT TABFILE` command.

Table processing differs slightly from database record:

- Tables may have indexes that can be used to access the table rows. Accessing rows through an index determines the sequence in which the rows are retrieved.
- The only commands that deal directly with variables in a table are `GET VARS` and `PUT VARS`. When retrieving a row of a table, move the values of the variables into local variables with a `GET VARS`. Make any modifications to the local variables. To update the values of variables in a table row, move the local variables into the table row with a `PUT VARS`. Values of key fields of the index being used may not be updated with `PUT VARS`.
- The names of the variables, indexes and tables may be up to 32 characters long.

Use the `OPEN TABLE` and `CLOSE TABLE` to open and close tables. If these are not used, the tables are opened automatically when referenced.

Row Processing Commands

Row processing commands access a specific table. These commands define a block of commands that is delimited with the `END ROW` command. Retrieve required row variables using the `GET VARS` command to make the variables available to other VisualPQL commands within the block. There are two commands that process rows:

- `PROCESS ROW` retrieves a specific set of rows and updates these if required.
- `ROW IS` (and the variants `NEW ROW IS`, `OLD ROW IS`) retrieves or creates a single row.

Any updates to the table, including the creation of new rows, require the `TUPDATE` keyword either on the `RETRIEVAL` command or on the row processing command. Create new rows with a `ROW IS` or `NEW ROW IS` block. Existing rows may be accessed with the other types of row blocks.

Indexes

Tables can have *Indexes* that may uniquely identify a row or may identify a subset of rows. Options on the row block commands specify a subset of the rows by specifying an index and a range of index values. A table may have more than one index and more than one variable in an index. VisualPQL locates individual rows through the index.

Commands in ROW blocks

Any command, including other row, case and record block commands, may be used within a row block. The following commands may only be used in row blocks:

- `DELETE ROW` Deletes the current row.
- `EXIT ROW` Terminates processing of the row block.
- `NEXT ROW` Retrieves the next row in a `PROCESS ROWS` block.
- `PREVIOUS ROW` Retrieves the previous row in a `PROCESS ROWS` block.

ROW functions

Specify these functions after the `ROW IS` block to which they apply:

- | | |
|-------------------------|--|
| <code>SYSTEM(27)</code> | Returns a 1 if the last <code>ROW IS</code> , <code>NEW ROW IS</code> , or <code>OLD ROW IS</code> block was executed. It returns 0 (zero) if the last <code>ROW IS</code> block was not executed. |
| <code>SYSTEM(28)</code> | Returns a 1 if the last <code>ROW IS</code> or <code>NEW ROW IS</code> block created a new row. It returns 0 (zero) if the block did not create a new row in the table. |

OPEN TABLE

```
OPEN TABLE tabfile_name.table_name [ MODE mode_num ]
```

Opens the specified table.

Specify the tabfile name and table name either as variables that contain the name or as quoted strings. Ensure that the names have the correct use of upper and lower case letters as both are allowed in tabfile and table names and thus no automatic conversions are done.

The `MODE` clause specifies whether the table is opened for read or write access. Specify the `mode_num` as a numeric variable or constant. 1 specifies `READ` mode, 2 specifies `WRITE` mode. The default if `MODE` is not specified is `READ` mode.

If the tabfile is not connected or the table does not exist, a run time error is issued.

```
OPEN TABLE "TESTFILE"."TABLE1" MODE 2
```

CLOSE TABLE

```
CLOSE TABLE tabfile_name.table_name
```

Closes the specified table.

Specify the tabfile name and table name as variables that contain the name or quoted strings. Ensure that the names have the correct use of upper and lower case letters as both are allowed in tabfile and table names and thus no automatic conversions are done.

See also the `CLOSETABLE` option on `PROCESS ROW` and `ROW IS`.

PQL CONNECT TABFILE

```
PQL CONNECT TABFILE tabfile_name_exp
  [ FILENAME filename_exp | attribute_exp ]
  [ MODE {varname | constant} ]
  [ SECURITY exp,exp,exp,exp ]
  [ IOSTAT = varname ]
```

Connects the specified tabfile at execution time. All of the parameters are expressions; enclose names in quotes if specifying a constant. When assigning string values to expressions, ensure names are upper case.

tabfile name

The internal name of the tabfile. Must be the same name as used when the tabfile was created.

FILENAME

The name of the operating system file if different to the internal tabfile name plus the .tbf suffix.

MODE

Specifies if the tabfile is opened for READ or WRITE. If MODE is not specified, it is connected for READ. Specify 1 for READ, 2 for WRITE.

SECURITY

Specifies *Group Name*, *Group Password*, *User Name* and *User Password* in this order.

IOSTAT

Specifies a variable to receive the return code generated by the file open operation. A return code of 0 (zero) indicates successful connection.

-7001 (Host error message number) indicates that the tabfile could not be opened.

Note: Because this command connects the tabfile at execution time, the tabfile may not be connected at compile time. If there are subsequent references to the tabfile in this VisualPQL program then they may not compile. You need to connect the tabfile before compiling.

PQL DISCONNECT TABFILE

```
PQL DISCONNECT TABFILE tabfile_name_exp [IOSTAT = varname]
```

Disconnects a tabfile.

IOSTAT

Specifies a variable to receive the return code generated by the file close operation. A return code of 0 (zero) indicates successful disconnection as specified. -88 (DBMS error message number) indicates that the tabfile could not be disconnected.

DELETE ROW

DELETE ROW

Deletes the current row. To delete a row, use the `DELETE ROW` command in a `ROW IS` or `PROCESS ROW` block. This command may only be used in `TUPDATE` mode.

END ROW

```
END ROW [IS]  
END PROCESS ROW
```

Terminates ROW IS and PROCESS ROW blocks.

END ROW IS terminates ROW IS blocks.

END PROCESS ROW terminates PROCESS ROW blocks.

EXIT ROW

EXIT ROW

Terminates processing of the row block.

NEXT ROW

NEXT ROW

Retrieves the next row in a `PROCESS ROWS` block.

PREVIOUS ROW

PREVIOUS ROW

Retrieves the previous row in a `PROCESS ROWS` block.

PROCESS ROWS

```

PROCESS ROWS [ tabfile.]tablename
  [ INDEXED BY indexname]
  [ CLOSETABLE num_val ]
  [ COUNT = total [,increment [,start ]] ]
  [ ONETIME ]
  [ REVERSE ]
  [ SAMPLE= fraction [,seed]]
  [ UPDATE | TUPDATE]
  [ AFTER (value list) ]
  [ AFTER (value list) THRU (value list) ]
  [ AFTER (value list) UNTIL(value list) ]
  [ FROM (value list) ]
  [ FROM (value list) THRU (value list) ]
  [ FROM (value list) UNTIL(value list) ]
  [ THRU (value list) ]
  [ UNTIL (value list) ]
  [ VIA (value list) ]

```

Defines a row processing block for the specified table. The commands within the block (that is terminated with `END ROW`) are executed once for each row accessed. If a tabfile name is not specified, the default tabfile is used.

INDEXED BY	Names the index to use. If this is not specified, the records are read sequentially as stored on the table.
CLOSETABLE	Specifies whether the table is closed when the block is exited. A value of 0 (zero) or undefined leaves the table open and is the default. Any other value closes the table. When a table is closed, the memory used to hold the table is released for other use. Unless memory problems are encountered, avoid using this option. See also the <code>CLOSE TABLE</code> command.
COUNT	Specifies the number of rows to retrieve. The values for <i>total</i> , <i>increment</i> and <i>start</i> are integer constants.

total

Specifies the maximum number of rows to retrieve. If more rows are requested than are available, the program retrieves all that exist. e.g. To process the first 5 rows in the table:

```

PROCESS ROWS TRIALTF.TABLE1 COUNT = 5
increment

```

Specifies the "skipping factor" for retrieving rows. An increment of 3

retrieves every third row. The default increment is 1 (one). e.g. To access a total of 5 rows, retrieving every tenth row:

```
PROCESS ROWS TRIALTF.TABLE1
      COUNT = 5 , 10
start
```

Specifies the first row processed. The default start is 1, the first row. For example, 3 starts retrieving at the third row.

REVERSE

Processes the specified rows of the table in reverse order.

SAMPLE=fraction

Retrieves a random sample of rows from the table. The fraction specifies the portion of cases to select. The number specified is a decimal number between 0 (zero) and 1 (one). For each row, a random number between 0 and 1 is generated. If it is between 0 and the specified number, the row is retrieved. Each row is evaluated for inclusion independently, and therefore the sample may not be exactly the requested size particularly for tables with a small number of rows. Sampling is done before COUNT takes effect (i.e. "SAMPLE .5/ COUNT 2" retrieves the first 2 of a 50% sample). e.g. To process 25% of the rows in the table:

```
PROCESS ROWS TRIALTF.TABLE1
      SAMPLE = .25
```

seed

Specifies the starting seed for the random number generator. A given seed guarantees that the same set of random numbers is generated. If a seed is not specified, a default seed is used.

```
PROCESS ROWS TRIALTF.TABLE1
      SAMPLE = .25,13579
```

TUPDATE

Specifies that the program can update data in the rows of this table. Use the PUT VARS command to update the row from local variables. TUPDATE need not be specified on the PROCESS ROWS command if it has already been specified on the routine command. UPDATE is a synonym for TUPDATE.

ONETIME

Forces the PROCESS ROW block to be entered at least once, even if no rows within the specified range exist. If no rows exist, without this keyword, the block is skipped. The values of the row variables are set to undefined if the block is executed and no rows exist.

value list

A list of values expressed as constants, variable names or array references. Each element in the list represents a value for an index key field. The values are matched with values of keyfield variables in the order defined for the index.

The value list may omit lower level keys. If a key is omitted, no lower keys can be specified. During execution, if a value is undefined or

	missing, the value list is treated as if it were terminated with the keyfield previous to the undefined value. N.B. This differs from the behaviour in version 2.n. of the software where an execution warning was reported and the block skipped.
AFTER	Specifies the key value to start processing at but not to include. This selects rows whose key value is greater than the key specified by the value list. Specify <code>THRU</code> or <code>UNTIL</code> to select a range of keys.
FROM	Specifies the key value to start processing and to include. This selects rows whose key value is greater than or equal to the key specified by the value list. Specify <code>THRU</code> or <code>UNTIL</code> to select a range of keys.
THRU	Specifies the key value to process up to and to include in the retrieved subset. This selects rows whose key value is less than or equal to the key specified by the value list. Specify <code>AFTER</code> or <code>FROM</code> to specify a beginning row for processing.
UNTIL	Specifies the key value to process up to but not to include in the retrieved subset. This selects rows whose key value is less than the key specified by the value list. Specify <code>AFTER</code> or <code>FROM</code> to specify a beginning row for processing.
VIA	Selects records whose key value matches (equals) the key specified by the value list. If a partial key value list is specified, all records matching the partial list are selected. <code>WITH</code> is a synonym for <code>VIA</code> .

Using Indexes

The keywords `AFTER`, `FROM`, `THRU`, `UNTIL` and `VIA` specify a subset of rows by specifying values of the keyfields of the table.

If an index has been defined for the table and is referenced on the `INDEXED BY` clause, each row is identified by its *key*. The key is a composite of the values of the key fields as defined for the index.

VisualPQL locates individual rows through the index that points to the location of a row within the tabfile. The *value list* specified with the keywords on `PROCESS ROW` supplies values of the keys that VisualPQL uses to perform an indexed search for the records. The values in the list are matched to values of index key variables in the rows being processed.

The order of the values determines the keyfields to which the values refer. The keyfields and their order are passed to the `PROCESS ROW` list from the index definition.

For example, consider a table called `REVIEW` that has variables called `JOBCODE`, `BOSSNAME` and `RATING` and an index defined as:

```
CREATE UNIQUE INDEX REVIDX ON REVIEW (JOBCODE,BOSSNAME)
```

A `PROCESS ROW` to select those records of an employee who was reviewed for job 3 by Supervisor Jones might be as follows. The retrieval translates the values 3 and JONES as being values for keyfields `JOBCODE` and `BOSSNAME`:

```
PROCESS ROW REVIEW INDEXED BY REVIDX VIA (3, 'JONES')
```

If multiple keyfields are defined for the table index, leading keyfields must be specified. Trailing keyfields can be omitted, but intervening keyfields must be specified. If any entries in the list are out of range or contain missing or undefined values, the valid portion of the list up to the first undefined value is used. For example, if A, B, and C represent the keyfields on an index, then:

```
VIA (A, B, C)    is legal
VIA (A)          is legal
VIA (A, B)      is legal
VIA (, ,C)      is invalid, needs A and B
VIA (A, ,C)     is invalid, needs B
VIA (, B, C)    is invalid, needs A
```

ROW IS

```
[ OLD | NEW ] ROW IS [tabfile_name.]table_name
                        [ INDEXED BY index_name (value list) ]
                        [ TUPDATE ]
                        [ CLOSETABLE num_value ]
                        [ AT (block,pos) ]
```

The `ROW IS` commands access a single row from the specified table. In update mode rows can be modified or created. Specify update mode with the `TUPDATE` keyword on the `ROW IS` or on the routine command.

<code>tabfile_name.table name</code>	Specifies the table to access. This table must be connected at compile time and at run time.
<code>ROW IS</code>	Accesses the row specified by the index key specified on the key field value list. This value list is a list of values matched with key fields of the index named on the <code>INDEXED BY</code> clause. When in update mode, a new row is created if it does not exist. When not in update mode and when an index is not specified, the first record in the table is accessed.
<code>OLD ROW IS</code>	Accesses the first row of the table or the row specified by the <code>INDEXED BY</code> clause and its key field value list. If the specified row does not exist, the block is skipped. A new row is never created.
<code>NEW ROW IS</code>	Creates a new row. <code>NEW ROW IS</code> is only allowed in update mode. If the index specified is a unique index, the block is skipped if the row exists. If the index is not unique or if an <code>INDEXED BY</code> clause is not used, a new row is created if not restricted by another unique index on the table.
<code>TUPDATE</code>	Allows the table to be updated. This keyword is required on <code>ROW IS</code> blocks that update the table if <code>TUPDATE</code> is not specified on the <code>PROGRAM</code> or <code>RETRIEVAL</code> command. <code>UPDATE</code> is a synonym.
<code>INDEXED BY</code>	Names the index used for accessing rows.
<code>CLOSETABLE</code>	Specifies whether the table is closed when the block is exited. A value of 0 (zero) or undefined leaves the table open and is the default. Any other value closes the table. See also the <code>CLOSE TABLE</code> command.
<code>AT (block,pos)</code>	The <code>AT</code> can only be used with the <code>OLD ROW IS</code> construct. It retrieves a row that was previously retrieved from the saved block and position. These can be got when the row is initially retrieved with the <code>SYSTEM</code> functions 18 and 19. This allows the program to

exit a block and find the row again even where duplicate keys are allowed in the index.

ODBC Client

VisualPQL programs can access ODBC data sources (using ODBC) and can directly access the SIRSQL Server. The VisualPQL program establishes a connection id and a statement id that are the key identifiers for other operations. It then passes the text of an SQL query and executes this. The program can enquire as to the columns and rows available from the query and can get data from each column, stepping through the rows one at a time.

When using the SIRSQL server to do a query across more than one data source, the program establishes a connection to the multiple data sources with the same user name and password.

A program can have multiple connections open at one time. A connection can have multiple statements. Query results are by connection/statement.

Every command has an `ERROR` clause that returns a status that indicates success or failure. The functions return an error code. Further information about the error can be retrieved by the `GETERR` function.

CONNECT

```
CONNECT conid SERVER name {DATABASE name | TABFILE name}  
[USER name]  
[PASSWORD name]  
[PREFIX name] [UPDATE | READ]  
[ERROR errid]
```

Creates a connection to ODBC or to a SIR SQLserver.

`conid` is a numeric variable that returns an arbitrary number assigned by the system and subsequently used to identify the connection.

A `name` as used in various parts of the command is a string expression i.e. a string variable or a constant enclosed in quotes.

`SERVER name` is either the string ODBC (must be uppercase) or the TCP/IP address of the server.

`DATABASE name | TABFILE name` is the name of the data source as used by ODBC or the server. This is nothing to do with any databases or tabfiles directly connected by SIR/XS.

`USER name` has three possible components. The first is an arbitrary name used to identify that this connection is one of those associated with a single 'user' (i.e. this program) if a query is done across multiple connections. If a tabfile is being connected, the second and third components are used to specify any group and user name for that tabfile. Separate multiple components by commas.

This is typically blank for non-SIR ODBC sources.

`PASSWORD name` has four possible components. The first is a password for the connection associated with a single 'user' (i.e. this program) if a query is done across multiple connections. If a database is being connected, the second, third and fourth components are used to specify the database password, the read password and the write password. If a tabfile is being connected, the second and third components are used to specify any group and/or user passwords for that tabfile. Separate multiple components by commas.

This is typically blank for non-SIR ODBC sources.

`PREFIX` is the directory prefix for the SIRSQL server to find the database. If this is not fully qualified, the SQLSQL Server takes it to apply from its own local directory structure. If the server is set to ODBC then the prefix ignored as it is already specified in the data source setup.

UPDATE | READ allow/disallow SQL statements that update the data source. The default is read.

ERROR `errid` is a numeric variable that returns a status code. A value of zero or less indicates that the connection failed.

DISCONNECT `conid` [ERROR name]

Disconnects the connection.

Statement

```
STATEMENT statid CONNECT conid [ERROR name]
```

Creates an arbitrary statement number for a connection that is subsequently used to identify the statement.

`statid` is a numeric variable that returns the statement number.

```
DELETE STATEMENT statid CONNECT conid [ERROR name]
```

Deletes a statement

```
PREPARE STATEMENT statid CONNECT conid  
{COMMAND text_expression | BUFFER buffer_name}  
[ERROR name]
```

Sends the text of an SQL statement. This can either be a string expression (e.g. a string variable in the VisualPQL) that contains text up to 254 characters or it can be the name of a buffer that contains the text of a long SQL statement (up to 4K).

```
BIND STATEMENT statid CONNECT conid (param_no,value)  
[ERROR name]
```

SQL queries may contain parametrised values, that is the value is not specified directly in the query but separately via a `BIND` command. A parameter is shown in the SQL query as a question mark ? e.g.

```
SELECT * FROM EMPLOYEE WHERE ID EQ ?
```

If a statement has multiple parameters, they are identified positionally, that is the first question mark is parameter 1, the second 2, etc.

The `BIND` supplies values for the parameters. Values may either be numeric or string expressions. e.g.

```
BIND STATEMENT statid CONNECT conid (1,10)
BIND STATEMENT statid CONNECT conid (2,'John')
```

Parameters may be bound before or after a statement is prepared. Parameters may also be bound using the `BINDPARM` function.

Because the type (numeric or string) of the parameter is not known at compile time, make sure it matches the data type of the variable that it applies to. Data for string, categorical vars, dates and times must be string expressions.

```
EXECUTE STATEMENT statid CONNECT conid [ERROR name]
```

This runs the prepared statement and produces a set of output. This might take some time depending on the size of the data source and the query.

The output can be examined using the VisualPQL Client/Server functions.

Example

```
PROGRAM
INTEGER*4 errid conid statid rnum cnum
STRING*20 cname colval
STRING*80 qtext errstr
CONNECT conid SERVER 'ODBC'
        DATABASE 'Company'
        USER      'me'
        PASSWORD  'mypwd,COMPANY,HIGH,HIGH'
        PREFIX    ''
        ERROR     errid
STATEMENT statid CONNECT conid ERROR errid
WRITE errid
PREPARE STATEMENT statid CONNECT conid
        COMMAND 'SELECT * FROM EMPLOYEE'
        ERROR errid
WRITE errid
EXECUTE STATEMENT statid CONNECT conid ERROR errid
WRITE errid
COMPUTE cnum = COLCOUNT (conid,statid)
COMPUTE rnum = ROWCOUNT (conid,statid)
WRITE 'Columns returned ' cnum ' Rows returned ' rnum
FOR I = 1,cnum
. COMPUTE cname = COLNAME (conid,statid,i)
. WRITE cname
END FOR
```

```
SET J (0)
LOOP
. COMPUTE j = j+1
. COMPUTE res = NEXTROW (conid,statid)
. IF (res LE 0) EXIT LOOP
. FOR I = 1,cnum
. IFTHEN (COLTYPE(conid,statid,i) eq 1)
.   COMPUTE colval = COLVALS (conid,statid,j,i)
. ELSE
.   COMPUTE colval = FORMAT (COLVALN (conid,statid,j,i))
. ENDIF
. WRITE colval
. END FOR
END LOOP
DELETE STATEMENT statid CONNECT conid
DISCONNECT conid
END PROGRAM
```

Graphical User Interface

VisualPQL provides the tools to build a portable and flexible graphical user interface for your applications.

The top level of the graphical interface is created by a main program that defines the main window and menu system including all sub-menus. This receives control when a menu item is selected and can perform an action directly or can invoke other VisualPQL programs or SIR/XS commands.

Programs can create dialogs that pass control back to the user and receive control back when the user takes an action such as pressing a button.

The member `SYSTEM.START` in the system procedure file is the main window program. Control is returned to this program until it issues an `EXIT` message and the system then stops. (N.B. If a different procedure file is specified with the `PROC=` execution parameter, it must contain the `SYSTEM` family and `START` member.)

Any VisualPQL program can output information into the main window (such as title and status) and can put text in the window. Programs can also save, print or clear the main window.

Once the system is running, programs can display and get information through dialogs. The system has to be active for graphical commands to be executed. Any VisualPQL can be compiled when the graphical interface is not active (e.g. in batch), but running a program that uses graphical commands without the main window active returns an error.

Predefined programs can be invoked through menus and any program can create and run with dialogs. A program that creates a dialog remains active until it issues an `EXIT` command.

WINDOW

Window and menu definition commands and functions can only be used in the initial main program that creates the main window and menus.

WINDOW

Defines the main window and the start of the system.

END WINDOW

Defines the end of the main window and exits the system.

WINDOW TITLE

```
WINDOW TITLE text_var
```

Sets title for main window.

This command can be executed from any program running while the main window is active i.e. It is not restricted to running in the program that defines the main window.

WINDOW STATUS

```
WINDOW STATUS LINE text_var
```

Puts a message into the status bar in the main window.

This command can be executed from any program running while the main window is active i.e. It is not restricted to running in the program that defines the main window.

WINDOW OUTPUT

WINDOW OUTPUT *text_var* [AT CURSOR] [HTML]

Writes text to the scrolled main window. Each output line is written to the next line in the window. Specify the keyword `AT CURSOR` to write to the cursor position (e.g. after selecting some text). This overwrites any text following the cursor.

Specify the keyword `HTML` to write text formatted using a subset of standard html tags. Supported tags are:

```
&nbsp; ;
&quot; ;
&amp; ;
&lt; ;
&gt; ;

<br>
<li>
<p>
  align
  center
  right
  left
<body>
  bgcolor
  tran[sparent]
  #rrggbb
  readonly
  true
  FALSE
  wrap
  off
  ON
<h1>
<h2>
<h3>
<h4>
<h5>
<h6>
<b>
<u>
<i>
<sub>
<sup>
```

```
<big>
<small>
<code>
<pre>
<center>
<ul>
<ol> (same as ul)
<font>
  color
    #rrggbb
  size
    [+|-]n
  face
    fontname
```

The `WINDOW OUTPUT` command can be executed from any program running while the main window is active i.e. It is not restricted to running in the program that defines the main window. Note that output from a standard `WRITE` command that does not specify a filename also goes to the scrolled main window when running without a default output file.

WINDOW CLEAR

WINDOW CLEAR

Clears the scrolled main window.

This command can be executed from any program running while the main window is active i.e. It is not restricted to running in the program that defines the main window.

WINDOW SAVE

WINDOW SAVE filename

Saves the scrolled contents of main window as a file.

This command can be executed from any program running while the main window is active i.e. It is not restricted to running in the program that defines the main window.

MENU

MENU name

Defines a menu block. A subsequent MENU defines a sub-menu. The name is a string variable or string constant. To define a mnemonic letter precede it by an &. The top level menu is a horizontal menu, all sub-menus are vertical menus. All menu definition must be within the WINDOW block.

END MENU

Defines the end of a menu or sub-menu.

MENUITEM

```
MENUITEM id,name_var,accel_ch,style
```

Defines a menu item. When this item is selected from the menu by the user, the `id` is passed to the processing logic and so it must be unique within all menu items. The `name` is a string variable or string constant and can contain an `&` to define a mnemonic letter.

`accel_ch` is a one character string variable or string constant containing a single accelerator character. Blank means there is no accelerator. A lower case letter is `CTRL+letter`; an uppercase letter is `CTRL+SHIFT+letter`. (N.B. This may be machine specific.)

`style` is an integer variable - 0 for not checkable; 1 for checkable. Checkable means that a menu item can be toggled between two states (checked and unchecked) by the `CHECK ITEM,UNCHECK ITEM` commands and tested by function `GETMCHK`. When a menu item is checked it has a visual indication (a tick) next to it.

Note The main window can be closed by the user without using a VisualPQL menu item. When this happens (by using a system facility), the system passes a zero as the `id`. A message zero must have the same effect as any menu 'Exit' logic and so use zero as the `id` for the menu item associated with exit.

MENUSEP

MENUSEP

Defines a menu separator. This is only valid in pulldown menus and appears as a horizontal line.

TBARITEM

```
TBARITEM id,bitmap,tip_text,style
```

Defines an item on the toolbar.

`Bitmap` is the name of the bitmap that is the image of the toolbar button.

`tip_text` is a string variable or constant that is text displayed when the cursor is on the toolbar. Empty tips are not displayed. These can be defined in any sequence at any convenient point and the definition sequence determines the sequence across the toolbar.

`style` is an integer variable - 0 for not checkable; 1 for checkable. Checkable means that a menu item can be toggled between two states (checked and unchecked) by the `CHECK ITEM`, `UNCHECK ITEM` commands and tested by function `GETMCHK`. When an item is checked, it has a visual indication (button depressed).

`id` is an integer value passed to the message procedure. If the `id` is the same as a `MENUITEM` `id` both must be the same style (checkable, regular) and they are kept synchronised. (i.e. checking is on or off for both). It is strongly recommended that all toolbar items are also menu items.

TBARSEP

TBARSEP

Defines a separator (small space) on the toolbar.

INITIAL

INITIAL

Defines an initial *message processing block* for the menu.

At this point, the system starts and the commands in the initial block are executed. Processing from this point on is through the message blocks.

MESSAGE

```
MESSAGE[S] COMMAND id_var  
MESSAGE DROPFILE string_var  
NEXT MESSAGE  
EXIT MESSAGE  
END MESSAGE[S]
```

If a particular type of message processing block is not defined, then messages of that type are simply ignored.

The `MESSAGE COMMAND` processing block is the normal message block for menus. This gets control when the user selects a lowest level menu item. The `id` of the selected menu item is passed to the block in the variable named `id_var`.

The `MESSAGE DROPFILE string_var` receives control if the user 'drags and drops' a file into the main window. The name of the file is passed to the block in the `string_var` variable.

`NEXT MESSAGE` returns control to the user.

`EXIT MESSAGE` finishes processing and, in a menu, stops the SIR/XS session.

`END MESSAGE` finishes the definition of a message processing block. If the command is reached during execution, it returns control to the user.

ENABLE MENUITEM
DISABLE MENUITEM

```
ENABLE MENUITEM id  
DISABLE MENUITEM id
```

Menu items are enabled by default. These commands enable (make selectable) or disable (grey out) a specified item.

These commands can be executed from any program running while the main window is active i.e. they are not restricted to running in the program that defines the main window.

CHECK MENUITEM
UNCHECK MENUITEM

```
CHECK MENUITEM id  
UNCHECK MENUITEM id
```

Menu items are unchecked by default. These commands check (tick) or uncheck a specific item.

These commands can be executed from any program running while the main window is active i.e. they are not restricted to running in the program that defines the main window.

The function `num = GETMCHK (id)` tests the state of a menu or toolbar item. Returns 1 if checked; 0 if unchecked.

DISPLAY POPUP LIST

```

DISPLAY POPUP LIST [(string_exp,string_exp,...) |
array_name,no_of_items]
    ANCHOR id | AT row,col
    [POSTYPE num_exp]
    RESPONSE num_varname
    [SEPARATOR (n,n...)]

```

Displays a pop up menu that remains on the screen until the user either selects an item or clicks at a point off the menu, thus canceling the menu. This command can be issued in menus, in standard dialogs or in DEDIT dialogs.

<code>(string_exp,...) array_name,no_of_items</code>	The menu consists of a list of entries. These can be defined either as a list of string expressions in parentheses or as a string array name followed by the number of items to be selected from the array. The menu is built with one line per item.
<code>ANCHOR id AT row,col</code>	The menu is either positioned relative to the control identified by the ANCHOR id or at the position specified by the AT clause. If the ANCHOR id is specified, the position of the menu is determined by the appearance of the current screen and the menu is positioned relative to the item referenced by that id in the current menu or dialog.
<code>POSTYPE num_exp</code>	If POSTYPE is specified with a numeric expression that evaluates to 1, then values specified in the AT clause are absolute positions; otherwise, the AT clause specifies standard row/col based positions.
<code>RESPONSE num_varname</code>	Specifies a numeric variable that is set to the position in the menu list of the item selected. If no item is selected, the response variable is set to -1.
<code>SEPARATOR n,n...</code>	If specified, displays a menu separator after the nth item(s).

DIALOG

```
DIALOG title  
END DIALOG
```

Defines a dialog. All dialog definition commands and message processing blocks must be contained in the `DIALOG` block. The `TITLE` must be specified and is a text variable or string in quotes that is displayed in the title bar of the dialog.

Dialogs have two parts:-

1) The dialog definition that specifies the dialog controls, their positions and order on the dialog. Controls are labels, buttons, edit fields, etc. that are the visual items that make up the dialog and allow the user to interact with the system. Each control is identified by a unique numeric **id**, that is used to reference the control during the execution of the dialog. Controls that are not referenced during execution (e.g. labels, lines) can use -1 as the id.

The following commands can be used to define dialog controls:

- POSTYPE
- BORDERS
- BUTTON
- CHECK
- CHOICE
- COMBO
- EDIT
- IMAGE
- LABEL
- LINE
- LIST
- RADIO
- SLIDER
- SPIN
- PROGRESS
- TEXT
- TREE

2) The dialog message processing routines that can be executed each time a message **event** takes place. Events include pressing a dialog button, pressing a character key in a text field, selecting an item in a menu etc.

If a message event occurs and there is no message processing routine for that type of event, it is ignored.

The optional `INITIAL` message processing routine is executed before the dialog appears on the screen.

Message processing routines can perform any appropriate function including creating a new sub-dialog and executing standard SIR/XS commands with the `EXECUTE DBMS` command.

Control is returned to the user by a `NEXT MESSAGE` command or by reaching the end of the message processing routine (the `END MESSAGE` command). This retains the dialog on the screen.

The `EXIT MESSAGE` command deletes the dialog from the screen and performs any commands that follow the `END DIALOG` command. When the end of the program is reached, control is returned to any higher level dialog or to the menu system.

If the user exits the dialog using the windowing system (e.g. by clicking on the X button in Windows), a button message is generated with an id of 0 (zero) so ensure that `MESSAGE BUTTON 0` exits the dialog.

BORDERS

BORDERS

Specifies that the borders of labels, check boxes or radio buttons are displayed. Is only used when designing a new dialog to establish the positions of controls and to ensure they do not overlap.

Cannot be switched off once turned on for a dialog.

POSTYPE

POSTYPE (0,1)

Positioning of dialog controls uses a vertical and horizontal system of co-ordinates starting at 0,0 in the top left. The horizontal units are 1/4 of the average width for the font being used. The vertical units are by row or by absolute units. The `POSTYPE` command changes the vertical units from row to absolute during definition (0 is by row and is the default; 1 is by absolute units). Absolute vertical units are 1/8 font height. For precise positioning, dialog metrics can be retrieved with functions.

BUTTON

`BUTTON id, row, col, width, default, text`

Defines a dialog button. This is a rectangle one row deep that can be clicked by the user to indicate an action to take. This generates a message event to be handled by a message processing routine.

`ID` is a numeric variable or constant that identifies the control.

`ROW` and `COL` are numeric variables or constants that define the starting position of the control.

`WIDTH` is a numeric variable or constant that defines the length of the control in horizontal units.

`DEFAULT` is a numeric variable or constant. A value of 1 makes this the default button.

`TEXT` is a string variable or string in quotes and is displayed in the button. To define a mnemonic letter precede it by an `&`. The button label is set to this value at definition time. If you wish to change the label during message processing, use the `SET ITEM` command.

CHECK

```
CHECK id,row,col,width,text
```

Defines a dialog check control. This is a small square box that the user can toggle (turn on and off) followed by a label.

`ID` is a numeric variable or constant that identifies the control. `ROW` and `COL` are numeric variables or constants that define the starting position of the control.

`WIDTH` is a numeric variable or constant that defines the length of the control in horizontal units.

`TEXT` is a string variable or string in quotes. The label is set to this value at definition time. If you wish to change the label during message processing, use the `SET ITEM` command.

Use `CHECK ITEM` or `UNCHECK ITEM` to set the control and the `GETCHK` function to test the setting (0 for unchecked, 1 for checked).

CHOICE

`CHOICE id,row,col,width`

Defines a dialog choice control. A choice control is a box in a single row with an associated pull down list. This contains values set by the program and the user can choose one of these. A single value from the list is displayed in the box.

`ID` is a numeric variable or constant that identifies the control.

`ROW` and `COL` are numeric variables or constants that define the starting position of the control.

`WIDTH` is a numeric variable or constant that defines the length of the control in horizontal units.

Set the values for the choice control in a message processing block (e.g. in the `INITIAL` routine) with the `APPEND ITEM` or `INSERT ITEM` commands or manipulate the values with the `REMOVE ITEM`, `REMOVE ALL` and `SWAP ITEM` commands. Use the `GETPOS` function to establish the position of a choice the user has selected or the `GETTXT` function to return the text value. If the values are numeric, the `GETFLT` or `GETINT` functions can also be used.

By default, the first item in the list is displayed in the box. Use the `SELECT ITEM` command to select a different item to display.

EDIT

```
EDIT id,row,col,width,pass,read
```

Defines a dialog edit field. This is a box on a single row and can contain text that can be set by the program and possibly entered by the user. The user can scroll horizontally if necessary to enter or view more text than can be displayed in the box.

ID is a numeric variable or constant that identifies the control.

ROW and COL are numeric variables or constants that define the starting position of the control.

WIDTH is a numeric variable or constant that defines the length of the control in horizontal units.

PASS is a numeric variable or constant. If this is 1 then the edit field is password protected and the data is not displayed but replaced by asterisks (this may vary on certain operating systems).

READ is a numeric variable or constant. If this is 1 then the data in the control is read only and the user cannot enter data.

Set the value of an edit control with the SET ITEM command and retrieve data with the GETTXT function (if the control is supposed to return numeric values, the GETFLT or GETINT functions can also be used.)

COMBO

combo id, row, col, width

A *ComboBox* control is a combination of an edit and a choice control. It is used when free text can be entered but a set of predefined values also exists.

ID is a numeric variable or constant which identifies the control.

ROW and COL are numeric variables or constants which define the starting position and depth of the control in vertical units.

WIDTH is a numeric variable or constant which defines the length of the control in horizontal units.

You can use the same commands and functions as with an edit control or choice control.

SPIN

SPIN *id, row, height, col, width*

A *Spin* or *Up/Down* control displays a numeric edit box with a pair of arrows. The up arrow increments the value in the edit control and the down arrow decrements it.

ID is a numeric variable or constant which identifies the control.

ROW, HEIGHT and COL are numeric variables or constants which define the starting position and depth of the control in vertical units.

WIDTH is a numeric variable or constant which defines the length of the control in horizontal units.

SELECT ITEM and SET ITEM work on the spin control as do the functions SETPOS GETTXT GETINT and GETPOS

IMAGE

`IMAGE id, row, height, col, width, border`

Defines a rectangular space for a bitmap image.

`ID` is a numeric variable or constant that identifies the control.

`ROW`, `HEIGHT` and `COL` are numeric variables or constants that define the starting position and depth of the control in vertical units.

`WIDTH` is a numeric variable or constant that defines the length of the control in horizontal units.

Set the actual bitmap into the control in a message processing block (e.g. in the `INITIAL` routine) with the `SET IMAGE` command.

LABEL

```
LABEL id,row,col,width,text
```

Defines a dialog label that is a single row of text.

ID is a numeric variable or constant that identifies the control. If you specify -1 as the id, the label text cannot be referenced (retrieved or modified) during the execution of the program.

ROW and **COL** are numeric variables or constants that define the starting position of the control.

WIDTH is a numeric variable or constant that defines the length of the control in horizontal units.

TEXT is a string variable or string in quotes. The label is set to this value at definition time. If you wish to change the label during message processing, use the **SET ITEM** command.

LINE

`LINE id,row,height,col,width`

Draws a box or line. The horizontal lines in a box are offset vertically so that they can enclose other controls if necessary. However a line (height of 1) takes a standard row position.

`ID` is a numeric variable or constant that identifies the control.

`ROW`, `HEIGHT` and `COL` are numeric variables or constants that define the starting position and depth of the control in vertical units.

`WIDTH` is a numeric variable or constant that defines the length of the control in horizontal units.

LIST

`LIST id, row, height, col, width, type`

Defines a dialog list control. A list control is a box in multiple rows. This contains values set by the program and the user can choose one (or more) of these. Selected items are highlighted. If there is insufficient room to display all values, the user can scroll vertically.

`ID` is a numeric variable or constant that identifies the control.

`ROW`, `HEIGHT` and `COL` are numeric variables or constants that define the starting position and depth of the control in vertical units.

`WIDTH` is a numeric variable or constant that defines the length of the control in horizontal units.

`TYPE` is a numeric variable or constant with the value 0, 1 or 2. These define the type of selection that the user can make from the list control as follows:-

- **0** Single - One and only one item can be selected. If the user clicks on a second item, the first is deselected.
- **1** Extend - Usually one item but more can be selected. If the user simply clicks on a second item, the first is deselected. However the user can select multiple items by using the shift key when selecting.
- **2** Multiple - Multiple items can be selected. If the user clicks on a second item, it is selected. If the user clicks on a previously selected item, it is deselected.

Set the values for the list control in a message processing block (e.g. in the `INITIAL` routine) with the `APPEND ITEM` or `INSERT ITEM` commands or manipulate the values with the `REMOVE ITEM`, `REMOVE ALL` and `SWAP ITEM` commands. Use the `GETPOS` function to establish the position of a choice the user has selected and the `GETITXT` function to return the text value. If the values are numeric, the `GETIFLT` or `GETIINT` functions can also be used.

RADIO

```
RADIO id,row,col,width,text
```

Defines a dialog radio control. A radio button is very similar to a check box except that it is round and that it may be in a group. A series of radio definitions without any other type of control constitutes a group. In a group of radio buttons, if the user checks one button, then automatically any other checked button is unchecked.

`ID` is a numeric variable or constant that identifies the control.

`ROW` and `COL` are numeric variables or constants that define the starting position of the control.

`WIDTH` is a numeric variable or constant that defines the length of the control in horizontal units.

`TEXT` is a string variable or string in quotes. The label is set to this value at definition time. If you wish to change the label during message processing, use the `SET ITEM` command.

Use `CHECK ITEM` or `UNCHECK ITEM` to set the control and the `GETCHK` function to test the setting (0 for unchecked, 1 for checked). If the program checks a radio control in a group, there is no automatic unchecking of other controls in the group.

SLIDER

`SLIDER id, row, height, col, width`

Defines a dialog slider control. A slider control is a horizontal representation of a percentage scale (0 to 100) that the user can move left or right to indicate an increase/decrease. It can be set and moved programmatically.

`ID` is a numeric variable or constant that identifies the control.

`ROW` and `COL` are numeric variables or constants that define the starting position of the control.

`HEIGHT` is a numeric variable or constant that defines the height of the control in vertical units.

`WIDTH` is a numeric variable or constant that defines the length of the control in horizontal units.

Set the values for the slider control in a message processing block (e.g. in the `INITIAL` routine) with the `SET ITEM` using values between 0 and 100. Use the `GETPOS` function to establish the position of a slider the user has manipulated.

PROGRESS

A *Progress* control displays a read only progress meter.

PROGRESS *id*, *row*, *height*, *col*, *width*

ID is a numeric variable or constant which identifies the control.

ROW, *HEIGHT* and *COL* are numeric variables or constants which define the starting position and depth of the control in vertical units.

WIDTH is a numeric variable or constant which defines the length of the control in horizontal units.

If *Height* is greater than *width* then the bar is drawn vertically, otherwise it is horizontal.

Use the *SETRANGE* function to define the maximum and minimum values for the bar and *SETPOS* and *GETPOS* to set and get the position of the progress bar within the control.

The Progress bar is a read only control and does not send any messages to the dialog program.

TEXT

```
TEXT id,row,height,col,width,read
```

Defines a multi-line text control. This is a rectangular box that can display text and can allow the user to edit text. The user can scroll horizontally and vertically as necessary.

ID is a numeric variable or constant that identifies the control.

ROW, HEIGHT and COL are numeric variables or constants that define the starting position and depth of the control in vertical units.

WIDTH is a numeric variable or constant that defines the length of the control in horizontal units.

READ is a numeric variable or constant with values 0 or 1. 0 allows the user to edit data; 1 means that the data is read only and cannot be edited.

Set the values for the text control in a message processing block (e.g. in the INITIAL routine) with the APPEND LINE OR INSERT TEXT commands. Use the GETLTX function to return the text values line by line.

TREE

```
TREE id,row,height,col,width,read
```

Defines a tree control that is a window that displays a hierarchical list of items, such as the headings in a document, the entries in an index, or the files and directories on a disk. This is a rectangular box that can display text and can allow the user to edit text. The user can scroll horizontally and vertically as necessary.

ID is a numeric variable or constant that identifies the control.

ROW, HEIGHT and COL are numeric variables or constants that define the starting position and depth of the control in vertical units.

WIDTH is a numeric variable or constant that defines the length of the control in horizontal units.

READ is a numeric variable or constant with values 0 or 1. 0 allows the user to edit data; 1 means that the data is read only and cannot be edited.

Set the values for the tree control in a message processing block (e.g. in the INITIAL routine) with the BRANCH function. Use the information passed by the standard message processing block to identify nodes selected by the user.

The following commands and functions are used with tree controls:

BRANCH Adds a new node to a tree. The node is added as a child node of the given parent node.

BRANCHD Deletes the node from a tree.

NBRANCH Returns the number of child nodes of the given node.

BRANCHN Returns the node number of the nth child nodes of the given node.

The following list commands also work on tree controls. Note that pos is not the ordinal position of the tree item but the user supplied node number.

```
SELECT ITEM id,pos
REMOVE ITEM id,pos
REMOVE ALL id
GETTXT(id)
GETITXT(id,pos)
GETPOS(id)
```

Dialog Message Processing

```
        INITIAL
END INITIAL
```

If you specify an `INITIAL` message processing routine, it is executed before the dialog appears on the screen. This can be used to populate lists and set the initial default state of controls.

```
MESSAGE[S] type id, arg1, arg2
END MESSAGE[S]
```

Messages are generated by a user action and are passed to the appropriate message processing block. If there is no appropriate message processing block, the message is ignored.

`TYPE` determines which messages are processed by the block and is one of the following keywords:

`ALL`, `BUTTON`, `CHECK`, `CHOICE`, `EDIT`, `LIST`, `RADIO` and `TEXT`.

If `ALL` is specified then do not specify any other message processing blocks. Otherwise you may specify one of each type. Specify either `ALL` or `BUTTON` and include appropriate logic to exit the dialog when a button message with an id of zero is received.

Every message processing routine specifies a numeric variable that is set to the `id` of the control that generated the message.

`arg1` and `arg2` are variables on some types of messages. Ensure that the correct number of variables (none, one or two) are specified for the appropriate message type. The values in the specified variables are set depending on the type of message processing block and contain frequently needed data for the type of control. However, this data can also be retrieved by functions if necessary (e.g. when using an `ALL` message processing routine). The following describes the arguments passed for each message type:-

`ALL id, position, dbl_click` Position is a numeric variable and is set to the position in a list or choice if appropriate. `Dbl_click` is a numeric variable that is set to 0 or 1 where 0 means a single mouse click and 1 means a double mouse click.

`BUTTON id` Has no further arguments.

`CHECK id, check` Check is a numeric variable that is set to 0 or 1 where 0 means unchecked and 1 means checked.

`CHOICE id, position` Position is a numeric variable and is set to the position in the choice list.

`LIST id, position, dbl_click` - Position is a numeric variable and is set to the position in the list. `Dbl_click` is a numeric variable that is set to 0 or 1 where 0 means a single mouse click and 1 means a double mouse click.

`RADIO id, check` Check is a numeric variable that is set to 0 or 1 where 0 means unchecked and 1 means checked.

`EDIT id, hastext` Hastext is a numeric variable that is set to zero if there is no text in the control and to a positive value if there is text.

`TEXT id, line, position` Line and position are set to the line and position of the cursor when text is entered. Line is set to zero if there is no text in the control.

Once the appropriate actions have been taken, control is returned to the dialog either by issuing a `NEXT [MESSAGE]` command or by reaching the end of the block. This dialog remains active until a message processing block issues a `EXIT [MESSAGE]` command that closes the dialog.

Other Message Types

There are three message types that are not processed by `MESSAGE ALL` and you must specify the message processing block explicitly if you need to process these messages. The messages are:

`MESSAGE FOCUS id`
`MESSAGE HELP id`
`MESSAGE TIMER`

`MESSAGE FOCUS id` This message is generated every time focus moves off a dialog item. `id` is the id of the control moved away from. Use the `GETFOCUS` function to return the id of control moved to.

`MESSAGE HELP id` When this message is specified, a small `?` is displayed in the top right corner of the dialog. If the user clicks on this, it becomes a floating `?` and the user can position this to a control to request help. When the user clicks again, a message is passed to the `MESSAGE HELP` block to display appropriate help for the identified control.

`MESSAGE TIMER` This message processing block receives messages that are automatically generated. This could be used to refresh the display of some image or animation.

Use the `ENABLE TIMER n` command to start automatic generation of timer messages every `n` tenths of a second. Use the `DISABLE TIMER` command to stop generation of timer messages.

Dialog Control Commands

The following commands can be used within a message processing block while a dialog is active. *id* is a numeric variable or constant. *pos* is a numeric variable or constant.

```
ENABLE ITEM id  
DISABLE ITEM id
```

Enables and disables (greys out) a control.

```
FOCUS ITEM id
```

Sets the focus on to the control.

```
SHOW ITEM id  
HIDE ITEM id
```

These two commands alter the appearance of the dialog while it is active. Items can be hidden and other items shown. Items can thus appear to be on different pages within a dialog or emulate tabbed dialogs.

```
SET DIALOG TITLE string
```

Sets the dialog title, enabling different pages within a dialog to have appropriate titles.

```
CHECK ITEM id  
UNCHECK ITEM id
```

Checks a radio button or check box. If the user checks a radio button, all others in that group are unchecked by the system. If a radio button is checked by a program, the program must uncheck all others in the group.

```
SET ITEM id,var
```

The *var* may be text, integer or floating point and the command sets this value as the label for label, button, check and radio and as data for edit. Sets a multi-line edit control to one line containing the value specified in the variable (that may be a null string).

```
SET ITEM FONT id,bold,italic,underline,size,face
```

This command changes the appearance of text in a label, edit, button, list or text box. The *bold*, *italic* and *underline* can be zero or one to turn these font attributes off or on. The *size* is zero or +/-1. 1 makes the size of the font larger than current, -1 makes the size of the font smaller than current and 0 does not change the size (executing the command several times with a +/-1 changes the size progressively). The *face* is a string and can be either a font name or a colour code. The colour codes are in the form [#RRGGBB][/#RRGGBB] where the first code sets the foreground colour and the second code sets the background colour. Specify the colour using exactly six characters; valid characters are 0 to 9 and A to F. These are three sets of hexadecimal specifications of the strength of the red, green and blue components of the colour. Each

setting has a value from 00 to FF. Either component can be omitted completely. Execute the command twice to specify both a font name and colour. Note that you cannot specify the colour of a button.

Example: SET ITEM FONT IDTEXT,1,0,0,0,"#FF0000/#FFFFFF" sets the font of IDTEXT to be bold with foreground red and background white.

```
SELECT ITEM id,pos
```

Selects an item in a list. (Unselects other item in choice or list if single selection)

```
CLEAR SELECT ITEM id,pos
```

Clears selection.

```
SELECT ALL id
```

Selects all items in multiple selection.

```
CLEAR ALL id
```

Clears selection for all items in multiple selection.

```
APPEND ITEM id,var
```

Adds an item to choice or list. The *var* can be text, integer or floating point.

```
INSERT ITEM id,pos,var
```

Inserts an item at position in choice or list. The *var* can be text, integer or floating point.

```
SWAP ITEM UP,DOWN id
```

Swaps current item in list with one above.

```
REMOVE ITEM id,pos
```

Removes an item from choice or list.

```
REMOVE ALL id
```

Removes all items from choice or list.

```
SET IMAGE id,filename [type]
```

Puts an image from graphical file (windows or OS/2 bitmap) into a defined image or button control. If the type number is specified for an image control it can either be 1 or 2 to centre or resize the image to fit the control.

```
APPEND LINE id,var [HTML]
```

Adds a text line to a multi-line text control. Specify the keyword HTML to write text formatted using a subset of standard html tags. Supported tags are listed on the WINDOW OUTPUT command.

```
INSERT TEXT id,var
```

Inserts text into multi-line control at cursor position. The *var* can be text, integer or floating point.

Other GUI Commands

`BEEP`

Issues a short beep (displaying an error box beeps automatically).

`DISPLAY TIPBOX str`

Pops up a message. `str` is a string variable or string constant in quotes that is the message to display.

`DISPLAY INFOBOX str`

Displays a message in a dialog with an OK button. `str` is a string variable or string constant in quotes that is the message to display.

`DISPLAY ERRBOX str`

Displays an error message. `str` is a string variable or string constant in quotes that is the message to display.

`DISPLAY OKCANBOX str RESPONSE var`

Displays a message and asks for an OK or Cancel response. `str` is a string variable or string constant in quotes that is the message to display. `var` is a numeric variable that receives 1 (OK) or 0 (Cancel).

`DISPLAY YESNOBOX str RESPONSE var`

Displays a message and asks for a Yes or No response. `str` is a string variable or string constant in quotes that is the message to display. `var` is a numeric variable that receives 1 (Yes) or 0 (No).

`DISPLAY YNCBOX RESPONSE var`

Displays a Yes, No, Cancel Box. `var` is a numeric variable that receives 1 (Yes) or 0 (No) or -1 (Cancel)

`DISPLAY TEXTBOX label [SECRET] RESPONSE var, mess_text`

Displays a text input box. `label` is a string variable or string constant in quotes that is displayed as the title on the box. This normally indicates to the user what is being asked for. `var` is a numeric variable that receives -1 (Cancel) or length of string. `mess_text` is a string variable that is set to the value of the text entered by the user. `SECRET` is a keyword that means that the text is echoed back as `***`.

`DISPLAY OPENBOX title,filter,ext,exists RESPONSE var,mess_txt`

Displays a file browse box. `title` is a string variable or string constant in quotes that is displayed as the title of the box.

`filter` is a string variable or string constant in quotes in format `Type|mask` e.g. `List files|*.lis|All files|*.*|`

`ext` is a string variable or string constant in quotes and is the file extension in lower case without a leading period.

`exists` is a numeric variable or constant. 1 means the file must already exist; 0 allows a new file to be created; -1 means that the command returns a directory name instead of a filename.

`var` is a numeric variable that receives 0 (Cancel) or length of string. `mess_text` is a string variable that is set to the value of the filename selected or entered by the user.

```
DISPLAY SAVEBOX title,filter,ext,overwrite RESPONSE var,filename
```

Displays a Save As file box. `title` is a string variable or string constant in quotes that is displayed as the title of the box.

`filter` is a string variable or string constant in quotes in format Type|mask e.g. List files|*.lis|All files|*.*|

`ext` is a string variable or string constant in quotes and is the file extension in lower case without a leading period.

`overwrite` is a numeric variable or constant. If this is 1 then, if the file already exists the user is prompted for permission to overwrite; if 0, then an existing file is simply overwritten.

`var` is a numeric variable that receives 0 (Cancel) or length of string. `mess_text` is a string variable that is set to the value of the filename selected or entered by the user.

```
PRINT filename [DEFAULT] [MARGINS l,r,t,b] [FONT n]  
[WRAP|PAGE|TRUNCATE]
```

Prints a text file. `filename` is a string variable or string constant in quotes and is the operating system filename not a SIR/XS internal file attribute.

If the `DEFAULT` keyword is specified printing commences immediately, otherwise a Print box to alter print specifications is displayed.

`MARGINS` are in mm and defaults are 25 left, right 20 top, bottom.

`FONT` is in points with a default of 10.

`WRAP`, `PAGE` and `TRUNCATE` specify the way long lines are handled. Wrap splits the long line over several print lines; Page will print long lines on separate pages and Truncate will only print that part of the line that will fit on the page.

```
INVOKE DDESIGN filename
```

Invokes the Dialog Designer. `filename` is a string variable or string constant in quotes and is the operating system filename not a SIR/XS internal file attribute. The file should be a file saved by the dialog designer.

DEDIT

```
DEDIT type,id,arg1,arg2
END DEDIT
```

The `DEDIT` dialog editor is a full screen dialog that allows the program and user to interact to place controls of various types and to visually edit these controls. The dialog visually resizes to accommodate controls. This is the basis for the dialog painter and `PQLForms` painter and can also be used to develop custom painting style applications.

Controls are placed on the dialog through commands and functions rather than through any definitions. There is no separate message definition command. Control is returned to this block each time a message is generated.

The command requires the specification of four numeric variables. These variables return the message type, the id of any control and two arguments, either x (across the screen) and y (down the screen) co-ordinates or width and height of the control on a size message.

Message types are as follows:

Value	Message	id	arg1	arg2
0	Initial	0	0	0
1	Exit	0	0	0
2	Key	key	0	0
3	Rclick	id	x	y
4	Ctrl-D	0	0	0
5	Move	id	x	y
6	Resize	id	w	h
7	Del	id	0	0
8	Dblclick	id	0	0
9	F9	0	0	0
10	F5	0	0	0
11	F1 (Help)	0	0	0
12	Ctrl-Z (Undo)	0	0	0

Initial is sent after the dialog editor window has been created, but before it is made visible. The program can put the initial set of controls into the editor.

Exit is sent when the user tries to close the dialog editor. The program needs to exit the dialog to actually stop the editor.

Rclick is sent when the user right clicks the mouse. If the mouse is positioned on a control, the control is selected (if not already selected) and the id is passed, otherwise any selected control is deselected and id is set to zero. X and Y position is passed.

Move and Resize are sent as the user moves/resizes a control. The program should just accept the new position or size and do nothing else. If a group of controls is moved/resized as a single operation, the program receives separate messages for each control. The program must not interact with the user nor change the editor's control set while processing this message.

Dbclick is sent when the user quickly left clicks the mouse twice. If the mouse is positioned on a control, the control is selected and id is passed, otherwise any selected control is deselected and id is set to zero.

Help is sent when the user presses F1. The application should display appropriate help.

Other messages are passed when the user presses various keys. It is up to the individual application to assign meanings to these. An application typically uses the `DISPLAY POPUP LIST` command to offer the user a set of appropriate actions to take e.g. to insert a control, to copy a control, etc. on given messages. The Rclick passes co-ordinates so is appropriate for inserting a new control at a given position.

The following keys are enabled during a `DEDIT` dialog:

Key				
ESC	No Message - Unselects if one or more controls are selected			
TAB	No Message - Selects next			
		Message	id	arg1 arg2
Shift+Esc		Exit	0	0 0
Esc (None Selected)		Exit	0	0 0
Arrow (Selected Item(s))		Move	id	x y
Shift+Arrow "		Resize	id	w h
Ctrl+Arrow "		Fine Move	id	x y
Ctrl+Shift+Arrow "		Fine Resize	id	w h
Delete		Delete	id	0 0
Return or Space none selected)		Dbclick	id	0 0 (id is zero if
Ctrl+Z		Undo	0	0 0
D		Ctrl-D	0	0 0
F1		Help	0	0 0
F2		Rclick	0	0 0
F5		F5	0	0 0
F9		F9	0	0 0
Insert		Key	0	0 0

The following keys (or Ctrl-key) have mnemonics to associate with control types if required

L:[Label]	Key	1	0	0
E:[Edit]	Key	2	0	0
B:[Button]	Key	3	0	0
K:[Check]	Key	4	0	0
R:[Radio Button]	Key	5	0	0
C:[Choice]	Key	6	0	0
M:[List]	Key	7	0	0
T:[Text]	Key	8	0	0
H:[Horizontal Line]	Key	9	0	0
V:[Vertical Line]	Key	10	0	0
S:[Box]	Key	11	0	0
I:[Image]	Key	12	0	0

Note The dialog editor is not reenterable. You cannot start a second instance while the first one is active.

INSERT DCONTROL

```
INSERT DCONTROL id,type,x,y,w,h,text
```

Inserts a control on a DEDIT dialog. Control types are as follows:

LABEL	1
EDIT	2
BUTTON	3
CHECK	4
RADIO	5
CHOICE	6
LIST	7
TEXT	8
HLINE	9
VLINE	10
LBOX	11
IMAGE	12

Positioning of dialog edit controls uses a horizontal (x) and vertical(y) system of co-ordinates starting at 0,0 in the top left. The horizontal units are 1/4 of the average width

for the font being used. The vertical units are 1/8 font height. See `POSTYPE` to use these same units in standard dialogs.

Height and width use the same units. Height is irrelevant to control types 1 to 6. Various controls have minimum height and/or width limits. A height of 12 and a width of 32 is sufficient for all control types.

The text is displayed in an appropriate place for the control and should be helpful to the user to identify the control in some way.

MODIFY DCONTROL

```
MODIFY DCONTROL id,x,y,w,h,text
```

Modifies the position, size and text of a control on a `DEDIT` dialog. The control type cannot be modified.

MODIFY DCONTROL FONT

```
MODIFY DCONTROL id,bold,italic,underline,size,font/colour
```

Modifies the font of a control on a `DEDIT` dialog. The specifications for the font are the same as `SET ITEM FONT`.

REMOVE DCONTROL

```
REMOVE DCONTROL id
```

Removes a control from a `DEDIT` dialog.

SELECT DCONTROL

```
SELECT DCONTROL id
```

Selects a control on a `DEDIT` dialog. The user can do this interactively without program intervention.

CLEAR DCONTROL

```
CLEAR DCONTROL id
```

De-selects a control on a `DEDIT` dialog. The user can do this interactively without program intervention.

DEDIT MESSAGE

```
DEDIT MESSAGE text_exp
```

Displays a text message in the message area at the bottom of a `DEDIT` dialog. The current position, size and id of any selected control are automatically displayed alongside the message area.

GRID

```

GRID title_string_exp
      list_of_arrays (1 or 2 dimension)
      [HEADERS=(list_of_col_headers)]
      [RESPONSE = integer_varname |
                    array_varname]
      [SIZE=rows]
      [DISPLAY=row,width]
      [UPDATE | NOUPDATE]

```

The `GRID` command displays data in a spreadsheet format and can be used to display arrays of data and to accept back changes. The grid is a dialog with predefined buttons and a grid of data. The columns are array variables; the rows are occurrences in the array. The grid displays very quickly and essentially has no size limitations beyond those imposed by processing very large arrays.

`title_string_exp` A string expression (e.g. 'My Data') that is displayed as the title of the dialog.

`array_name,`
`array_name,`
`...` A list of array variables of one or two dimensions. The first dimension represents the number of rows of data. If the variable has a second dimension, this is taken as multiple columns. The following example displays 20 rows and 5 columns:

```

INTEGER*4 ARRAY MYARRAY (20)
INTEGER*4 ARRAY MYARRAY1 (20,4)
GRID 'Example' myarray, myarray1

```

Variables can be any type. Maximum total number of columns is 256.

`HEADERS=`
`(heading,`
`heading, ...)` A list of string expressions that are used as column headings. The position in the list corresponds to the column. The default heading is the variable name. The default headings for columns defined by a two dimensional array is the variable name for the first occurrence, then the variable name and subscript value for subsequent occurrences.

`RESPONSE =`
`integer_varname |`
`array_varname` Either a single integer variable that contains 0,1 or -1 or a two dimensional array that contains 0,1 or -1 in each element. 0 means not updated; 1 means updated; -1 means an error occurred. If rows are deleted, then the array is shorter and response values after the

end of the new array size are set to missing.

SIZE=rows

Number of rows available to the user. The default is the first dimension of the smallest array being used (including any response array). If the user chooses to insert rows, the dimension of all arrays must be large enough to allow the insertion.

DISPLAY =
row, width

Number of rows and width of visible grid. Rows must be between 10 and 50; width between 50 and 150 characters. The default is 10 rows, 80 characters.

UPDATE |
NOUPDATE

Whether the user is allowed to update the data. The default is UPDATE.

PQLForms Overview

PQLForms is a set of commands that extend VisualPQL allowing you to create and run sets of linked, interactive screens for data entry, retrieval and update. A complete set of screens is a single VisualPQL routine known as a *Form*.

A Form can be created and maintained completely through the Forms Painter and this is the recommended way to develop forms. However, it may be necessary to use PQLForms commands and this chapter describes the various commands available.

The PQLForms commands define what variables are on each screen or page of a screen, how they are displayed and edited, how the screen is to look, and how screens are linked together. A PQLForm has built in buttons and associated logic to allow the user to navigate through a set of records and to display, edit and insert data according to the database description. A developer can use all standard VisualPQL commands as necessary and these are executed at appropriate places in the form.

A PQLform can be re-compiled every time it is used or the compiled version of the form can be saved as an executable member on the procedure file. A PQLForm can also be compiled and saved as a sub-routine and can then be executed as part of another PQLform or standard retrieval. A PQLform is run in the same way as any other VisualPQL routine either directly or from a menu.

Once a form has been developed, it can be used by many people for data entry or for querying data.

A default form can be generated for a database and can be used directly to view, create, or delete records.

Form Structure

A form definition consists of a set of commands, each of which may have various clauses. Normal VisualPQL syntax rules apply.

The form definition starts with the `FORM` command. This is similar to a `RETRIEVAL/PROGRAM/SUBROUTINE` command and can take all relevant clauses as per those commands plus PQLForms specific clauses. There are no required clauses on a `FORM` command. The entire form definition is terminated by the `END FORM` command.

The `FORM` command may be followed by any standard VisualPQL commands, (for example defining any local variables) and then optionally a `CALL SCREEN` command that transfers control to the named screen. The first `SCREEN` command begins definition of a

screen. All further commands are in a screen definition. Definition of a screen is terminated by the `END SCREEN` command. Commands in the screen define the set of fields that are displayed; these can be split into a number of separately displayed pages if necessary.

A form can contain any number of screens. Screens are linked to other screens with the `CALL SCREEN` command so that the user or the application can pass control at appropriate points. A single screen can be called from any number of different screens.

The standard VisualPQL commands up to the first `SCREEN` or `CALL SCREEN` command are executed and then execution starts with the first screen or called screen.

This keyword `MENU`, `RECORD` or `TABLE` on the `SCREEN` command specifies the type of screen being defined:-

- Menu screens are independent of any database record or table. They can act as a table of contents with a choice as to where to go next.
- Record screens relate to database records. These display data and can be used to enter new data and modify existing data.
- Table screens relate to tables on tabfiles. These display data and can be used to enter new data and modify existing data.

Within each screen, further commands are used to describe the components of the screen and their individual behaviour and appearance. The most common of these is the `FIELD` command. This defines an individual variable, possibly displaying it on the screen allowing the user to retrieve data and maybe update data. Default formats and edit rules from the data dictionary are applied automatically. There are clauses on this command to extend the edit rules and alter the position or format of the data.

A form has a structure similar to the following:

```
FORM
. SCREEN RECORD record_name
. FIELD field_name
. FIELD field_name
. CALL SCREEN record_name1
. END SCREEN
. SCREEN RECORD record_name1
. FIELD field_name
. FIELD field_name
. END SCREEN
ENDFORM
```

A very simple form definition (on the example `COMPANY` database) might be:

```
form
. screen record EMPLOYEE
```

```
. field id
. field name
. field currpos
. call screen OCCUP
. end screen
. screen record OCCUP
. field id
. field position
. field startsal
. end screen
end form
```

Position on the screen

By default, each defined element is displayed one row down and in the same column as the previous element. The display position can be explicitly specified with the `AT` clause on the command that specifies the element.

The visual size of the screen is determined by the maximum position taken by any display element plus space for buttons and an area to display messages. There is no absolute maximum row and column size and it is possible to create screens that are too big to view all at once. There is a default font face, color and size of characters. These can be altered and specific colors/sizes/fonts can be used.

Examples

There are a number of example forms in the family `EXAMPLE` on the example `COMPANY` database. These are named `FORMnnnn` and contain comments as to the various features that they illustrate.

Commands

The following commands can only be used in a `PQLForm`:

`FORM`

Begins the definition of a form.

`SCREEN`

Begins a set of commands for a screen.

`PAGE`

Defines a page of fields within a screen. A page shares data and logic with all other pages in that screen. It is a means to display and input data that belong to a single record or table but is too large to display on a single screen. Fields are automatically split into pages where necessary.

FIELD

Defines a field. Fields are always within a screen.

GENERATE

Creates a default set of fields from the schema.

CALL SCREEN

Passes control from one screen to another.

ABUTTON

Equivalent to the user pressing a button except the action is taken under program control.

FBUTTON

Sets the display position of a forms standard button or defines a user button and the code that is executed when it is pressed.

FDISPLAY

Displays text, lines, boxes or images on the screen.

END SCREEN

Defines the end of a previous screen.

END FORM

Defines the end of the form.

Specifying VisualPQL in PQLForms

There are three general places in a screen definition where standard VisualPQL can be used.

Execution Clauses

Some clauses on PQLForms commands allow standard VisualPQL commands that are then executed at specific points in the form execution. Where VisualPQL commands can be specified as part of a PQLForm clause, enclose the complete set of commands in brackets () and use a semi-colon ; after a command to indicate the start of a new command. For example, the `FIELD` command has the `EDITIN` and `EDITOUT` clauses that allow the specification of commands to transform data as it is read from the screen or displayed. :

```

FIELD SALARY  EDITOUT (fieldout = pformat(salary,'$zzzz.zz'))
                EDITIN (if (sbst(fieldin,1,1) eq '$') fieldin =
sbst(fieldin,2,len(fieldin)-1);
                    salary = numbr(fieldin);
                    ifthen (salary lt 1000);
                        failmess = 'Salary too low';
                        failfld  = 1;
                    fi)

```

Condition Clauses

Some clauses on PQLForms commands allow the specification of *conditions*. Where a single, standard VisualPQL condition can be specified as part of a PQLForm clause, again enclose it in brackets (). A condition must eventually resolve to true or false and cannot extend over multiple commands. For example, the `FIELD` command has the `IF` clause that determines whether the field is enabled or disabled (greyed out).

```

FIELD SALARY prompt 'Salary:'  IF (EDUC EQ 1)

```

Intermixed commands

Standard commands can also be intermixed with PQLForms `FIELD` commands. The commands are executed when the user presses `ENTER` as follows:-

- Commands before the first `FIELD` command are executed whenever the user presses `Enter`.
- Commands after a `FIELD` command are executed when a user is positioned on that field and presses `Enter` (with a valid value) after the value from the screen is put into the variable but before the screen is redisplayed.

The VisualPQL can perform any standard VisualPQL function including record access, calling subroutines, displaying sub-dialogs, etc.

Predefined Variables

To allow easy communication between the predefined PQLForms logic and user specified standard VisualPQL, certain predefined variable names have been used:-

<code>FAILFLD</code>	A numeric variable that can be set by commands that check a specific field as it is entered. A value of zero (0) is the default and means accept the field; a positive value means warn the user that the field has failed validity tests but they have the option to accept it; a negative value means the field has failed validity tests and is not accepted. The absolute numeric value has no specific meaning. The standard error message used if <code>FAILFLD</code> is not zero is number 57 'Failed Edit tests' and this can be overridden by setting a value in <code>FAILMESS</code> .
----------------------	--

FAILMESS	A string variable set by commands to the text of the message to display for a test that fails. If a positive FAILFLD or FAILSCR code is set, the text 'OK to save?' is appended to the message.
FAILSCR	A numeric variable set by commands that check record validity. During field processing or in the WRITE clause, a zero value accepts the update and is set by default. A positive value means the user has the option to accept the update after a warning. A negative value means the update is not done. The absolute numeric value has no specific meaning. The standard error message used if FAILSCR is not zero is number 110 'Record failed write tests'. If a SELECT clause is specified on the SCREEN command, this can set FAILSCR to a non-zero value to indicate that the record should be skipped.
FIELDIN	A string variable used as the starting point for data from the screen for any EDITIN commands to check or transform when the user presses Enter on that field.
FIELDOUT	A string variable used as the result of any EDITOUT commands to display the field on the screen.

Help

If the user requests help on a field (by clicking on the question mark from the top of the dialog and positioning it on a field or by pressing F1 on the field), a pop-up box is displayed with any defined help text. Define help text for a field with the HELP clause followed parentheses enclosing any standard PQL expression that resolves to a string. The expression can be a simple string constant in quotes or a concatenated string. To display multiple lines in the pop up box, concatenate `char(13)` as a line break character. For example:

```
FIELD BIRTHDAY
  PROMPT 'Date of Birth'
  HELP ('Enter as MMM DD, YYYY' + char(13) +
        'For example: May 24, 2001')
```

If there is no defined help text for a field, nothing is displayed i.e. no box is popped. If there is no help text for any fields, the question mark is suppressed.

Error Messages

Error messages are displayed on the status line with a number in parentheses following the message. A full listing of the error messages is contained in Messages. Error messages can be altered to reflect the needs of specific projects by specifying an ERROR clause for the error message number at any level (form, screen, field). This replaces the default error message for everything within that level. For example:

```
FIELD SSN  
ERROR 47 'Social security number is in the wrong format'
```

In this example, error message 47 "Not a valid value", is replaced by the message "Social security number is in the wrong format" if a user makes an error.

Using PQLForms

Run PQLForms as per any other VisualPQL program from the SIR/XS menus. Any database or tabfile required by a form must be connected.

Every PQLForm screen has a set of buttons that take standard actions. These buttons vary according to the type of screen but all include an `Exit` button to return one level. Typically screens also have various fields for the display and entry of data. Error messages are displayed at the bottom of the screen.

Each data field can have up to three visual elements: a prompt, a data area to enter or to display the data and the set of value labels displayed as a pull down choice list. When a screen is initially displayed, fields may already have data in them or the fields may be blank waiting for data to be entered or a record to be retrieved.

To enter data into a field, type in the data and press `Enter`. To skip over fields, use the `Tab` keys or position to a field with the screen cursor using a mouse or other pointing device. `Shift-Tab` goes back a field, `Tab` goes forward a field. `Tab` does not process the data nor execute any VisualPQL associated with the field.

Field Editing Operations

When positioned to a data entry field, edit the contents of the field if necessary. `Left Arrow` and `Right Arrow` position the cursor within the field. The data in a field may be longer than the display space and is scrolled horizontally as characters are typed or the arrow keys are used.

Keys to edit the field are the normal keys for the GUI being used. On Windows systems, `Del` deletes the next character, `Ctrl-Del` deletes the whole field; `Ctrl-C` cuts highlighted characters, `Ctrl-V` pastes cut characters, `Ctrl-Z` restores the previous edit, etc.

Press `Enter` to process the data in a field and move to the next field. If you update a field but do not press `Enter`, the edits have no effect.

Moving from screen to screen

A form often consists of multiple screens, one per record type or table. A screen is a single logical entity that may be split into several pages if the data does not fit on a single display screen.

If a screen is on multiple pages, the `Page Down` button moves to the next page; the `Page Up` button moves to the previous page. The title of the screen changes to indicate the current page.

To call a screen, press the appropriate button. The `EXIT` button returns to the calling screen. Lower level screens can call other screens and screens may be nested as deeply as necessary.

Accessing Records and Rows

Key fields identify the record or row and are usually the first fields at the top of the screen. Key fields must be specified on a screen. If the complete set of key fields is not specified, a warning is given at compile time and the missing key fields are automatically added as the last fields on the screen.

There are buttons to browse through sets of records:-

- *First* retrieves the first record;
- *Last* retrieves the last record;
- *Next* and *Previous* go through the set of records one at a time in the specified direction. If no more records are available in the set, a message is displayed.

If the screen is a top level screen, (it has not been called by another screen) then the set of records is all of the records of that type in the database or table. However, if a screen is called by other screens, then some part of the overall key may have been set by the calling screen. In this case, those key fields are read-only (cannot be modified) and the set of records is a subset with that specified part of the key. The browse buttons operate within that set of records i.e. The first button retrieves the first matching record in the subset not the first on the file.

To locate a record, enter data in all the key fields. Press Enter at the last key field and the matching record is then retrieved. If no record matches the exact key, a message that a no record has been found is displayed. This positions in the set of records even if a partial key is entered or the key does not exist and the Next or Previous buttons browse from that point.

Updating a Record

Once a record is displayed, the data fields can be modified.

At the end of processing a screen (as the user retrieves another record or exits the screen), if the form allows updates and any of the fields have been updated, the record is written back. This can be done automatically or a message can be displayed asking the user to confirm that the record should be written.

At any point after updating some field(s), press the `WRITE` button to write the changed data. The `WRITE` button is only displayed if the form allows updates and is only enabled when some data has been updated.

Prior to writing the data back, the `RESET` button restores the original data "undoing" whatever changes were made.

The `Clear` button clears the screen data fields except for any key values preset by the `CALL` to this screen.

Deleting Records

Once a record is displayed, the `DELETE` button deletes the record. The `DELETE` button is only displayed if the form allows updates and a record has been retrieved. The record is deleted after asking the user for confirmation.

PQLForms General Clauses

There are a number of settings or general clauses that can be specified at different levels and, if not over-riden at a lower level, apply to all field definitions within the level. Settings can be specified at the following levels:

```
FORM
  SCREEN
    PAGE
      FIELD
```

A setting at one level acts as the default for all lower levels. For example, the width of labels could be set on the `FORM` command for the whole form and would apply to every field on every screen. A setting is overridden for a particular level by specifying the clause at that level. If a particular setting is not specified at a given level, the current default applies.

These general clauses are:-

```
[NO]PROMPT [AT r,c] [WIDTH n] [FONT
([NO]BOLD|[NO]ITALIC|[NO]UNDERLINE|SIZE=N|FACE =
'fontname'|FGROUND=RRGGBB|BGROUND=RRGGBB)] ['prompt-
string'|VARDESC|VARLABEL|VARNAME]
[NO]DATA [AT r,c] [WIDTH n] [FONT
([NO]BOLD|[NO]ITALIC|[NO]UNDERLINE|SIZE=N|FACE =
'fontname'|FGROUND=RRGGBB|BGROUND=RRGGBB)]
[NO]LABELS [AT r,c] [WIDTH n] [FONT
([NO]BOLD|[NO]ITALIC|[NO]UNDERLINE|SIZE=N|FACE =
'fontname'|FGROUND=RRGGBB|BGROUND=RRGGBB)]
ERROR number 'error text'
```

Field Elements

There are three main elements to each field as displayed on the screen, from left to right, the *Prompt*, the *Data* and the *Label*.

- By default, a prompt is shown for each field and is the variable label (defined with a `VAR LABEL` command). The default position for the prompt is one row down and in the same column as the previous prompt or other displayed element (text, button, etc.) and is 18 columns wide. The first prompt is in the top left corner, row 1 column 1.

- By default, the data is the same row as the prompt starting one column after the end of the prompt and is 13 columns wide.
- By default, a label is not displayed. Labels only apply to fields with value labels. If a label width is specified, then the label is displayed (if the field has value labels) immediately after the end of the data field.
- The row, column and width of the prompt, data and label can be specified with common clauses on `FORM`, `SCREEN`, `PAGE` and `FIELD` commands. (See below.) Because the three elements are positioned from left to right relative to each other, if the prompt position or width is specified, the data and label positions are automatically adjusted. Similarly, if the data position or width is specified, the label position is automatically adjusted.

Screen co-ordinates

PQLForms uses a notional set of *Row* and *Column* co-ordinates to specify vertical rows and horizontal columns. The top row is 1 and the leftmost position is column 1.

The absolute row and column size are dependent on the font size being used. A single row is sufficient to display a field or button. A column approximates to an average character in the font. The number of rows and columns displayable on a screen depend on font size and screen resolution.

AT

`AT [row] [, column]`

`AT` is a clause on a number of commands and alters the starting position for the specified display element.

Row and column positions can be specified in either absolute or relative terms.

Absolute positions are specified with unsigned integers. For example, `AT 5` positions at row 5, `AT ,60` positions at column 60, `AT 5 ,60` positions at row 5 column 60.

Relative positions are specified by prefixing the integer with a plus or minus sign. Relative positions are relative to the default position for this element. For example, `AT -1 ,+40` specifies one row higher and forty columns to the right and means that the field appears on the same row as the previous field.

An asterisk (*) may be used instead of row or column numbers to indicate the default maximum row that is the current pagesize (default 20) or default maximum column (80).

Example AT clauses:

AT 1,1 positions to the upper left corner.

AT *,1 positions at row 20, column 1.

AT ,12 positions at column 12 of the current line.

AT -1,1 positions at the beginning of the previous line.

AT ,+10 positions 10 columns along from the default column position.

If AT is specified on PROMPT, the default DATA and LABEL column positions are updated. Similarly, if AT is specified on DATA, the default LABEL column position is updated.

WIDTH

WIDTH n

WIDTH is a clause on a number of commands and alters the width for the specified display element or the width of all lower level fields if specified on a higher level command.

The physical width of any field depends on font sizes in use and, since most fonts are variable width, there is no exact correspondence between columns and characters that can be displayed. A column approximates to an average character in the font. If the displayed data or label is wider than the display width, the user can scroll horizontally with the right and left arrows.

[NO]DATA

[NO]DATA [AT r,c] [WIDTH n]

Specifies position and/or width of the data area for fields.

AT

Specifies the starting position (row and column) of data area. If position is specified on a higher level command, it applies to the first field. If AT is specified on DATA, the default LABEL column position is updated.

WIDTH n

Specifies the width of data area. If WIDTH is specified on DATA, the default LABEL column position is updated.

NODATA

Suppresses the data area of the field. This also suppresses any label for the field.

[NO]LABELS

LABELS [AT R,C] [WIDTH n] | NOLABELS

Specifies position and/or width of the label area of a field. Labels correspond to value labels and are only displayed for fields that have value labels defined. Labels are displayed as a choice control. This is a pull down list with all allowed descriptions and the selected description corresponds to the value in the data field. Choosing a description updates the value in the data field.

AT

Specifies the starting position (row and column) of label area. If position is specified on a higher level command, it applies to the first field.

WIDTH n

Specifies that value labels, "n" characters wide, are displayed where appropriate.

NOLABELS

Suppresses the labels area of the field.

Example: To display 20 characters of value labels next to all data fields that have value labels, specify:

```
FORM TESTFORM LABELS WIDTH 20
```

[NO]PROMPT

[NO]PROMPT ['prompt-string' | VARDESC | VARLABEL | VARNAME] [AT r,c]
[WIDTH n]

Specifies how prompts for fields are displayed. Prompts are left justified and followed by a colon (:).

NOPROMPT suppresses the prompt. This does *not* alter the default start position of the associated data.

prompt string

Specifies a prompt string for a field. This format can only be used when specifying prompt defaults on a specific `FIELD` not on any higher level commands.

`VARDESC`

Specifies that the variable name and label are the prompt.

`VARLABEL`

Specifies that the variable label is the prompt. If a label does not exist, the variable name is used. This is the default.

`VARNAME`

Specifies that the variable name is the prompt.

`AT`

Specifies the starting position (row and column) of prompt area. If position is specified on a higher level command, it applies to the first field. If `AT` is specified on `PROMPT`, the default `DATA` and `LABEL` column positions are updated.

`WIDTH n`

Sets the width of the prompt. If `WIDTH` is specified on `PROMPT`, the default `DATA` and `LABEL` column positions are updated.

FONT

`FONT ([NO]BOLD | [NO]ITALIC | [NO]UNDERLINE | SIZE=N | FACE = 'fontname' | FGROUND=RRGGBB | BGROUND=RRGGBB)` Non-standard fonts can be specified with the `FONT` clause wherever a `DATA`, `PROMPT` or `LABEL` clause is specified. Follow the `FONT` keyword with a set of specifications enclosed in brackets () using as many as necessary. Note that some changes alter the amount of space required to display the item and other positioning specifications may need to be adjusted accordingly. Fonts specified on higher-level commands alter the default font for all fields within that level. The various specifications are:

- `BOLD` Display the element as bold.
- `ITALIC` Display the element as italic.
- `UNDERLINE` Display the element underlined.
- `SIZE=N` Increase or decrease the size of the element. Specify a negative number to decrease size.
- `FACE= 'fontname'` Display the element using a different font. Specify the name of the

font enclosed in quotes.

`FGROUND=RRGGBB` Display the element using a different foreground color. Specify the color using exactly six characters; valid characters are 0 to 9 and A to F. These are three sets of hexadecimal specifications of the strength of the red, green and blue components of the color. Each setting has a value from 00 to FF.

`FGROUND=RRGGBB` Display the element using a different background color. Specify the color using exactly six characters; valid characters are 0 to 9 and A to F. These are three sets of hexadecimal specifications of the strength of the red, green and blue components of the color. Each setting has a value from 00 to FF.

For example

```
SCREEN RECORD EMPLOYEE PROMPT FONT (FACE='Ariel')
FIELD ID PROMPT FONT (BOLD)
```

Note that higher level font settings apply to `FIELD` definitions and do not alter defaults for buttons. If you wish to alter the font on a buttons, specify the font on a `FBUTTON` command. Currently buttons do not support color but any other font specification can be used.

ERROR

```
ERROR message_number 'error_text'
```

Specifies alternate text for a message. Each message consists of the `message_number` followed by text. See messages for a full list of default messages.

Error messages apply to the level (`FORM`, `SCREEN`, `PAGE` or `FIELD`) at which they are defined and to all lower levels unless overridden by another error message defined at the lower level.

Example: To change the messages for errors 15 and 47 to "This field must be two digits." and "Jobcode must be between 10 and 60.", specify:

```
FIELD JOBCODE ERROR 15 'This field must be two digits.'
                ERROR 47 'Jobcode must be between 10 and 60.'
```

FORM

FORM

```
[appropriate standard RETRIEVAL/PROGRAM/SUBROUTINE clauses]
[PQLForms general clauses]
[AUTO]
[CLEAR|NOCLEAR]
[NODATABASE]
[PAGESIZE rows]
[PQLFILE='filename']
[SUBROUTINE name [(input_list)]]
[UPDATE]
```

General Clauses

```
[NO]DATA [AT r,c] [WIDTH n]
[NO]LABELS [AT r,c] [WIDTH n]
[NO]PROMPT [AT r,c] [WIDTH n] [VARDESC|VARLABEL|VARNAME]
ERROR number 'error text'
```

The `FORM` command is required. All other clauses are optional. All standard `RETRIEVAL` clauses can be specified. The `PQLForms` general clauses can be specified. The following specific clauses can be specified on the `FORM` command:

<code>AUTO</code>	Specifies that, if updates are allowed, any new records are written to the database without asking for confirmation. <code>NOAUTO</code> specifies that the user is asked to confirm that the record should be written and is the default.
<code>CLEAR</code>	Specifies that, if updates are allowed, the data fields on the screen are cleared after writing. <code>NOCLEAR</code> specifies that the data is not cleared and is the default.
<code>NODATABASE</code>	Specifies that the generated form is a <code>PROGRAM</code> not a <code>RETRIEVAL</code> and can run without being attached to a database.
<code>PAGESIZE n</code>	Specifies the maximum number of data rows in a page of a screen. The default is 20. (See the <code>PAGE</code>) command.
<code>PQLFILE</code>	Specifies that the named file is created. This contains standard VisualPQL code (i.e. all <code>PQLForms</code> extensions have been replaced) that performs identical functions to the specified form. This can be used as the basis for a program with other functionality if necessary.

SUBROUTINE Specifies that the routine is compiled and saved as a **SUBROUTINE**. The subroutine name is required and is the name of the compiled subroutine. The name of the subroutine can be qualified with procedure file and family prefixes and passwords.

A subroutine may have input parameters. These are positional parameters corresponding to the **EXECUTE SUBROUTINE** list of parameters. The parameters are read-only and are local variables in the subroutine. These variables must be defined explicitly within the subroutine.

To pass key variables to a **SCREEN** defined in a **FORM SUBROUTINE** you would use:

```
FORM SUBROUTINE name (keys)
CALL SCREEN scrname USING (keys)
. SCREEN RECORD scrname
...
```

UPDATE Specifies that record and table screens allow the reading and writing of records. By default, screens do not update the database or tabfile. Individual screens can be set to allow/disallow updates. Local variables can be input regardless of update status.

SCREEN

```

SCREEN
  {MENU name |
   RECORD name [/database.record][ INDEXED BY indexname ] |
   TABLE name [/tabfile.table] [ INDEXED BY indexname ]}
  [AUTO|NOAUTO]
  [CLEAR|NOCLEAR]
  [DELETE (pql code)]
  [INITIAL (pql code)]
  [NOBUTTON]
  [PAGESIZE rows]
  [READ (pql code)]
  [SELECT (pql code)]
  [TITLE (pql code)]
  [UPDATE | NOUPDATE]
  [WRITE (pql code)]

```

General Clauses

```

[NO]DATA [AT r,c] [WIDTH n]
[NO]LABELS [AT r,c] [WIDTH n]
[NO]PROMPT [AT r,c] [WIDTH n] [VARDESC|VARLABEL|VARNAME]
ERROR number 'error text'

```

A `SCREEN` command starts a screen definition. The screen name must be specified and must be unique within a form. This is displayed as the title of the screen. The name must be a valid SIR/XS name.

Follow the screen command with further commands to display fields, to control the appearance of the screen or to link to other screens. The sequence of the fields within the screen definition determines the sequence for moving to the next field. If the display element positioning clauses are used, this may not necessarily be the order of fields as they appear on the screen.

End the set of commands for a screen with the `END SCREEN` command.

There are three screen types defined by the `MENU`, `RECORD` or `TABLE` keyword.

MENU

Within a menu screen, `FIELD` commands can only refer to local variables. Menu screens may consist simply of buttons offering choices as to which screens to go to or can display fields or allow data entry providing that these are local variables.

The `AUTO`, `READ`, `SELECT` and `WRITE` clauses do not apply to `MENU` screens.

RECORD

Record screens access and display one record at a time. Within a screen, commands can reference any variable from that record plus any local variables. Commands can also reference common variables in a case structured database.

Specify the screen name and, optionally, the name of the record and the name of the database. If a record name is not specified, the screen name must be the record name. If a database name is not specified, the current database is the default.

The name `CIR` can be used as a record name on a case structured database to refer to the common information record.

An index can be specified for a record and the index variables are treated as the key fields. Only one index can be specified for a record screen. If an index is not specified, the record is processed by any case and record keys.

The same record type can be associated with multiple screen definitions in the same form definition to allow different ways of viewing the same data. Each screen name must be unique.

When control is passed to a record screen, a record can be retrieved or a new entry created. The user can enter the key fields to locate the record, or can find the appropriate record with `FIRST`, `LAST`, `NEXT` or `PREVIOUS` buttons. The record or set of records to retrieve can be determined by clauses on the `CALL SCREEN` command.

TABLE

Table screens access and display one row at a time. Within a screen, commands can reference any variable from that table plus any local variables.

Specify the screen name and, optionally, the name of the table and the name of the tabfile. If a table is not specified, the screen name must be the table name. If a tabfile is not specified, the default tabfile is used.

An index can be specified for a table and the index variables are treated as the key fields. Only one index can be specified for a table screen. If an index is not specified, the table is processed sequentially.

The same table can be associated with multiple screens in the same form definition to allow different ways of viewing the same data. Each screen name must be unique.

When control is passed to a table screen, a row can be retrieved or a new entry created. The user can enter the key fields to locate the row, or can find the appropriate row with `FIRST`, `LAST`, `NEXT` or `PREVIOUS` buttons. The row or set of rows to retrieve can be determined by clauses on the `CALL SCREEN` command.

Clauses

<code>AUTO NOAUTO</code>	Specifies that any new records are written to the database without asking for confirmation. <code>NOAUTO</code> specifies that the user is asked to confirm that the record should be written and is the default.
<code>CLEAR</code>	Specifies that, if updates are allowed, the data fields on the screen are cleared after writing. This is the default if not set at the <code>FORM</code> level. <code>NOCLEAR</code> specifies that the data is not cleared.
<code>DELETE (PQL Code)</code>	<p>Specify VisualPQL code, enclosed in brackets, that is executed when a record or row is about to be deleted. The record (row) is deleted by the user pressing the <code>DELETE</code> button. The VisualPQL can be any set of commands. The executed code can create and display sub-dialogs or error boxes as necessary. Separate multiple commands with a semi-colon <code>';</code>.</p> <p>The <code>DELETE</code> commands can set a value in <code>FAILSCR</code> to warn the user and reject the delete.</p> <p>Example: To test that when employee has a current position, tell the user and ask whether to accept the delete:</p> <pre>SCREEN RECORD EMPLOYEE DELETE (IFTHEN (EXISTS(CURRPOS)); COMPUTE FAILSCR = 15; COMPUTE FAILMESS='Employee in current position'; FI)</pre>
<code>INITIAL (PQL Code)</code>	<p>Specifies VisualPQL code, enclosed in brackets, that is executed when the user first initiates a screen before anything is displayed, before the first command in the screen and before a record is accessed. The set of VisualPQL to execute is any set of commands but should not display any sub-dialogs or any other graphical elements. Separate multiple commands with a semi-colon <code>';</code>.</p> <p>Example: To set the local variable <code>INTIME</code> to the current time when the screen is accessed, specify:</p> <pre>SCREEN RECORD EMPLOYEE INITIAL (COMPUTE INTIME=NOW(0))</pre>
<code>PAGESIZE n</code>	Specifies the maximum number of rows in a page of a screen. The

default is 20. If the number of rows on a screen exceeds this size, then a new page is created automatically. When a screen has multiple pages, each page is displayed separately, the Page Up/Page Down buttons are displayed and the screen title contains the current page number. There is no limit to the number of pages on one single screen. All pages within a screen are the same size visually. All pages within a screen are one logical entity i.e. record and table screens access one single record or row. Page breaks can be set specifically at given points by the `PAGE` command.

`READ (PQL
Code)`

Specify VisualPQL code, enclosed in brackets, that is executed when the user reads a record. The code is executed when a new record is retrieved, after the record is read and before the data is displayed. The VisualPQL can be any set of commands. The executed code can create and display sub-dialogs or error boxes as necessary. Separate multiple commands with a semi-colon ';':

`SELECT (PQL
Code)`

Specify VisualPQL code, enclosed in brackets, that is executed as the user moves to another record. The code is executed after a record is read and selects whether this record is wanted. The VisualPQL can be any set of commands that eventually set `FAILSCR` to non-zero if the record is not wanted. Records that are not wanted are skipped and the user is presented with the next wanted record. Separate multiple commands with a semi-colon ';':

Example: To ignore records where salary is under 2500, specify:

```
SCREEN RECORD EMPLOYEE SELECT (IF (SALARY LT 2500)
FAILSCR = -1)
```

`TITLE (PQL
Code)`

Specify VisualPQL code, enclosed in brackets, that is executed as each page of the dialog is displayed and constructs the title of the dialog. If this clause is specified, the default title is suppressed. The VisualPQL can be any set of commands that issue a `SET DIALOG TITLE` command to set the title. There are two predefined variables available. `PAGENO` is the current page number; `PAGES` is the total pages. These are both *string* variables so can easily be included in a title expression if required. Separate multiple commands with a semi-colon ';':

Example: To put out a title, specify:

```
SCREEN RECORD EMPLOYEE TITLE (SET DIALOG TITLE 'People
Page ' + PAGENO + ' of ' + PAGES)
```

`[NO]UPDATE`

`UPDATE` specifies that this record or table screen allows the reading and writing of records. `NOUPDATE` specifies that this record or table screen does not allow the reading and writing of records. By default, screens are set to the update status of the form. Local variables can be input

regardless of update status.

WRITE (PQL
Code)

Specify VisualPQL code, enclosed in brackets, that is executed when a record or row is written. If data has been updated, the record (row) is written directly by the user pressing the WRITE button or when the user retrieves a new record (row) on this screen or exits from the screen. The code is executed after all local variables have been set to the values displayed on the screen. This VisualPQL Code could be used to implement ACCEPT/REJECT RECORD and REQUIRED field functionality from old style SIRForms.

The VisualPQL can be any set of commands. The executed code can create and display sub-dialogs or error boxes as necessary. Separate multiple commands with a semi-colon ';'. The WRITE commands can check the validity of a record (row) and can set a value in FAILSCR to warn the user and reject the write.

The WRITE commands can check the validity of a record (row) and can set a value in FAILSCR to warn the user and reject the write.

Example: To compute the sum of VAR1, VAR2 and VAR3, specify:

```
SCREEN RECORD EMPLOYEE WRITE (COMPUTE TOTAL = VAR1 + VAR2
+ VAR3)
```

To test that when salary is greater than 5,000, tell the user and ask whether to accept the record:

```
SCREEN RECORD EMPLOYEE
  WRITE (IFTHEN (SALARY GT 5000);
        COMPUTE FAILSCR = 15;
        COMPUTE FAILMESS='Salary over $5,000';
        FI)
```

Caution

Since there is only one fail flag for a screen, if you want to test multiple conditions then you should ensure that you don't reset an error status to a warning. For example:

```
WRITE (
IFTHEN (EXISTS(NAME) EQ 0) SET FAILSCR(-1); SET FAILMESS ("You MUST
enter a name"); ENDIF;
IFTHEN (EXISTS(DOB) EQ 0) SET FAILSCR(1) ; SET FAILMESS ("You really
should enter a birthday"); ENDIF;
IFTHEN (EXISTS(GENDER) EQ 0) SET FAILSCR(1) ; SET FAILMESS ("You really
should enter a gender"); ENDIF;
)
```

If none of the variables above is entered then only one *warning* about gender is displayed.

However, with:

```
WRITE (
SET FAILMESS ("You really should enter:")
IFTHEN (EXISTS(DOB) EQ 0) SET FAILSCR(1) ; COMPUTE FAILMESS =
FAILMESS + "Birthday; "; ENDIF;
IFTHEN (EXISTS(GENDER) EQ 0) SET FAILSCR(1) ; COMPUTE FAILMESS =
FAILMESS + "Gender; "; ENDIF;
IFTHEN (EXISTS(NAME) EQ 0) SET FAILSCR(-1); SET FAILMESS ("You MUST
```

```
enter a name"); ENDIF;  
)
```

If none of the variables above is entered then an *error* message on name is displayed. Then on the next write attempt the *warning* message about birthday and gender is displayed.

END SCREEN

END SCREEN

A screen must be ended with the `END SCREEN` command. There are no further clauses on the command.

PAGE

```
PAGE  
[PAGESIZE rows]
```

General Clauses:

```
[NO]DATA [AT r,c] [WIDTH n]  
[NO]LABELS [AT r,c] [WIDTH n]  
[NO]PROMPT [AT r,c] [WIDTH n] [VARDESC|VARLABEL|VARNAME]  
ERROR number 'error text'
```

The `PAGE` command specifies that a new page begins at this point. A new page resets the row position of the next display item to the top of the screen. When a screen has multiple pages, the Page Up/Page Down buttons are displayed and the screen title contains the current page number.

There is no limit to the number of pages on one single screen. All pages within a screen are one logical entity related to one single record or row.

If any of the general clauses are specified, these apply to the first field on the page.

Clauses

```
PAGESIZE n
```

Specifies the maximum number of data rows in a page of a screen. The default is 20. If the screen has less rows than the page size, then the dialog is the minimum size to accommodate the visual elements.

If the row placement for a field or button exceeds the page size, a new page is created, the row is reset to the default row for a page (1 if not specified) and subsequent display elements are included in this new page. Lines, boxes and other elements specified with `FDISPLAY` commands do not trigger automatic paging.

The page size is checked after any positioning clauses are processed and a new page is created if necessary.

`PAGESIZE n` can be specified on the `FORM` command, on a `SCREEN` command or on a `PAGE` command.

FIELD

```
FIELD variable_name
[EDITIN (pql commands)]
[EDITOUT (pql commands)]
[HELP (help string expression)]
[IF (pql condition)]
[NOECHO]
[READONLY]
[TYPE [INTEGER | STRING | REAL | DATE | TIME]]
```

General Clauses:

```
[NO]DATA [AT r,c] [WIDTH n]
[NO]LABELS [AT r,c] [WIDTH n]
[NO]PROMPT [AT r,c] [WIDTH n] ['prompt-
string'|VARDESC|VARLABEL|VARNAME]
ERROR number 'error text'
```

The `FIELD` command displays the current value of a variable on the screen and provides the capability to invoke standard PQL for editing and validating data values.

The only required clause on the `FIELD` command is the name of the variable. A `FIELD` command without any clauses displays the data at a default position on the screen using the dictionary definitions to control the prompt, the data format and the edit rules.

The `FIELD` command is used in record screens for record variables, in table screens for table columns and in any screen for local variables. The same variable can be referenced by multiple `FIELD` commands on a screen.

The sequence of the `FIELD` commands determines the sequence followed by the cursor on the screen when the user presses Enter or uses the Tab keys. Field commands do not have to correspond to the sequence of variables in a row or record. All of the fields on a record or table do not have to be on a screen.

The `FIELD` command creates visual entries in the screen together with appropriate logic to display and modify the data.

If VisualPQL commands are interspersed with `FIELD` commands then:

- Commands specified before the first `FIELD` in a screen are executed whenever the user presses Enter;
- Commands specified after a `FIELD` are executed when a user is positioned on that field and presses Enter.

Clauses

`EDITIN` (pql commands)

Specify VisualPQL code, enclosed in brackets, that takes the value from the predefined string variable `FIELDIN` and sets the value of the record, row or local variable. Separate multiple commands with a semi-colon ';'.
Example: A field (named SSN) for entering a social security number may be displayed with hyphens. The following removes these:

```
FIELD ssn EDITIN (COMPUTE ssn = REPLACE (fieldin, '-  
,',',',LEN(fieldin),1,0))
```

If the `EDITIN` commands check the validity of a field, set a value in a predefined variable, `FAILFLD`, to indicate what is to be done with the field. A value of zero (0) is the default and means accept the field; a positive value means warn the user that the field has failed validity tests but they can choose to accept it; a negative value means the field has failed validity tests and is not accepted. The standard error message is number 57 'Failed Edit tests'. Set the value of the string variable `FAILMESS` to display a different message for the test that fails. If a positive error code is set, the text 'OK to save?' is appended to the message.

The `EDITIN` commands completely replace the default field assignment and validation so your PQL code *must* assign a value to the variable named in the `FIELD variable_name`.

`EDITOUT` (PQL commands)

Specify VisualPQL code, enclosed in brackets, that alters the way the field is displayed. The specified commands are typically a function or set of functions that use the field as the input and create the predefined string output field `FIELDOUT`. The form then uses this as the field displayed in the screen. Separate multiple commands with a semi-colon ';'.
Example: To display a social security number in the format ddd-dd-dddd:

```
FIELD SSN EDITOUT (compute fieldout = edit (ssn, "^^^--^^-^^^^"))
```

N.B. If the user just edits the field then any characters added or taken out by this process become the input. This must be dealt with, either by the user clearing the field when editing or by appropriate logic in the `EDITIN` command.

`HELP (string expression)`

Specifies a text string or expression to display when help for a specific field is requested by the user. When the user requests help, the string expression is resolved and displayed as a pop up box.

`IF (pql condition)`

Controls whether a field is enabled or disabled (greyed out). All fields are normally enabled. The specified condition is tested every time the data in the screen is updated and the field is enabled if the condition is true otherwise it is disabled.

Example: Allow monthly rent to be entered if `OWNHOME` is not equal to 1.

```
FIELD MONRENT IF (OWNHOME NE 1)
```

`NOECHO`

Makes the field protected so the user does not see the characters in the field as they are entered or displayed. Depending on the specific operating system, the characters are replaced with asterisks or blanks.

`READONLY`

Specifies that the field is read only. It cannot be modified nor can new data be entered.

`TYPE`

When specifying local variables on a `FIELD` command, the `TYPE` specifies the type of field being referenced. By default, the type is `INTEGER`. The local variable should also be defined using standard PQL in the beginning of the form.

CALL SCREEN

```
CALL SCREEN screen_name

[AT r,c]
[AUTO [(pql condition)]]
[HELP (help expression)]
[IF (pql condition)]
[ONCALL FIRST | LAST]
[PROMPT 'prompt']
[USING (caseid,* | key,...)]
[VIA (* | key,...)]
[WIDTH n]
```

The `CALL SCREEN` command passes control from one screen to another, typically when the user presses the button generated by the command.

The only required clause on the `CALL SCREEN` is the called `screen_name`. This references another screen that is included in this form. If the name is a non-standard name, ensure that it exactly matches the screen name as specified on the called screen.

The default prompt is the called screen name.

A called screen can be blank for the user to enter any keys and retrieve any records or rows, or the key fields can be passed by this command to control the set of records or rows that the called screen is to reference.

The data fields on the called screen may be blank for the user to select the required screen (first or last in the set) or a record or row may be retrieved automatically with an `ONCALL FIRST` or `ONCALL LAST` clause.

Clauses

`AT`

Specifies where on the screen the `CALL SCREEN` button is displayed.

`AUTO [(pql condition)]`

Specifies that screen is called automatically when the user presses Enter on the previous field. The `AUTO` option suppresses the display of a button and does not affect the position of subsequent fields. The visual clauses `AT`, `PROMPT` and `WIDTH`

thus have no effect. If the user needs a normal button to choose to call the screen, specify a second `CALL` command.

Specify a VisualPQL condition (enclosed in brackets) that controls whether the screen is called at this point. If the condition is true, the screen is called.

`HELP (string expression)`

Specifies a string expression to display when help is requested by the user when positioned at the `CALL SCREEN` field.

If the user requests help, the string is displayed as a pop up box.

`IF (PQL condition)`

Controls whether the button is enabled or disabled (greyed out). Call buttons are normally enabled. The condition is tested every time the data in the screen is updated and the button is enabled if the condition is true, otherwise it is disabled.
`CALL NEXTOFKIN IF (RELATIVE = 1)`

`ONCALL FIRST | LAST`

Specifies that either the `FIRST` or `LAST` record in the set of records available to the called screen is retrieved and displayed automatically.

`PROMPT 'string'`

Specifies the label on the button. If a `PROMPT` is not specified, the name of the called screen is used.

`USING (list,...)`

Specifies a list of variables that the called screen uses as keys to a different case on a case structured database. Specify the case id value, optionally followed by the key field values.

Example: To call a screen `INDEX` that references a new case with a case id of -1 and two other keyfields:

```
CALL INDEX USING (-1,NAME, ID)
```

`VIA (list, ...)`

Specifies a list of variables that the called screen uses as keys. Use `VIA` to select records within this case, to access other records on a caseless database or to access a `SCREEN TABLE`. Do not specify a case id on the `VIA` clause. Do not use `USING` and `VIA` on the same `CALL SCREEN` command.

If a `CALL` is within a `SCREEN RECORD` the key specification for either `USING` or `VIA` keys can contain an asterisk (*) indicating that the key fields of the record from the current screen are used. The asterisk can be preceded or followed by other values. If an asterisk is preceded by values, these values are used positionally. e.g. If a record has three keys, the expression `(1,*)` means use '1' as the first key and take the second and third key values from this record.

If an asterisk is followed by values, the values are used positionally after the number of keys in the calling record. e.g. If the current record has three keys, the expression `(*,1)` means use the first, second and third keys from this record and use '1' as the fourth key (presumably the record being called has at least four keys).

If one or more of the lower level keys are omitted, all records with the specified keys are accessible.

If a `CALL` is within a `SCREEN RECORD` and a `USING` or `VIA` clause is not specified, all higher level keys are passed automatically that is equivalent to `USING (*)` or `VIA (*)`.

`WIDTH n`

Specifies the width of the `CALL SCREEN` button.

A `CALL SCREEN` can appear before the first `SCREEN` block and is typically used to pass on keys that have been sent to the `FORM` routine:

```
FORM SUBROUTINE name (keys)
CALL SCREEN scrname USING (keys)
. SCREEN RECORD scrname
...
```

FDISPLAY

```
FDISPLAY
  [AT r,c]
  [[DRAW [HEIGHT N] [IMAGE (bitmap_filename_expression) [BORDER]]]
  [TEXT (text_expression) [FONT
([NO]BOLD|[NO]ITALIC|[NO]UNDERLINE|SIZE=N|FACE =
'fontname'|FGROUND=RRGGBB|BGROUND=RRGGBB)]]
  [WIDTH n]
```

The `FDISPLAY` command puts a line, box, image or text on the screen. Expressions are evaluated and the display updated when the screen is refreshed (e.g. when a new record is read).

AT

Specifies a start position. The default start position is the current default `PROMPT` start position.

DRAW

Creates vertical and horizontal lines and boxes. Lines and boxes avoid the exact placements of fields so the same row column references as fields can be used to group sets of fields visually. Lines and boxes do not alter the default position for the next field.

`WIDTH` specifies how wide the line/box is and `HEIGHT` specifies how high the line/box is. If the height is zero (or unspecified), a horizontal line is produced. If the width is zero (or unspecified), a vertical line is produced. If both width and height are specified, a box is produced.

The `IMAGE` clause specifies an expression that is resolved to the name of a bitmap file and the image from this is displayed in the box. The `BORDER` clause puts a border around the image. If an image is displayed, the default position is updated by the height of the image.

TEXT

Specifies a text expression that is resolved and displayed as text. The default position is updated by the standard single row. The `WIDTH` specifies how wide the

text is. The default is the current default `PROMPT` width. A `FONT` sub-clause can be specified as per the `FONT` sub-clause on the standard `PROMPT` clause.

For example:

```
FDISPLAY TEXT ('Welcome to the Administration System') FONT (UNDERLINE
SIZE=2) WIDTH 30
FDISPLAY TEXT ('Please contact ext. 123 for help') at +2,+5
FDISPLAY TEXT (NAME) AT 4,30 WIDTH 20      | Displays value in variable
NAME
FDISPLAY DRAW WIDTH 80                    | Draws line for 80 columns
FDISPLAY AT 1,1 DRAW HEIGHT 10 WIDTH 50   | Draws box
FDISPLAY DRAW HEIGHT 5 WIDTH 40 IMAGE ('LOGO.BMP') | Displays image
from file
```

ABUTTON

```
ABUTTON {FIRST | LAST | NEXT | PREVIOUS | EXIT | RESET | WRITE | CLEAR  
| DELETE | PAGEDOWN | PAGEUP}
```

The `ABUTTON` command is equivalent to the user pressing a button, except that it is done under program control. This command can be used anywhere in-line VisualPQL can be used but cannot be used in clauses on other `PQLForms` commands.

FIRST

Gets the first record for this screen.

LAST

Gets the last record for this screen.

NEXT

Gets the next record for this screen.

PREVIOUS

Gets the previous record for this screen.

EXIT

Returns to the calling screen or exits from the form if this is the top screen.

RESET

Restores all the record or table variables to their original values.

WRITE

Writes the record or row to the database or tabfile.

CLEAR

Clears all data fields and any key fields that are not preset by a calling screen.

DELETE

Deletes the record or row from the database or tabfile.

PAGEDOWN

Displays the next page of the screen.

PAGEUP

Displays the previous page of the screen.

FBUTTON

```
FBUTTON [button_name | ACTION (pql commands)]
        [AT r,c]
        [ID n]
        [IF (pql condition)]
        [PROMPT ' ' ]
        [REMOVE]
        [WIDTH w]
```

The `FBUTTON` command alters the visual appearance (position, width and prompt text) of a system button or defines a user button.

`button_name`

Use one of the following button names to identify a system button:

```
FIRST | LAST | PREVIOUS | NEXT | EXIT | CLEAR |
STATUS | WRITE | RESET | DELETE | PAGEUP | PAGEDOWN |
```

`ACTION (pql commands)`

Specifies a user button. Specify PQL commands to execute when pressed.

`AT`

Specifies the starting position (row and column) of the button.

`ID`

Specifies an internal id for an `ACTION` button. Only specify this if you need to refer to this button in some other VisualPQL code in the form, for example, to enable or disable the button under particular circumstances rather than using the `IF` clause on this command. The specified ID is a number from 13 to 99 (1 - 12 are used for standard buttons). The id for a button is normally automatically allocated by the compiler and is one greater than the previous button. If you specify an id, it must not conflict with any previously automatically allocated id, and, if there are subsequent automatically allocated ids, there must be sufficient numbers available under 100 to allocate. It is strongly recommended that, if it is necessary to specify an id for a button, you specify ids for all user buttons on that form.

`IF (pql condition)`

Controls whether a button is enabled or disabled (greyed out). An action button is normally enabled. System buttons may be enabled and disabled depending on other processing. The specified condition is tested every time the data in the screen is updated and the button is enabled if the condition is true otherwise it is disabled.

PROMPT 'string'

Specifies the label on the button. If a user button is created and PROMPT is not specified, the prompt is USER BUTTON plus a unique number.

REMOVE

Removes (deletes) the button from the screen.

WIDTH

Specifies the width of the button.

GENERATE

```
GENERATE EXCLUDE | INCLUDE
          SCREENS (screennames, ...)
          VARS    (variable, ...)
```

General Clauses:

```
[NO]DATA    [AT r,c] [WIDTH n]
[NO]LABELS  [AT r,c] [WIDTH n]
[NO]PROMPT  [AT r,c] [WIDTH n] [VARDESC|VARLABEL|VARNAME]
ERROR number 'error text'
```

The `GENERATE` command in record or table screens uses the record or table schema to give the equivalent of default `FIELD` commands for every field. `GENERATE` in a menu screen produces the equivalent of a default `CALL SCREEN` to every previously defined record or table screen.

The `EXCLUDE` and `INCLUDE` clauses specify fields that are affected or not by the `GENERATE` and allow specific `FIELD` commands to be combined with a `GENERATE` command. When a `FIELD` command is specified for a variable, the variable should be `EXCLUDED` from the `GENERATE` otherwise it appears on the screen twice.

The general clauses on the `GENERATE` command apply to all fields generated by the command. The `DATA`, `LABELS` and `PROMPT AT` clause refers to the row position of the first field.

`EXCLUDE`

Omits screens or variables from the `GENERATE`. When `EXCLUDE` is specified, any variable not `EXCLUDED` is `INCLUDED` automatically (similarly any screen not `EXCLUDED` in a generated menu is `INCLUDED`).

`INCLUDE`

Includes screens or variables in the `GENERATE`. When `INCLUDE` is specified, any variables (or screens in menus) not `INCLUDED` are `EXCLUDED`.

SCREENS

Specifies screen names to INCLUDE or EXCLUDE in menu screens.

VARS

Specifies a list of named variables.

For example, to specify particular options for three fields and then bring in all the remaining fields in the EMPLOYEE record:

```
SCREEN RECORD EMPLOYEE/COMPANY.EMPLOYEE
TEXT 'This is the Demographic Record' AT +2
FIELD ID PROMPT 'Employee ID:'
FIELD NAME PROMPT 'Name of Employee'
FIELD SSN EDITOUT (COMPUTE FIELDOUT = EDIT (SSN, "^^^_^^_^^^"))
      EDITIN (COMPUTE SSN = REPLACE (FIELDIN, '-
', '', LEN(FIELDIN), 1, 0))
GENERATE EXCLUDE VARS (ID NAME SSN)
END SCREEN
```

Old Forms

There is an existing SirForms system that is a stand-alone system that uses a character style interface. While the PQLForms commands are different in design and operation, they can resemble old Forms. If comparing the PQLForms commands with old Forms note that:-

1. The basic syntax in PQLForms is VisualPQL and conforms to VisualPQL rules in terms of names, continuation rules, etc. For example, command continuation lines are recognised by a blank in the first position.
2. Some old Forms command names have been used but others conflicted with existing VisualPQL commands in that the same command syntax does different things in the two systems. PQLForms commands avoid these conflicts e.g. CALL SCREEN resembles the old Forms CALL; SCREEN RECORD resembles old Forms RECORD.
3. VisualPQL already contained commands that were equivalent to various functions in old Forms e.g. to define variables, to compute values, etc.
4. The intention is to use standard VisualPQL wherever possible so many options on commands take standard VisualPQL replacing specific old Forms keywords e.g. the EDITOUT and EDITIN clauses on the FIELD command (instead of keywords such as DSPEDIT); the INITIAL, READ and WRITE clauses on the SCREEN command (instead of keywords such as ACCEPT).
5. The VisualPQL LOOKUP command approximates this command in old Forms.
6. The Help system operates for fields. If screen level help is required, use the FBUTTON ACTION to define a help button.
7. The concept of groups of fields has been dropped.

8. The concept of separate forms based permissions has been dropped. The same security logins/permissions as for any use of VisualPQL now apply.

PQLForms Error Messages

Number	Text
27	Record not found
30	Record Written
37	Case not found
38	No more records
47	Not a valid value
57	Failed Edit tests
110	Record failed write tests
118	Record was modified - OK to save ?

PQLServer

The SIR/XS PQLServer is an executable that allows another standard SIR/XS session to connect to it as a client and to transmit commands to the server, execute those commands remotely and retrieve output. This is done with a set of PQL functions. The PQLServer must be started to enable clients to communicate to it across the network. The client processes do not require any access to files or databases that are local to the server and the two processes (client/server) may be using different hardware/operating systems e.g. client on windows, server on Unix.

From the client point of view processing is as follows:

- Client logs on to server and establishes a connection. This is the current connection used by all subsequent server functions until another logon/logoff.
- Client sends any number of lines of text including SIR commands. Control usually passes back without any actual transmission taking place - transmission only happens when maximum message size is reached.
- Client starts execution of previously sent commands. Any commands not yet sent to the server are transmitted, any settings or output from a previous execution from the same client are re-initialised, the commands are run and a completion code is returned at which point any output is waiting on the server. Commands can include all SIR commands and can use procedures, etc. Note that commands must include connecting any databases/tabfiles/procedure files needed each time commands are submitted and executed. There are no saved settings between executions.

The process may read/write files, update databases and generally do anything that a batch run of SIR could do.

- The client may choose to wait for the execution to finish or to carry on processing locally and subsequently test to find if the execution has completed successfully.
- Client gets count of number of lines of output and can then get each text line or skip over unwanted lines. Lines are physically passed by the server in groups. If skipping lines and the lines have not yet been transmitted, they are skipped on the server. Lines once returned or skipped are no longer available. The client can get a count of the number of lines available at any point.
- Client can repeat the process.
- Client logs off (specify a blank server) when finished.

e.g.

program

```
compute rc = serlog ('TONYDELL:4000','')
write rc
compute x = sersend ('PROGRAM')
compute x = sersend ('WRITE "HELLO WORLD"')
compute x = sersend ('END PROGRAM')
compute rc = serexec (1)
write 'rc = ' rc
compute olines = serlines(x)
write 'lines ' olines
for i=1,olines
. compute line = serget (0)
. write line
rof
compute rc = serlog ('','')
end program
```

See the Environment documentation for running the PQLServer.

Buffers

Buffers can be used to enter and edit unlimited amounts of text with minimal programming.

A program can invoke the editor, either the SIR editor or a system editor depending on parameter settings. Once the editor is invoked, control does not return to the program until the user exits the editor. The editor can use buffers to store data and there are VisualPQL commands to create, read and manipulate the contents of a buffer. The commands are:

- `CLEAR BUFFER`, `CREATE BUFFER` and `DELETE BUFFER` that affect the whole buffer.
- `DELETE LINE`, `GET LINE`, `INSERT LINE` and `PUT LINE` that affect individual lines in the buffer.
- `EDIT BUFFER` that passes control to the editor for the user to edit the buffer. Control returns to the program when the user exits the editor.

CLEAR BUFFER

```
CLEAR BUFFER buffer_name_exp
```

Removes all the lines currently in the specified buffer. Specify an existing buffer name as a string constant in quotes or as a string variable.

CREATE BUFFER

```
CREATE BUFFER buffer_name_exp
```

Creates a new, empty buffer. Specify the buffer name as a string constant in quotes or as a string variable.

If the buffer already exists, a warning is issued but the program continues.

DELETE BUFFER

```
DELETE BUFFER buffer_name_exp
```

Removes the specified buffer. Specify the buffer name as a string constant in quotes or as a string variable. If the buffer does not exist the command is ignored and no warning is issued.

DELETE LINE IN BUFFER

```
DELETE LINE IN BUFFER buffer_name_exp  
    NUMBERED num_value
```

Removes a specific line in the buffer. Subsequent lines are renumbered. Specify the buffer name as a string constant in quotes or as a string variable.

EDIT BUFFER

```
EDIT BUFFER buffer_name_exp
```

Invokes the SIR editor or the external editor with the specified buffer. Specify the buffer name as a string constant in quotes or as a string variable.

GET LINE FROM BUFFER

```
GET LINE FROM BUFFER buffer_name  
    NUMBERED num_value  
    INTO string_var
```

Transfers a copy of the specified line to a string variable. If the line number is greater than the number of lines in the buffer, the string is set to undefined.

INSERT LINE INTO BUFFER

```
INSERT LINE INTO BUFFER buffer_name  
    NUMBERED num_value  
    FROM string_var
```

Inserts a new line into the buffer before the specified line number. That is the old line with the specified line number becomes that line number+1 and the new line becomes the specified line number.

PUT LINE TO BUFFER

```
PUT LINE TO BUFFER buffer_name
    NUMBERED num_value
    FROM string_var
```

Replaces the specified line in the specified buffer with the contents of the string argument specified.

Example of Buffer Manipulation

The following example uses a bibliographic database in which abstracts of books are stored. The case identifier variable is `BOOKID`, a string. A record type called `ABSTRACT` has an integer keyfield called `LINENUM` and an 80 character string variable called `TEXTLINE`. Each line of text of the abstract is stored as a record in this record type.

The retrieval has two parts, the control structure of the program and a set of subprocedures that do the various program tasks such as looking for existing abstracts, editing the abstract and saving the abstract in the database.

```
RETRIEVAL UPDATE NOAUTOCASE
STRING * 80 TMLINE
INTEGER * 2 EDITEND
CREATE BUFFER 'ABSTRACT'
LOOP
structure
. ERASE SCREEN
. DISPLAY TEXTBOX 'Enter Book ID:'
    RESPONSE RESVAR, BOOKID
. IFTHEN(RESVAR LE 0)
. DELETE BUFFER 'ABSTRACT'
. EXIT RETRIEVAL
. ELSE
. EXECUTE SUBPROCEDURE GETBOOK
. EXECUTE SUBPROCEDURE EDITBOOK
. IFTHEN(EDITEND = 299)
    CLEAR BUFFER 'ABSTRACT'
. NEXT LOOP
. ELSEIF(EDITEND = 277)
. EXECUTE SUBPROCEDURE SAVEBOOK
. NEXT LOOP
. END IF
. END IF
END LOOP

C** -- subprocedure definitions
```

```

SUBPROCEDURE GETBOOK          | gets abstract from db
OLD CASE IS BOOKID
. PROCESS REC ABSTRACT
. INSERT LINE INTO BUFFER 'ABSTRACT' | load lines into buffer
      numbered LINENUM from TEXTLINE
. END REC
END CASE
IFTHEN (SYSTEM(14) = 0)      | if book is not in database
. DISPLAY ERRBOX 'New Book'  | give a message
ENDIF
END SUBPROCEDURE

SUBPROCEDURE EDITBOOK        | edit the abstract
EDIT BUFFER 'ABSTRACT'
END SUBPROCEDURE

SUBPROCEDURE SAVEBOOK        | store text in database
CASE IS BOOKID               | create or access book
SET LINENUM (0)              | initialise line counter
LOOP                           | go thru lines in buffer
. LINENUM = LINENUM + 1
. GET LINE FROM BUFFER 'ABSTRACT' numbered LINENUM into      TMLINE
. IFTHEN(EXISTS(TMLINE)=1)
.   RECORD IS ABSTRACT (LINENUM)
.   PUT VARS TEXTLINE = TMLINE
.   END REC
. ELSEIF(EXISTS(TMLINE)=0)   | if (end of buffer)
.   PROCESS REC ABSTRACT    | go thru any other records
      FROM (LINENUM)
.   DELETE REC              | and delete the record
.   END REC
.   EXIT LOOP
. END IF
END LOOP
CLEAR BUFFER 'ABSTRACT'
END CASE
END SUBPROCEDURE
END RETRIEVAL

```

DISPLAY WDL

```
DISPLAY WDL { 'string_val' | varname }
```

Sends either the specified string constant (in quotes) or the contents of the specified variable to the OutputHandler callback routine in SirAPI. The variable must be a string.

Functions

Functions return a single numeric or string value derived from the arguments of the function. Arguments are separated by commas. In general, the functions can appear in any string, arithmetic or logical expression in a VisualPQL program. Schema functions can be used in a `PROGRAM`. The functions are listed by type and in alphabetical order with a full explanation of each.

List of Functions by Type

Function types are:

- Trigonometric
- Mathematical
- Argument List
- Across record
- Date and Time
- Global and Parameter
- String
- Concurrent
- Miscellaneous
- Session
- Schema & Database
- Tabfile & Table
- Read/Write
- Dialog and Menu
- Client/Server
- CGI

For a detailed description of all functions, see the alphabetic list of functions.

Trigonometric Functions

ACOS arc cosine (also ARCCOS)

ASIN arc sine (also ARSIN)

ATAN arc tangent

COS trigonometric cosine

SIN trigonometric sine

TAN trigonometric tangent

TANH hyperbolic tangent

Mathematical Functions

ABS absolute value

AINT truncation (also TRUNC)

ALOG natural logarithm (also LN or LOG)

ALOG10 base 10 logarithm (also LG10 or LOG10)

AMOD remainder of division (also MOD)

EXP exponentiation (base e)

FEQ compares two floating point numbers within a tolerance

RAND random uniform number (0-1) (also RANF)

REAL4 returns the REAL*4 value of a REAL*8 number

RND rounding

SIGN transfer of sign

SQRT square root

TRUNC truncates least significant digits

Argument List Functions

CNT count the number of arguments that are not missing

FST return the first argument that is not missing

LST return the last argument that is not missing

MAX return the largest argument that is not missing

MEAN compute the mean of the arguments that are not missing

MIN return the smallest argument that is not missing

STDEV compute the standard deviation of non missing values

SUM compute the sum of all arguments that are not missing

Across Record Functions

The "across records" functions may only appear in `PROCESS REC` or `PROCESS ROW` blocks. They compute a result based on a single variable in each record or row processed in the `PROCESS REC` or `PROCESS ROW` block. They ignore values that are missing or undefined. Records that contain missing or undefined values are not counted nor are they used in the calculation of averages. Some of these functions can be used with string values, others do not apply to strings. If a function returns a string (e.g. as a maximum), a maximum of 32 characters are returned.

`CNTR`, `FSTR`, `LSTR`, `MAXR`, `MINR`, `CNT`, `FST`, `LST`, `MAX`, `MIN` can all be used with strings. `MEANR`, `STDEV`, `SUMR`, `AMOD`, `MEAN`, `STDVEV`, `SUM` are only relevant to numeric values.

`CNTR` counts the number of times the variable occurs.

`FSTR` returns the first value processed.

`LSTR` returns the last value processed

`MAXR` returns the largest value processed

`MEANR` computes the average value ($SUMR / CNTR$)

`MINR` returns the smallest value

`STDEV` computes the standard deviation

`SUMR` computes the sum of values

Date and Time Functions

CDATE converts a date string to a date integer

CTIME converts a time string to a time integer

DATEC converts a date integer to a date string

DATET returns the current date and time as a string

DTTOTS takes a date and a time integer and returns a *Timestamp* as a real*8 value

JULC converts a date integer to a date string

JULN converts day, month, and year to a date integer

NOW returns the current time as a time integer

TIME converts hours, minutes, and seconds to a time integer

TIMEC converts a time integer to a time string

TODAY returns current date as a date integer

TSDODT takes a *Timestamp* and returns the date component as an integer

TSDOTM takes a *Timestamp* and returns the time component as an integer

Global Functions

DGLOBAL Deletes a global variable

DSN Returns a full filename associated with an attribute

GLOBALN Assigns a numeric value to a global variable

GLOBALS Assigns a string value to a global variable

NARG Returns a numeric argument from run parameter list

NGLOBAL Returns the value of a global numeric variable

SARG Returns a string argument from run parameter list

SGLOBAL Returns the value of a global string variable

String Functions

- CAPITAL** Capitalises the first letter of each word in string
- CATINT** Returns an integer value of a categorical variable
- CATSTR** Returns a string value of a categorical variable
- CENTER** Returns a centred string
- CHAR** Returns the character with the numeric internal value
- COMMA** Separates thousands by inserting commas in a numeric string
- EDIT** Applies editing template to a data string
- FILL** Replaces blanks in string
- FORMAT** Converts a number to a string
- ICHAR** Returns the numeric internal value of a character
- LEN** Returns the string length in characters
- LOWER** Converts all characters in string to lower case
- NUMBR** Converts a string to a number
- PACK** Returns the string with compressed blanks
- PAD** Pads a string with character to specified length
- PATTERN** Returns whether a pattern is found in a string
- PFORMAT** Converts a number to a formatted string
- PICTURE** Validates a string by comparing to a picture
- REGEXP** Searches a string for a substring specified by a regular expression
- REGREP** Searches a string for a substring specified by a regular expression and replaces it according to a second regular expression

REPLACE Replaces substrings with a specified string

REVERSE Returns a string spelled backwards

SBST Returns a substring of a string

SGET Returns the value of a string variable

SPREAD Returns a string with single blanks between characters

SPUT Stores string value in string variable

SRST Searches for a substring

SUBSTR Returns a substring of a string

TRIM Trims trailing blanks from a string

TRIML Deletes blanks from the left

TRIMLR Deletes blanks from the left and the right

TRIMR Deletes blanks from the right

UPPER Converts all characters of string to upper case

VARGET Gets value from string variable where variable name is an expression.

VARPUT Puts value into string variable where variable name is an expression.

Concurrent Functions

CASELOCK Change lock type for and retry current CIR (also CIRLOCK)

RECLOCK Change lock type for and retry current record

SYSTEM(36) Whether current record is locked

SYSTEM(37) Whether current CIR is locked

SYSTEM(38) Whether a concurrent session using Master

SYSTEM(39) Returns the ordinal number of the default database. No Database returns 0

Miscellaneous Functions

ARRDIMN Returns number of dimensions of a local array variable

ARRDIMST Returns start value of a dimension of a local array variable

ARRDIMSZ Returns number of entries of a dimension of a local array variable

CLIPAPP Adds text to the clipboard

CLIPGET Gets text from the clipboard

CLIPLINE Gets count of lines in the clipboard

CLIPSET Clears the clipboard and adds text to the clipboard

CRYPTKEY Sets the key for the encryption functions.

DECRYPT Decrypts an encrypted string.

ENCRYPT Encrypts a string.

ERROR Displays a text message error box

EXISTS Indicates if variable exists (not missing or undefined)

EXTERN Invokes user-supplied external functions returning a numeric value. These must be in the **EXTERN** dll.

EXTERN_S Invokes user-supplied external functions returning a string value. These must be in the **EXTERN** dll.

HELP Invokes the Help system

MISNUM Returns the "type" of missing or undefined value

MISSING Returns the original value for missing values

MSGTXT Returns the error message text for an error number

PROGRESS Controls the display of a progress bar

SEEK Sets a position on an open file.

SRCH Searches a table of values

STDNAME Checks a name and wraps curly brackets around if it is a non-standard name.

SYSTEM Returns various system values such as CPU time used, whether a database access has been successful, etc.

YESNO Displays a text question box and returns response

Session Functions

APPDIR Returns application directory

ATTRNAME Attribute n name (str)

BUFNAME Buffer n name

CURDIR Returns current directory

DEFFAM Default family name

DEFMEM Default member name

DEFTFN Default tabfile name

DELDIR Deletes the named directory

DELFILE Deletes the named file

EDITNAME Editor name

FAMNAME Family n name

FILECNT Returns a count of files in named directory

FILEIN Browses for a file

FILEIS Returns if name exists as file or directory

FILEN Returns the nth file in directory

FILEOUT Browses for an output file

FILESTAT Returns various data about named file e.g. Date/time of creation/access, size of file, etc. Times and dates are in system format

FILETIME Returns various data about times of creation/access of file with times and dates converted to SIR formats

GETENV Gets a string environment variable value

GLOBNAME Gets the name of the nth global variable

LINES Lines remaining on output page

MAKEDIR Creates a directory

MEMCOUNT Count of members in family

MEMINFO Various information about a member

MEMNAME Name of nth member

NSUBDIR Name of nth subdirectory

OUTFNAME Name of current output file

PAGELEN Length of output page

PAGENO Current output page number

PAGEWID Width of output page

PROCFILE Procedure file filename

PROCNAME Procedure file attribute name

RACCESS Returns the read access level of user

RNMFILE Renames a file

SETDIR Sets current directory

SETRC Sets a return code

SIRUSER Sets/returns the current user name

SUBDIR Concatenates a subdirectory to existing directory path in correct system specific manner

SYSTEM Returns various "system" status values

UPGET Gets User Preference (from INI file)

UPSET Sets User Preference (in INI file)

WACCESS Returns the write access level of user

WINCNT Returns the number of lines in the output window

WINLIN Returns the nth line from the output window

WINMOVE Moves and resizes the main window

WINPOS Moves to line in output window

WINSELL Returns selected line from output window

WINSELP Returns position in line selected from output window

Schema & Database Functions

- COUNT** Number of records of a given type in the current case
- DATEMAP** Returns the date format (map) of a date variable
- DBINDN** Name of nth index on default database
- DBINDR** Number of record type indexed
- DBINDS** Number of indexes on default database
- DBINDT** Name of the nth variable in index followed by either ASC or DESC and UPPER if uppercase index
- DBINDU** Returns 1 if index is unique
- DBINDV** Number of variables in index
- DBNAME** Name of the nth database
- DBTYPE** Returns database type, case or caseless
- JOUFLAG** Returns whether journaling is on for default database
- KEYNAME** Returns name of a record type key field
- KEYORDER** Returns sort order ("A" or "D") of a key field
- MAXRECS** Returns the maximum allowed number of a particular record type
- MISS** Returns the original (string) for missing values (0 - 3) for a variable
- MKEYSIZE** Returns the size of the largest record key in the database
- MRECSIZE** Returns size of the largest record in the database
- NKEYS** Returns the number of key fields for a record type
- NLABELS** Synonym for NVALLAB. Returns the number of variable value labels defined for a variable
- NMAX** Returns the maximum value of a numeric variable range

NMIN Returns the minimum value of a numeric variable range

NOFCASES Returns the maximum number of cases for the database

NRECS Returns the maximum number of record types for the database

NUMCASES Returns the number of cases in the database

NUMRECS Returns the number of records of a given type in the database

NVALID Returns the number of defined valid values for a variable

NVALLAB Returns the number of variable value labels defined for a variable

NVARDOC Returns the number of lines of variable documentation defined for a variable

NVARS Returns the number of variables for the specified record type

NVARSC Returns the number of variables including common vars for the specified record type

NVVAL Returns the nth valid value of a numeric variable

RECDOC Returns the nth line of documentation for a record or database

RECDOCN Returns the number of lines of documentation for a record

RECLEVEL Returns the update level at which a record was last written to database

RECNAME Returns the name of the record type

RECNUM Returns the number of the record name

RECSIZE Returns the record size of a specified record type

RKEYSIZE Returns the key size of a specified record type

RRECSEC Returns the read security level of a record type

RVARSEC Returns the read security level of a variable

SMAX Returns the maximum value of a string variable range

SMIN Returns the minimum value of a string variable range

STATTYPE Returns whether a numeric variable is defined as observation, control or neither

SVVAL Returns the nth defined string valid value of a variable

TIMEMAP Returns the time format (map) string format of a time variable

UPDLEVEL Returns the current update level of database

VALIDATE Validates a value of a database variable against schema

VALLAB Returns the value label for the current value of a variable

VALLABSC Returns the value label for a specified value of a variable

VALLABSN Returns the nth value label of a variable

VALLABSP Returns the number (nth) of the value label associated with a specified value of a variable.

VALLABSV Returns a string that is the nth value associated with value labels of a variable.

VARLAB Returns the variable label for a variable (up to 78 characters)

VARLABSC Returns the variable label of a specified variable (up to 78 characters)

VARDOCSN Returns the nth line of variable documentation of a specified variable

VARNAME Returns the name of the variable using counts *excluding* common vars.

VARNAMEC Returns the name of the variable using counts *including* common vars.

VARPOSIT Returns the input position of the variable

VARTYPE Returns the storage type of a variable (string or numeric)

VFORMAT Returns a string representing the variable input format

VTTYPE Returns the SIR data type of a variable (7 types)

WRECSEC Returns the write security level of a record type

WVARSEC Returns the write security level of a variable

Tabfile & Table Functions

TABINDN Index name of nth index

TABINDS Number of indexes on nth table

TABINDT Variable name and sequence of nth variable on index

TABINDU If nth index is unique

TABINDV Number of variables in nth index

TABNAME Name of nth table

TABRECS Number of rows on nth table

TABVARS Number of cols (variables) in nth table

TABVINFN Various numeric data about column

TABVINFS Various string data about column

TABVNAME Column name

TABVRANG Value of valid/missing range for column

TABVTYPE Column type

TABVVALI Validates table column

TABVVLAB Value label for table column

TABVVVAL Value label value for table column

TFACCESS Access a(uto),r(ead) w(rite) of nth tabfile

TFATTR Internal attribute name of nth tabfile

TFCOUNT Number of connected tabfiles

TFFILE Filename of nth tabfile

TFGRNAME Group name of nth tabfile

TFGRPW Group password of nth tabfile

TFJNNAME Journal name of nth tabfile

TFNAME Name of nth tabfile

TFTABS Number of tables on nth tabfile

TFUSNAME User name of nth tabfile

TFUSPW User password of nth tabfile

Read/Write Functions

NGET Gets the value of a numeric variable

NPUT Stores a value in a numeric variable

NREAD Pops up a box on the screen with a prompt and returns a number from the user

SGET Gets the value of a string variable

SPUT Stores a value in a string variable

SREAD Pops up a box on the screen with a prompt and returns a string from the user

TWRITE Writes a string to the scrolled output window

Dialog & Menu Functions

BRANCH Adds a branch to the tree at a particular place.

BRANCHD Deletes a branch from the tree.

BRANCHN Returns id of nth branch.

FINDITEM Search list for text

GETBTNH Returns the height of button control

GETCHCH Returns the height of choice control

GETCHKH Returns the height of check control

GETFLT Gets floating point as per GETTXT

GETFOCUS Returns id of control with focus

GETCHK Returns check or radio state

GETIFLT Returns floating point from a list

GETIINT Returns integer of item from a list

GETINT Gets integer as per GETTXT. 0 if not integer

GETITXT Returns text of item from a list

GETLBLH Returns the height of labels

GETLTXG Gets the text from a line in a multi-line text control

GETMAXCH Returns the height of the maximum single line control

GETMCHK Tests the state of a menu or toolbar item

GETMSEL Returns pos of nth selected item from multiple selection

GETNITEM Returns number of items in choice or list

GETNLIN Gets number of lines in multi-line text control

GETNSEL Returns number of items selected in multiple selection

GETPOS Returns pos of current selection in list or choice or keyboard focus in multiple selection

GETRADH Returns the height of radio control

GETRSTEP Returns the size of the row step

GETTXT Gets text from edit and from highlighted item in choice or list. (Gets label text from label, button, check and radio)

GETTXTH Returns the height of text control

IDSTATUS Returns the status of a control

NBRANCH Returns number of braches in a tree below a point.

SCROLLAT Gets a position in a gui scrollable item

SCROLLTO Sets a position in a gui scrollable item

SETPOS Sets a position in a gui multi-line item

SETRANGE Sets minimum/maximum values in a gui slider/spin/progress control. Sets the maximum number of characters allowed in an edit or text field (ignoring the minimum parameter).

Dialog Editor

DITEM. . . There are a number of functions that return information about items on a DEDIT dialog, that is a dialog used for screen painting. These functions are all named DITEMxxx.

Client/Server Functions

There are three servers in SIR/XS and a client program may be communicating with one of those or with an ODBC server from another software supplier. The three SIR/XS servers are *Master* that controls concurrent updates, the *SQLServer*, that acts as an ODBC server for other packages, and the *PQLServer* that is a server that can run any SIR/XS processes requested by a client. Most communication with master is done transparently i.e. it does not require specific functions, however there are a set of functions that can be used to administer master if required.

Client Functions to administer Master

DELMCLID Deletes the client from Master.

GETAKL Returns the client AutoKill Limit from Master.

GETDFC Returns the difference file copy interval from Master.

GETMCADD Returns the client tcp/ip address from Master.

GETMCLID Returns the nth client id from Master.

GETMCLST Returns the time of last message for nth client from Master.

GETMCON Returns the time of log on for nth client from Master.

GETMDBN Returns the name for nth database from Master.

SETAKL Sets the AutoKill time Limit for master clients.

SETDFC Sets the difference file copy interval for Master.

Client Functions to SQLServer/ODBC

BINDPARM Binds an SQLServer/ODBC parameter.

COLCOUNT Returns a count of columns created by the execute

COLLABEL Returns the label of a specific column created by the execute

COLLEN Returns the length of a specific string column created by the execute

COLNAME Returns the name of a specific column created by the execute

COLTYPE Returns the type of a specific column created by the execute

COLVALN Returns the numeric value of a specific column created by the execute

COLVALS Returns the string value of a specific column created by the execute

GETERR Returns the oldest error posted for this application and deletes the message.

NEXTROW Steps through the rows one at a time

ODBTABS Produce list of tables on the data source

ODBCCOLS Produce list of columns from the named table on the data source

ROWCOUNT Count of rows created by the execute

Client Functions to PQLServer

SERADMIN Various server administration capabilities (returning numeric values)

SERADMIS Various server administration capabilities (returning string values)

SEREXEC Instructs server to execute previously sent commands

SERGET Gets a line of output from server

SERLINES Asks server how many lines of output are left

SERLOG Logs on to the server

SERSEND Sends a string to the server

SERSENDB Sends a buffer to the server

SERTEST Asks server if execution has completed

PQLServer Functions

(These have no effect if used in a program that is not running on the server)

SERNOOUT Suppresses server output

SERWRITE Writes a line of output from server

CGI Functions

Buffer functions return number of lines. All parameters are string expressions.

CGIBUFPN Get buffer of value of parameter

CGIBUFPN Get buffer of value of parameter

CGIBUFSV Get buffer of value of server variable

CGIVARPV Get parameter value

CGIVARSV Get server variable value

List of Functions from A to Z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

ABS

num = ABS(X)

Returns the absolute value of X.

ACOS

num = ACOS(X)

Returns the arc cosine of X. The result is in the range 0 to PI radians. Values of X outside the range $-1 \leq X \leq 1$ return undefined.

AINT

num = AINT(X [,n])

Returns the truncated value of X. If n is omitted or 0, truncates to an integer value. If n is specified, truncates to that power of 10. e.g. AINT (1266,2) truncates to 1200, AINT(1.266,-2) truncates to 1.26.

ALOG

num = ALOG(X)

Returns the natural logarithm (base e) of X. Values less than or equal to zero return undefined.

ALOG10

num = ALOG10(X)

Returns the base 10 logarithm of X. Values less than or equal to zero return undefined.

AMOD

num = AMOD(X , Y)

Returns the remainder of X divided by Y. If Y is zero, undefined is returned.

APPDIR

```
str = APPDIR(0)
```

Returns SIR/XS application directory. That is the directory where the SIR executables are installed.

ARCOS

```
num = ARCOS( X )
```

See ACOS function.

ARRDIMN

```
num = ARRDIMN(array_name_exp)
```

Returns the number of dimensions for specified local array variable.

ARRDIMST

```
num = ARRDIMST(array_name_exp,dim)
```

Returns the start value for the specified dimension for specified local array variable (normally 1 unless array specified to start at different value).

ARRDIMSZ

```
num = ARRDIMSZ(array_name_exp,dim)
```

Returns the size of the specified dimension for specified local array variable.

ARCOS

```
num = ARCOS( X )
```

See ACOS function.

ARSIN

```
num = ARSIN( X )
```

See ASIN function.

ASIN

```
num = ASIN( X )
```

Returns the arc sine of X. The result is in the range $-\pi/2$ to $+\pi/2$ radians. Values outside the range $-1 \leq X \leq 1$ return undefined.

ATAN

```
num = ATAN( X )
```

Returns the arctangent of X. The result is in the range $-\pi/2$ to $+\pi/2$ radians.

ATTRNAME

```
str = ATTRNAME(n)
```

Returns the nth Attribute name (SYSTEM(52) = attribute count).

BINDPARM

```
num = BINDPARM(conid,statid,paramno,num_exp)
```

Binds an sqlserver/odbc parameter.

BRANCH

```
num = BRANCH(control_id,parent_id,node_id,text)
```

Creates a new node under the given parent node in the tree control.

BRANCHD

```
num = BRANCHD(control_id,node)
```

Deletes a branch in the tree control given by the node.

BRANCHN

```
num = BRANCHN(control_id,node,n)
```

Returns id of nth child of the given node.

BUFNAME

```
str = BUFNAME(n)
```

Returns the nth Buffer name (SYSTEM(56) = buffer count).

CAPITAL

```
str = CAPITAL( str )
```

Capitalises the first alphabetic character of the string and the first alphabetic character following a blank. All other characters remain unedited. For example:

```
NAME = 'this is the first day of the week'  
NAME = CAPITAL(NAME)  
Returns: This Is The First Day Of The Week
```

CASELOCK

```
num = CASELOCK(locktype)
```

Changes the lock type for the current Case (CIR) and attempts to read the current CIR from the database. See `SYSTEM(37)` function to determine if current CIR is locked. The locktype codes are (any other values set concurrent read):

- 1 = Exclusive
- 2 = Concurrent Read
- 3 = Concurrent Write
- 4 = Protected Read
- 5 = Protected Write
- 6 = Exclusive

CATINT

```
num = CATINT(A, B)
```

Returns an integer corresponding to the category in categorical variable A that B matches. A and B may be variables, string constants or expressions. Returns a zero if no match is found.

CATSTR

```
str = CATSTR(A)
```

Returns a string corresponding to the current value of the specified categorical variable.

CDATE

```
num = CDATE(X, date format)
```

Returns the date integer equivalent to the date string X that may be a string constant, variable or expression. The date format is a string expression. See date formats for a complete description. If a date earlier than October 15, 1582 is specified, undefined is returned. Example:

```
INTDATE = CDATE('6/3/7', 'MM/DD/YY')
```

CENTER

```
str = CENTER(X, strY)
```

Returns string strY centred in a string X characters in length. Example:

```
RESULT = CENTER(9, 'ABC')
```

```
returns: "   ABC   "
```

CGIBUFPN

```
n = CGIBUFPN(buf, pn)
```

Used when dealing with CGI from webserver. Puts value of CGI parameter into named buffer. Returns number of lines. Parameters are string expressions.

CGIBUFSV

```
n = CGIBUFSV(buf, sv)
```

Used when dealing with CGI from webserver. Puts value of server variable into named buffer. Returns number of lines. Parameters are string expressions.

CGIVARPN

```
str = CGIVARPN(pn)
```

Used when dealing with CGI from webserver. Returns named parameter value in string variable. Parameter name is a string expression.

CGIVARSV

```
str = CGIVARSV(sv)
```

Used when dealing with CGI from webserver. Returns named server variable value in string variable. Server variable name is a string expression.

CHAR

```
str = CHAR(N)
```

Returns a single character. The character returned is the character with the internal value of N. (See the `ICHAR` function.) If N is larger than 255, N is divided by 256 and the remainder is taken. This gives the set of standard characters. If N is missing, a null string is returned (length 0).

CIRLOCK

See `CASELOCK` function

CLIPAPP

```
num = CLIPAPP(text)
```

Appends text to the clipboard (the place holding text you cut and paste).

CLIPGET

```
str = CLIPGET(line)
```

Gets a line of text from the clipboard (the place holding text you cut and paste).

CLIPLINE

```
num = CLIPLINE(dummy)
```

Gets number of lines of text currently in the clipboard (the place holding text you cut and paste).

CLIPSET

```
num = CLIPSET(text)
```

Clears the clipboard and puts text into the clipboard (the place holding text you cut and paste).

CNT

```
num = CNT(X1 , X, . . . . . , Xn)
```

Counts the number of values in a list that exist (not missing or undefined). There may be up to 128 variables in the list. Returns zero (0) if no values exist.

CNTR

```
num = CNTR( X )
```

Returns the number of values of X found during a `PROCESS REC` or `PROCESS ROWS` loop that are not missing or undefined. Returns zero if all values are missing or undefined.

COLCOUNT

```
num = COLCOUNT (conid,statid)
```

Client/Server function. Returns a count of columns created by the execute.

COLLABEL

```
str = COLLABEL (conid,statid,colno)
```

Client/Server function. Returns the label of a specific column created by the execute.

COLLEN

```
num = COLLEN (conid,statid,colno)
```

Client/Server function. Returns the length of a specific string column created by the execute.

COLNAME

```
str = COLNAME (conid,statid,colno)
```

Client/Server function. Returns the name of a specific column created by the execute.

COLTYPE

```
num = COLTYPE (conid,statid,colno)
```

Client/Server function. Returns the type of a specific column created by the execute. This is one of the following:

1 = String

2 = Timestamp string

3 = Date

4 = Time

5 = Integer

6 = R4

7 = R8

8 = Scaled Integer

A timestamp string is a formatted 18 byte string containing a combination date/time as follows:

YYYYMMDDHHMMSSttt where YYYY is the year, MM is the month, DD is the day number, HH is the 24 hour number, MM is the minutes, SS is the seconds and tttt is the thousandths of a second. Any of these components may be zero.

COLVALN

```
num = COLVALN (conid,statid,rowno,colno)
```

Client/Server function. Returns the numeric value of a specific column created by the execute. This does not have to be the same type as returned by COLTYPE. e.g. Integers can be assigned to a real.

COLVALS

```
str = COLVALS (conid,statid,rowno,colno)
```

Client/Server function. Returns the string value of a specific column created by the execute.

COMMA

```
str = COMMA( str )
```

Places a comma between every third digit to the left of a decimal point (actual or implied) in a string with a numeric form. For example, `COMMA ('4500000')` returns '4,500,000'.

COS

```
num = COS( X )
```

Returns the trigonometric cosine of X, where X is measured in radians.

CRYPTKEY

```
num = CRYPTKEY( encryption_key )
```

Sets the key used by the encryption functions `ENCRYPT/DECRYPT`. The key is a 256 bit (32 character) string. The key only needs to be set once in a session that uses the encryption functions. If the key is not set, the encryption functions use a key of all blanks.

The specified key is an expression. i.e. a variable name or a string in quotes.

COUNT

```
num = COUNT( X )
```

Returns the number of records for record type X within the current case.

CTIME

```
num = CTIME( X , time format )
```

Returns the number of seconds from midnight to the given time. Specify a time string X and a time format. See time formats for a complete description. If the hour, minute or second field is omitted, they default to zero. For example:

```
WINTIME = CTIME( '18:36:45' , 'HH:MM:SS' )
```

CURDIR

```
str = CURDIR( 0 )
```

Returns the current directory.

DATEC

```
str = DATEC( X , date format )
```

Returns a date string equivalent to the date integer X formatted according to the date format. The format is a string constant. See date formats for a complete description. For

example, DATEC(XBEG, 'Www, Mmm DDth, YYYY') produces a result such as "Thu, May 25th, 2007"

DATEMAP

```
str = DATEMAP (rtnum, varname_exp)
```

Returns a string with the date format (map) of the specified date variable. If the record number (rtnum) is negative, the function applies to a summary variable; if rtnum is one more than the maximum record count (i.e. NRECS(0)+1) then this applies to a standard variable.

The varname is an expression. If this is a constant, enclose the name in single quotes. Undefined is returned if the variable is unknown or is not a date variable. For example, if the variable BIRTHDAY in record type one has the date format "MM DD YY" defined in the schema, then DATESTR equals "MM DD YY".

```
DATESTR = DATEMAP (1, 'BIRTHDAY')
```

DATET

```
str = DATET(N1 , N)
```

Returns a 27-character string containing the current date and time. The string is composed of the following substrings:

Date

```
1- 3 Day of the week (SUN,MON,etc.)
4- 5 Comma and blank
6- 8 Month of the year (JAN,FEB,etc.)
9 Blank
10-11 Day of the month
12-13 Comma and blank
14-17 Year
18-19 Comma and blank
```

Time

```
20-21 Hour (1 to 12)
22 Period
23-24 Minutes
25 Blank
26-27 AM or PM
```

The two arguments N1 and N are constants in the range 1 to 27 that select a substring of the 27-character string. For example, suppose the current date is May 25, 2000, and the time is 1:05 PM; the day is Thursday.

PROGRAM

```
TODATE = DATET (6, 17)
NOWTIME = DATET(20, 27)
WKDAY = DATET(1, 3)
```

```
WRITE TODATE NOWTIME WKDAY  
END PROGRAM
```

```
OUTPUT:  MAY 25, 2000 01.05 PM THU
```

DTTOTS

```
real*8 = DTTOTS (date,time)
```

Takes a date and time integer and returns a *timestamp*. A timestamp is a real*8 representation and is the number of seconds since the start of the SIR/XS calendar. You can do calculations between timestamps but the individual date and time components must be extracted using the `TSTODT` and `TSTOTM` functions before using any other date and time functions e.g. for print formatting.

DBINDN

```
str = DBINDN (index)
```

Returns name of nth index.

DBINDR

```
n = DBINDR (index)
```

Returns number of record type indexed by nth index.

DBINDS

```
n = DBINDS (dummy)
```

Returns number of indexes on default database.

DBINDT

```
str = DBINDT (index,varno)
```

Name of nth variable in nth index plus ASC/DESC and UPPER.

DBINDU

```
n = DBINDU (index)
```

Returns 1 if the index is unique or 0 if the index is not unique.

DBINDV

```
n = DBINDV (index)
```

Returns number of variables in nth index.

DBNAME

```
str = DBNAME (n)
```

Returns a string with the name of the nth attached database. If n is zero, returns the name of the default database.

DBTYPE

```
num = DBTYPE (dummy)
```

Returns 1 if a case structured database or 0 if caseless.

DECRYPT

```
str = DECRYPT(string,length)
```

Decrypts an encrypted string. Set the encryption key (using `CRYPTKEY` prior to the first invocation of this function. Obviously the key must be the same as was used to encrypt the string.

DEFFAM

```
str = DEFFAM(0)
```

Returns the default family name.

DEFMEM

```
str = DEFMEM(0)
```

Returns the default member name.

DEFTFN

```
str = DEFTFN(0)
```

Returns the default tabfile name.

DELDIR

```
n = DELDIR(dir_name)
```

Deletes the named directory. Returns 0 for success.

DELFILE

```
n = DELFILE(file_name)
```

Deletes the named file (use filename not attribute). Returns 0 for success.

DELMCLID

```
str = DELMCLID(id,password)
```

Deletes the client from master. Get the client id from GETMCLID. Specify a password as a string variable or string in quotes if the Master is started with a password.

DGLOBAL

```
num = DGLOBAL(string_exp)
```

Deletes a global. The string expression may be the global name enclosed in quotes or a string variable.

DITEM

DITEM...

The DITEM series of functions all pertain to the *Dialog Editor* used to construct screen painting applications.

DITEMCOL

num = DITEMCOL(n) Returns the column the nth DEDIT item is positioned at.

DITEMH

num = DITEMH(n) Returns the height of the nth DEDIT item.

DITEMID

num = DITEMID(n) Returns the id of the nth DEDIT item.

DITEMROW

num = DITEMROW(n) Returns the row the nth DEDIT item is positioned at.

DITEMS

num = DITEMS(0) Returns the number of items on DEDIT dialog.

DITEMSEL

num = DITEMSEL(0) Returns the number of items selected on DEDIT dialog.

DITEMSID

num = DITEMSID(n) Returns the id of the nth selected DEDIT item.

DITEMTXT

str = DITEMTXT(n) Returns the text of the nth DEDIT item.

DITEMTYP

num = DITEMTYP(n) Returns the type of control of the nth DEDIT item.

DITEMW

num = DITEMW(n) Returns the width of the nth DEDIT item.

DSN

str = DSN(string_exp)

Returns the operating system filename of an attribute. The attribute may be a variable or constant.

EDIT

output_str = EDIT(input_str, edit_str)

EDIT applies an edit string to data to produce the output. The edit string is made up of circumflexes (^), that represent a character of the input string, and any other characters to insert. Example:

```
PROGRAM
SSN = EDIT( '123456789', '^^^_^^_^^^' )
WRITE SSN
END PROGRAM
```

Output: 123-45-6789

EDITNAME

str = EDITNAME(0)

Returns the name of the current text editor.

ENCRYPT

str = ENCRYPT(string,length)

Encrypts a string. Set the encryption key (using `CRYPTKEY`) prior to the first invocation of this function.

The string is a SIR string and the encrypted string is also a normal SIR string but it should be noted that encryption may result in non-text characters and so encrypted strings should not be written to text files.

The encryption algorithm encrypts eight (8) characters at a time and any input string is padded with blanks so that the output is a correctly encrypted string. If you truncate this and save an encrypted string that is not a multiple of eight, the last few characters will not decrypt properly.

ERROR

```
num = ERROR( strX )
```

Displays an error box with the specified text and waits for acknowledgment.

EXISTS

```
num = EXISTS( X )
```

Returns 1 if X exists, 0 if X is missing or undefined. To test several numeric variables for existence, use the `CNT` function.

EXTERN

```
num = EXTERN ( X )
```

Invokes a user-supplied external function from the `EXTERN.dll`. The function can take a numeric or string parameter and calls a different user function for each case. The `extern.dll` library supplied by SIR contains dummy functions which return zero.

EXTERNS

```
str = EXTERNS ( X )
```

Invokes a user-supplied external function from the `EXTERN.dll`. The function can take a numeric or string parameter and calls a different user function for each case. The `extern.dll` library supplied by SIR contains dummy functions which return blank (a zero length string).

EXP

```
num = EXP( X )
```

Returns the value of e raised to the X power. e is the constant 2.71828.

FAMNAME

```
str = FAMNAME(n)
```

Returns the nth family name in the default procf. (SYSTEM(57) = Count of families).

FEQ

```
str = FEQ(real1,real2,exponent)
```

Tests two floating point numbers for equality within a limit of accuracy. The function returns 0 if approximately equal, 1 if unequal. For example, if the exponent was -3, the numbers would be equal if within .001.

FILECNT

```
n = FILECNT(str)
```

Counts the files in the directory given by the string argument. The string must be a filename or mask. For example, specify a mask like '*.pql' for the count of those files with extension "pql" in the current directory.

FILEIN

```
str = FILEIN(filter,default_extension)
```

Displays a file browse box for user to choose existing file. Returns zero length string if user cancels.

FILEIS

```
n = FILEIS(file_name_string)
```

Tests if file exists. Returns -1 if name is a directory; 0 if no such name; 1 if file exists.

FILEN

```
str = FILEN(string,n)
```

Returns the nth file name in the directory given by the string argument. The string must be a filename or mask. For example, to list all files with extension "pql" in the current directory:

```
FOR N = 1,FILECNT("*.pql")
. WRITE [FILEN("*.pql",N)
END FOR
```

FILEOUT

```
str = FILEOUT(filter,default_extension)
```

Displays a file browse box for user to choose output file. Returns zero length string if user cancels.

FILESTAT

```
n = FILESTAT(filename_string,type_of_data)
```

Returns various system specific data about a named file.

Type of data

- 1 gid Numeric identifier of group that owns file (UNIX-specific)
- 2 st_atime Time of last access of file. (system date/time integer)
- 3 st_ctime Time of creation of file. (system date/time integer)
- 4 st_dev Drive number of the disk containing the file (same as st_rdev).
- 5 st_ino Number of the information node (the inode) for the file (UNIX-specific).
- 6 st_mode Bit mask for file-mode information. The `_S_IFDIR` bit is set if path specifies a directory; the `_S_IFREG` bit is set if path specifies an ordinary file or a device. User read/write bits are set according to the file's permission mode; user execute bits are set according to the filename extension.
- 7 st_mtime Time of last modification of file. (system date/time integer)
- 8 st_nlink Always 1 on non-NTFS file systems.
- 9 st_rdev Drive number of the disk containing the file (same as st_dev).
- 10 st_size Size of the file in bytes; (Can exceed I4 in size)
- 11 uid Numeric identifier of user who owns file (UNIX-specific)

FILETIME

```
n = FILETIME(filename_string,type_of_data)
```

Returns SIR times or dates about a file

Type of data

- 1 Time of last access of file. (SIR time integer)
- 2 Date of last access of file. (SIR date integer)
- 3 Time of creation of file. (SIR time integer)
- 4 Date of creation of file. (SIR date integer)
- 5 Time of last modification of file. (SIR time integer)
- 6 Date of last modification of file. (SIR date integer)

FILL

```
str = FILL(strX , strY)
```

Replaces all blank characters in strX with the first character of string argument strY. The length of strX does not change. For example:

```
RESULT = FILL (' $100.00', '*')
returns  "***$100.00"
```

FINDITEM

```
num = FINDITEM (id,pos,txt)
```

Search dialog choice or list for text and return position. Can start from partway through.

FORMAT

```
str = FORMAT( X [,W [,D ]] )
```

Converts X to a string. `FORMAT(X)` returns free-field format as wide as necessary to fit the value. `FORMAT(X,W)` returns a free field format of width W. `FORMAT(X,W,D)` returns a number with D decimal places in width W. X, W and D, can be variables, constants or expressions. W, if specified, must be greater than or equal to zero. D, if specified, can be -1 or greater than or equal to zero. -1 is equivalent to not specifying a value (free field format). If D is specified, W must be greater than D. For example:

```
STR = FORMAT (1.3)           returns '1.3'
STR = FORMAT (1.3, 4)       returns ' 1.3'
STR = FORMAT (1.3, 5, 2)    returns ' 1.30'
```

FST

```
num|str = FST( X1, X.., Xn )
```

Returns the first value in the list of up to 128 variables that is not missing or undefined.

FSTR

```
num|str = FSTR( X )
```

Returns the first value of X encountered during a `PROCESS REC` or `PROCESS ROWS` loop that is not missing or undefined.

GETBTNH

```
num = GETBTNH (dummy)
```

Returns the height of button control (for positioning).

GETCHCH

```
num = GETCHCH (dummy)
```

Returns the height of choice control(for positioning).

GETCHKH

```
num = GETCHKH (dummy)
```

Returns the height of check control(for positioning).

GETAKL

```
n = GETAKL(0)
```

Returns automatic disconnection timeout for idle clients (from Master) in minutes.

GETDFC

```
n = GETDFC(0)
```

Returns time (from Master) of difference file copy interval in minutes.

GETENV

```
str = GETENV(variable_str)
```

Returns the value of the named environment variable.

```
COMPUTE OSPATH = GETENV('PATH')
```

GETERR

```
str = GETERR (dummy)
```

Client/server function. Returns the oldest error posted for this application and deletes the message. Returns a zero length string if no messages. Errors are not specific to a connection or statement, rather they are posted for this instance of SIR/XS and, if errors are not retrieved when an error condition occurs, multiple error messages may be waiting. One logical error may also give rise to multiple error messages from the server.

GETFLT

```
dbl = GETFLT (id)
```

Gets floating point as per GETTXT. Returns 0.0 if not f.p. number.

GETFOCUS

```
num = GETFOCUS (0)
```

Returns id of control with focus.

GETICLK

```
num = GETICLK (id)
```

Returns check or radio state.

GETIFLT

```
dbl = GETIFLT (id,pos)
```

Returns floating point from a list.

GETIINT

```
int = GETIINT (id,pos)
```

Returns integer of item from a list.

GETINT

```
int = GETINT (id)
```

Gets integer as per GETTXT. 0 if not integer.

GETITXT

```
str = GETITXT (id,pos)
```

Returns text of item from a list.

GETLBLH

```
num = GETLBLH (dummy)
```

Returns the height of labels (for positioning).

GETLTXT

```
str = GETLTXT (id,p)
```

Gets the text from a line in a multi-line text control.

GETMAXCH

```
num = GETMAXCH (dummy)
```

Returns the height of the maximum single line control (for positioning).

GETMCADD

```
str = GETMCADD (id)
```

Returns tcp/ip address for client from master.

GETMCHK

```
num = GETMCHK (id)
```

Returns check state from menu item.

GETMCLID

```
n = GETMCLID (n)
```

Returns id for nth client from master.

GETMCLST

```
n = GETMCLST (n)
```

Returns time of last message for nth client from master.

GETMCON

```
n = GETMCON (n)
```

Returns time of log on for nth client from master.

GETMDBN

```
str = GETMDBN (n)
```

Returns name of nth database from master.

GETMSEL

```
num = GETMSEL (id,N)
```

Returns the position of the nth selected item from a single or multiple selection list. In the case of a single selection list then `GETMSEL(id,1)` is the same as `GETPOS(id)`.

GETNITEM

```
num = GETNITEM (id)
```

Returns number of items in choice or list.

GETNLINE

```
num = GETNLINE (id)
```

Gets number of lines in multi-line text.

GETNSEL

```
num = GETNSEL (id)
```

Returns number of items selected in multiple selection or returns 1 for a single selection list.

GETPOS

```
num = GETPOS (id)
```

Returns pos of current selection in list or choice or keyboard focus in multiple selection.

GETRADH

```
num = GETRADH (dummy)
```

Returns the height of radio control (for positioning).

GETRSTEP

```
num = GETRSTEP (dummy)
```

Returns the size of the row step (for positioning).

GETTXT

```
str = GETTXT (id)
```

Gets text from edit and from highlighted item in choice or list. (Gets label text from label, button, check and radio).

GETTXTH

```
num = GETTXTH (dummy)
```

Returns the height of text control (for positioning).

GLOBALN

```
num = GLOBALN(globvar_exp , numeric_exp)
```

Assigns a numeric value to a global variable. The first argument is the name of the global variable, the second argument is the numeric expression (or variable name). For example: To assign the global variable RT the value 25.5:

```
COMPUTE Y = GLOBALN('RT' , 25.5)
```

GLOBALS

```
num = GLOBALS(stringexp , stringexp)
```

Assigns a string value to a global variable. The first argument is the name of the global variable, the second argument is the string expression (or variable name). For example: To assign the global variable TEMP the value JOE SMITH:

```
COMPUTE Y = GLOBALS ('TEMP', 'JOE SMITH')  
GLOBALN and GLOBALS return:
```

0 if the assignment was made.

-1 if the first argument is not a valid global variable name.

-2 if the second argument is missing.

Do not try to use the value of globals set by GLOBALN or GLOBALS for text substitution *in the same program* (by using the global variable name within angle brackets) because the functions work at execution time and text substitution happens at compile time.

GLOBNAME

```
str = GLOBNAME(n)
```

Returns the name of the nth global variable. (SYSTEM(53) = Global Count).

HELP

```
error = HELP( help page )
```

Invokes the HELP system, beginning with the specified help page. The page must be in html format in the help directory. The page name can contain directory names using forward slashes to delimit. Enclose the name in quotes. The system converts this to a file name prefixed with a path pointing to the help directory and suffixed with the file extension .htm. For example:

```
COMPUTE X = HELP ('visualpql/function/foreword')
```

ICHAR

```
num = ICHAR(C)
```

Returns a numeric value equivalent to the position in the character collating sequence of the first character in string C. The character collating sequence for a given computer is the set of numeric codes used for internal character representation (ASCII).

IDSTATUS

```
num = IDSTATUS(id)
```

Returns the status of a gui element. -1 = the control does not exist; 0 = the control is disabled; 1 = the control is enabled.

JOUFLAG

```
num = JOUFLAG (dummy)
```

Returns whether journaling is on (1) or off (0) for the database.

JULC

```
str = JULC( X )
```

Converts a "date integer", X, into an 12-character string of the form 'MMM DD, YYYY'.

JULN

```
num = JULN(month,day,year)
```

Returns a "date integer" where the three numeric arguments are month, day and year. A "date integer" is the number of days since the start of the Gregorian calendar on October 15, 1582.

Years can be specified as various values. If a year of zero is specified, this year is used. If a year of less than 10 is specified, this decade is used. If a year between 10 and 99 is specified, the `CENTYR` parameter is used to determine the appropriate century to use. If a year between 100 and 999 is specified, then values greater than 583 are taken to be in the last millennium (1583 - 1999), values smaller than 583 are taken to be in the current millennium (2000+).

A value of undefined is returned if a date earlier than October 15, 1582 is specified.

Because of leap years, this routine is only accurate for dates up to Dec 31 29999.

For example:

```
DURATION = JULN(4, 8, 0) - BEGINDAT
```

```
ENDPROJ = JUNL(TMON, TDAY, TYEAR)
```

KEYNAME

```
str = KEYNAME (rtnum, keynum)
```

Returns the name of the specified keyfield for the specified record type. `RNUM` is the record number. `KEYNUM` is the number of the keyfield, i.e. 1 is the case id, 2 is the first key field in the record type, etc.

KEYORDER

```
str = KEYORDER (rtnum, varname_string)
```

Returns "A" or "D" for the sort order of the specified keyfield. The variable name argument is an expression.

LEN

```
num = LEN( strX )
```

Returns an integer value that is the length, in characters, of the string strX, leading and trailing blanks included.

LG10

```
num = L10( X )
```

See ALOG10 function.

LINES

```
num = LINES(filename)
```

Returns number of lines remaining on current page being written to a file.

LN

```
num = LN( X )
```

See ALOG function.

LOG

```
num = LOG( X )
```

See ALOG function.

LOG10

```
num = LOG10( X )
```

See ALOG10 function. (LG10 is also allowed.)

LOWER

```
str = LOWER( string )
```

Returns the string with all characters converted to lower case.

LST

```
num|str = LST(X1 , X ,....., Xn)
```

Returns the last value in the list of up to 128 variables that is not missing or undefined.

LSTR

```
num|str = LSTR( X )
```

Returns the last value of X encountered in a PROCESS REC or PROCESS ROWS loop that is not missing or undefined.

MAKEDIR

```
num = MAKEDIR( name )
```

Creates a new directory using the name. Returns -1 if the directory cannot be created.

MAX

```
num|str = MAX(X1 , X ,....., Xn)
```

Returns the maximum value in the list of up to 128 variables.

MAXR

```
num|str = MAXR( varname )
```

Returns the maximum value of the specified variable encountered during a PROCESS REC or PROCESS ROWS loop that is not missing or undefined.

MAXRECS

```
num = MAXRECS( rectype )
```

Returns the maximum number of records allowed for this record type.

MEAN

```
num = MEAN(X1 , X ,....., Xn)
```

Returns the mean (arithmetic average), of the values within the list that are not missing or undefined. The maximum number of variables allowed in the argument list is 128.

MEANR

```
num = MEANR( varname )
```

Returns the mean (arithmetic average), for the values of the specified variable encountered during a `PROCESS REC` or `PROCESS ROWS` loop that are not missing or undefined.

MEMCOUNT

```
num = MEMCOUNT( famname )
```

Returns the count of members in the named family of the default profile.

MEMINFO

```
num = MEMINFO( member_name, info_type )
```

Returns information about the named member. The member name can include the `:type` qualifier. The function returns missing if the member does not exist.

The INFOTYPES are:

1. *TYPE* - returns 1 for :T; 2 for :E; 4 for :O ; 8 for :V
If the member type is not given in the member name, and there is more than one type for this name then the sum of the types is returned. e.g.
SYSPROC.MENU.ABOUT is a :t and an :o so
MEMINFO ("SYSPROC.MENU.ABOUT" , 1) returns 5
MEMINFO ("SYSPROC.MENU.ABOUT:O" , 1) returns 4
MEMINFO ("SYSPROC.MENU.ABOUT:E" , 1) returns missing.
2. *SECURITY* returns 0 (no password), 1 (password & public), 2 (password & not public).
3. *LENGTH* returns the number of bytes in the member (note there is some condensing going on here).
4. *CREATE DATE* returns the creation date as a Julian date integer.
5. *CREATE TIME* returns the creation time as number of seconds since midnight.
6. *MOD DATE* returns the last modification date as a Julian date integer.
7. *MOD TIME* returns the last modification time as number of seconds since midnight.
8. *STATUS* (mainly for :e :o and :v types):
0 - good
-1 incorrect version of SIR
-2 Default database not connected
-3 Database creation date/time mismatch
-4 incorrect PQL Retrieval version
-5 CIR/Record Schema level mismatch.

MEMNAME

```
str = MEMNAME( famname, n )
```

Returns the name of the nth members in the named family of the default profile.

MIN

```
num|str = MIN(X1 , X , . . . , Xn)
```

Returns the minimum value in the list of up to 128 variables.

MINR

```
num|str = MINR( varname )
```

Returns the minimum value of the specified variable encountered during a `PROCESS REC` or `PROCESS ROWS` loop that is not missing or undefined.

MISNUM

```
num = MISNUM( X )
```

If X is undefined, 0 is returned. If X is missing, the missing type is returned (1 to 3). If X is neither missing nor undefined, undefined is returned. X may be a numeric or string variable.

MISS

```
str = MISS (rtnum, varname_str , n)
```

Returns the original value for undefined and first, second and third missing values (as a string), where n is 0, 1, 2, or 3. If the record number (rtnum) is negative, the function applies to a summary variable; if rtnum is one more than the maximum record count (i.e. NRECS(0)+1) then this applies to a standard variable.

MISSING

```
num|str = MISSING(varname)
```

Returns the original value of a variable if it is missing, otherwise undefined is returned. If the variable is a time, date, or categorical integer, the original string value is returned. If the variable is an integer or floating point variable, the original numeric value is returned. If the argument is not a single variable name, undefined is returned.

MKEYSIZE

```
num = MKEYSIZE ( dummy )
```

Returns the maximum key size in bytes for the database. The key for a record is comprised of the case identifier (in a case structured database), the record type number,

and the keyfields defined for the record type (if any). Refer to the schema definition command `MAX KEY SIZE` for more information about maximum key size.

MOD

See `AMOD` function.

MRECSIZE

```
num = MRECSIZE (dummy)
```

Returns the size of the largest record type in the database. Size is expressed in the number of double words that are equivalent to eight characters.

MSGTXT

```
str = MSGTXT (num)
```

Returns the text of the error or warning message given by the number `num`.

NARG

```
num = NARG( num )
```

Returns numeric arguments from the command parameter list. (String parameters are retrieved with the `SARG` function.)

The argument is the position of the parameter in the list. An argument value of zero returns the number of parameters in the list. If the argument is greater than the number of parameters in the parameter list or the argument is a string, undefined is returned. For example, to return the value of the third argument of the parameter list (which must be numeric):

```
COMPUTE ARG4 = NARG(3)
```

NBRANCH

```
num = NBRANCH (control_id,node)
```

Returns number of branches of the given node in a tree control.

NEXTROW

```
num = NEXTROW (conid,statid)
```

Client/server function. Steps through the rows one at a time. This must be issued before getting data for the first row. Returns the row number or zero if no more rows.

NGET

```
num = NGET( varname_str )
```

Returns the value of the specified numeric variable. The variable name is specified as a string variable, quoted string constant or string expression whose value is the name of a common, record or program variable.

NGLOBAL

```
num = NGLOBAL( C )
```

Returns the value of numeric global parameters. C is a character variable, constant or expression whose value specifies the name of the global parameter. If C is not a defined global parameter, undefined is returned. String global parameters are retrieved with SGLOBAL. For example, to set the variable NVAL to the value of global parameter RACETIME:

```
COMPUTE NVAL = NGLOBAL ( 'RACETIME' )
```

NKEYS

```
num = NKEYS ( rtnum )
```

Returns the number of keyfields (sort-ids) for the specified record type (excluding the case id).

NLABELS

```
num = NLABELS ( rtnum , varname_str )
```

See NVALLAB.

NMAX

```
num = NMAX ( rtnum , varname_str )
```

Returns the highest valid numeric value for the specified variable. If the record number (rtnum) is negative, the function applies to a summary variable; if rtnum is one more than the maximum record count (i.e. NRECS(0)+1) then this applies to a standard variable. See VAR RANGES for more information about valid ranges.

NMIN

```
num = NMIN ( rtnum, varname_str )
```

Returns the lowest valid numeric value for the specified variable. If the record number (rtnum) is negative, the function applies to a summary variable; if rtnum is one more than

the maximum record count (i.e. NRECS(0)+1) then this applies to a standard variable. See `VAR RANGES` for more information about valid ranges.

NOFCASES

```
num = NOFCASES ( dummy )
```

Returns the current maximum number of cases as defined in the schema See `N of CASES` for more information.

NOW

```
num = NOW( dummy )
```

Returns a "time integer" representing the current time of day as the number of seconds since midnight. The argument is a dummy argument, specify zero.

NPUT

```
num = NPUT ( A , Y )
```

Stores the value of numeric argument Y in numeric variable A. A is a string variable name, quoted string constant or string expression whose value is the name of a common, record, or program variable. The value returned by the function is the value actually stored in A (possibly undefined, missing, etc.). If A refers to a common or record variable, the Retrieval must be in update mode. The following example stores 175 in the variable Height:

```
COMPUTE DUMMY = NPUT( 'HEIGHT' , 175 )
```

NREAD

```
num = NREAD( strX )
```

Pops up a box on the screen with a prompt and returns a number from the user.

If a non-numeric field is entered, a message is issued and the user is prompted again.

NRECS

```
num = NRECS ( dummy )
```

Returns the maximum number of record types for the database. This is the maximum possible number of record types, not the actual number of record types defined for the database.

NSUBDIR

```
str = NSUBDIR (n)
```

Returns the name of the nth sub-directory.

NUMBR

```
num = NUMBR( strX )
```

Returns the numeric value of the string strX. strX is a string constant, variable name or expression and contains only numerical characters, at most one decimal point and a plus or minus sign or is in E+exponent format.

NUMCASES

```
num = NUMCASES ( dummy )
```

Returns the number of cases in the database. Same as functions SYSTEM(24) or NUMRECS(0).

NUMRECS

```
num = NUMRECS ( rtnum )
```

Returns the number of records of the specified record type.

NVALID

```
num = NVALID ( rtnum , varname_str )
```

Returns the number of valid values for the specified variable. If the record number (rtnum) is negative, the function applies to a summary variable; if rtnum is one more than the maximum record count (i.e. NRECS(0)+1) then this applies to a standard variable.

NVALLAB

```
num = NVALLAB ( rtnum , varname_str )
```

Returns the number of value labels defined for a variable. NLABELS is a synonym. If the record number (rtnum) is negative, the function applies to a summary variable; if rtnum is one more than the maximum record count (i.e. NRECS(0)+1) then this applies to a standard variable.

See VALUE LABELS for more information about value labels.

NVARDOC

```
num = NVARDOC ( rtnum , varname_str )
```

Returns the number of lines of documentation defined for a variable. Variable documentation does not apply to a summary variables. (Just use comment lines in programs for documentation.) See `VAR DOC` for more information about variable documentation.

NVARS

```
num = NVARS ( rtnum )
```

Returns the number of variables (not Common) defined in the specified record type.

NVARSC

```
num = NVARSC ( rtnum )
```

Returns the number of variables (including Common) defined in the specified record type.

NVVAL

```
num = NVVAL ( rtnum , varname_str , n )
```

Returns the value of the **n**th valid value of a numeric variable. If the record number (rtnum) is negative, the function applies to a summary variable; if rtnum is one more than the maximum record count (i.e. `NRECS(0)+1`) then this applies to a standard variable.

ODBCCOLS

```
num = ODBCCOLS ( conid,statid,tabname )
```

Client/server function. Does an ODBC query that produces a result set that contains a list of columns from the named table on the data source and can be interrogated using the standard functions.

ODBCTABS

```
num = ODBCTABS ( conid,statid )
```

Client/server function. Does an ODBC query that produces a result set that contains a list of tables on the data source and can be interrogated using the standard functions.

OUTFNAME

```
str = OUTFNAME ( 0 )
```

Returns the name of the default output file.

PACK

```
str = PACK ( strX )
```

Returns a string with leading and trailing blanks deleted and multiple blanks compressed into one blank. The argument strX may be a string constant, variable name or string expression.

PAD

```
str = PAD( input, pad , len , trunc)
```

Pads the input string with the specified pad character to the pad length and then truncates the string to the truncation length.

PAGELEN

```
n = PAGELEN(filename)
```

Returns the current setting for page length for specified file.

PAGENO

```
n = PAGENO(filename)
```

Returns the current page number on specified file.

PAGEWID

```
n = PAGEWID(filename)
```

Returns the current setting for page width on specified file.

PATTERN

```
num = PATTERN (strX , pattern_str)
```

Returns 1 if the pattern specified by pattern_str is in strX, otherwise 0. Both arguments are strings and can be variables, constants or expressions. The pattern can contain the match anything character "@". Undefined is returned if either argument is missing or undefined. For example, the following returns 1.

```
THERE = PATTERN ('Mr. Ralph Jones', 'Mr.@Jones@')
```

PFORMAT

```
str = PFORMAT(num, picture)
```

Formats a number according to a picture, returning a string. A picture is a string of characters, enclosed in quotes. Within the picture certain characters have special meanings. The meanings are identical to those used on the output specification of the WRITE command as per the following:

Each digit can be represented by a "9", a "z", a "*" or a "\$". "9" specifies that leading zeros are replaced by blank; "z" specifies that leading zeros are written; "*" specifies that leading zeros are replaced by "*"; "\$" after an initial "\$" character, represents a floating dollar sign where leading zeros are suppressed. If the field has a value of zero, a picture of all "9"s results in blanks and all "\$"s results in a single "\$" since all leading zeros are suppressed; if a single zero is wanted, specify a single "z" as the last character of the picture.

A period represents the decimal point and separates the specification into characters before and after the decimal point. There can only be one decimal point (period) in the picture. If there are insufficient digits to display the integer portion of the field (including any minus sign when negative and \$ when specified), the field is written as all 'X's. The decimal component is rounded to match the number of decimal digits specified. If there are no decimal digits in the picture, the field is rounded to the integer value.

Specify comma (,) to insert this character. If leading zeros are suppressed (by blanks or a floating dollar), any leading commas are suppressed. If a single dollar sign is specified, it is output in that position. If multiple dollar signs are specified, these suppress leading zeros and result in a floating dollar sign that is output in front of the first significant digit. After the decimal point, the special characters "9", "Z", "\$" and "*" are all equivalent and specify a digit. Any other characters are treated as any other special character.

Negative numbers, by default, are output with a minus sign ahead of the first significant character. If an explicit minus sign is included as the last character in the picture, and the number is negative, the minus is written at that point. Any other characters are output at the position specified in the picture. For example:

```
PROGRAM
write [ ""+PFORMAT(2500,'$zzzz.zz')+"" ] 40t
"$2500.00"
write [ ""+PFORMAT(12345.67,'zzzzzzzz.zzzzz')+"" ] 40t
"00012345.67000"
write [ ""+PFORMAT(SQRT(99),'999999.999999')+"" ] 40t      ""
9.9498744"
write [ ""+PFORMAT(12345.67,'z z z z z.ZZZZ')+"" ] 40t      "" 1 2
3 4 5.6700"
write [ ""+PFORMAT(12345.67,'ZZZZZ')+"" ] 40t
"12346"
write [ ""+PFORMAT(12345.67,'*****.**')+"" ] 40t
"*****12345.67"
write [ ""+PFORMAT(12345.67,'9,999,999.99')+"" ] 40t      ""
12,345.67"
write [ ""+PFORMAT(12345.67,'9,9,9,9,9,9,9,9,9,9,9,9')+"" ] 40t      ""
1,2,3,4,5.6,7,0"
write [ ""+PFORMAT(-9,'ZZZ')+"" ] 40t      "" -
09"
END PROGRAM
```

PICTURE

```
num = PICTURE(str, picture)
```

Validates a string according to a specified picture. A picture is a string of characters, enclosed in quotes. string. Within the picture certain characters have special meanings: *Note:* the character codes are lower case. They are:

```
a - any letter
d - any digit
n - any letter or digit
s - numeric value components (0-9, decimal point,+,+"," ,E)
u - any uppercase letter
l - any lowercase letter
x - any character
```

The first example returns a 0 showing the string matches the picture. The second example returns a 6 to show that the string does not match the picture in the sixth position:

```
X = PICTURE ('123-45-6789', 'ddd-dd-dddd')
X = PICTURE ('123-45-67', 'ddd-d-ddd')
```

PROCFILE

```
str = PROCFILE(0)
```

Returns the filename of the default procfile (e.g.: c:\SIRXS\company.sr4)

PROCNAME

```
str = PROCNAME(0)
```

Returns the attribute of the default procfile. In SIR/XS this is always PROCFILE.

PROGRESS

```
num = PROGRESS ( type,percent )
```

Controls the display of a progress bar.

x = PROGRESS (0,0) initiates the display.

x = PROGRESS (1,n) displays progress up to n where n is a percentage from 1 to 100.

x = PROGRESS (2,0) closes the display.

The initiation, updating and closing do not have to be in the same VisualPQL program.

Once initiated in a program, the progress display is closed only by this function, not automatically at the end of the program and thus can be used to display progress through a suite of programs. If the progress display has not been initiated, the function has no effect.

RACCESS

```
num = RACCESS ( dummy )
```

Returns the read security access level of the current user. That is the level corresponding to the read security password of the user.

RAND

```
num = RAND( dummy )
```

Returns a uniform random number between 0 and 1. The normal way to call the function is with a dummy argument of zero. Multiple calls then return a sequence of random numbers. If the function is called with a number as the argument, this resets the seed and returns the first random number generated from that seed. For a given seed, the same sequence of random numbers is generated. You can also alter the default seed by specifying a seed on the `RETRIEVAL` or `PROGRAM` command.

RANF

See `RAND` function.

REAL4

```
num = REAL4(real*8)
```

Converts a `real*8` into a `real*4` number.

RECDOC

```
num = RECDOC(recno,lineno)
```

Returns the *n*th line of documentation for the record. If the line number is zero, the function returns the record label. If the record number is zero, the function returns the *n*th line of database level documentation and if both record and line number are zero then the `DATABASE LABEL` is returned.

RECDOCN

```
num = RECDOCN(recno)
```

Returns the number of lines of documentation for the record. If the record number is zero, the function returns the number of lines of database level documentation.

RECLEVEL

```
num = RECLEVEL( dummy )
```

Returns the update level of the current record (when it was last written to the database). The update level changes with each modification to the record (during a `RETRIEVAL`

UPDATE, Batch Data Input run, FORMS updating session, etc.). This function can only be used in a `PROCESS REC` loop.

RELOCK

```
num = RELOCK(locktype)
```

Changes the lock type for the current record for concurrent operations and attempts to read the current record from the database. See `SYSTEM(36)` function to determine if current record is locked. The `locktype` codes are (all other values set concurrent read):

```
1 & 6 = Exclusive  
2 = Concurrent Read  
3 = Concurrent Write  
4 = Protected Read  
5 = Protected Write
```

RECNAME

```
str= RECNAME( rtnum )
```

Returns the name of the specified record type. If `rtnum` is 0, "CIR" is returned. Only used in a `RETRIEVAL`.

RECNUM

```
num= RECNUM( recname )
```

Returns the number of the specified record name. If the name does not exist, returns undefined. Only used in a `RETRIEVAL`.

RECSIZE

```
num = RECSIZE ( rtnum )
```

Returns the record size for the specified record type in double words. For example, to find the length of record type 1:

```
EMPSIZE = RECSIZE (1)
```

REGEXP

```
num = REGEXP(string,regular_expression,nth,style)
```

Searches a string for the `nth` occurrence of a substring as specified by the regular expression. Returns -1 if error in regular expression, 0 if not found in string or `n` where `n` is start position of `nth` occurrence of found string.

A regular expression is one where special characters describe the matching that is required. The meaning of the special characters needs to be specified and there is a standard for regular expressions used by many packages. SIR has had its own regular expression processor and these functions `REGEXP/REGREP` allow you to choose whether to use standard PERL or POSIX expressions or SIR expressions. A `style` of 1 specifies SIR expression, 2 specifies PERL and 3 specifies POSIX. PERL is the default. Please see standard documentation for PERL and POSIX for a full explanation of the syntax of their regular expressions.

SIR Expressions

A SIR expression consists of the following:

- c literal character (eg: "Name: ")
- ? any character except end of line (eg: "?and");
- % beginning of line (eg: "%first");
- \$ end of line (null string before end of line) (eg: "last.\$");
- [...] character class (any one of 'these' characters)(eg: [a-zA-Z0-9#@%_]);
- [!...] negated character class (all but these characters) (eg: [!a-z]);
- * closure (zero or more occurrences of the previous pat)(eg: [a-z]*);
- + closure (one or more occurrences of the previous pat)(eg: [a-z]+);
- @c escaped character (eg: @%, @[, @*);

Any special meaning of characters in an expression is lost when escaped, inside [...] or in the following cases:

- % not at the beginning (eg: [0-9.]+%);
- \$ not at the end (eg: \$[0-9.]+);
- * at the beginning of a pattern;
- + at the beginning;

A character class consists of zero or more of the following elements, surrounded by [and]:

- c literal character, including [
- a-c range of characters (digits, lower or upper case)
- ! negated character class (if at beginning)
- @c escaped character (@!, @-, @@, @])

Special meaning of characters in a character class is lost when escaped or for:

- ! not at the beginning
- at the beginning or end

Any part of expression may be specified to be 'tagged':

- < start a tagged substring
- > end a tagged substring

(Tagged substrings are numbered from left to right. See the substitution expression for replacement of tagged substrings.)

A substitution pattern consists of zero or more of the following elements:

- c literal character

& ditto, i.e. whatever was matched

@c escaped character (@&)

@n tagged substring insertion

An escape sequence consists of the character @ followed by a single character:

@n end of line

@t tab character

@c any other character (including @@)

Examples:

1) Mark numbers with []s:

```
PROGRAM
c
c Put numbers in square brackets
c
. STRING*80 TEXT
. SET TEXT ("this is 123 or 456 test")
. COMPUTE TEXT = REGrep(TEXT, "[0-9]+", "[&]", 1, 1)
. WRITE TEXT
END PROGRAM
```

2) Swap Last name and first name + possible initial:

```
RETRIEVAL
. PROCESS CASES
.   PROCESS RECORD 1
.     GET VARS NAME
.     WRITE [REGREP(NAME, " ", "@3 " + CHAR(9) + "@1 @2 ", 1, 1)]
.   END RECORD
. END CASE
END RETRIEVAL
```

3) Find each word in a sentence. In POSIX, the expression `\w*(\W|$)` matches any number of letters (`\w*`) followed by a non-letter (`\W`) or (`|`) end of line (`$`). The expression `\w(\W|$)` will find a single letter followed by a non-letter and so point to the end of a word.

```
PROGRAM
c
c Find words in a string
c
STRING*80 TEXT
SET TEXT ("Find the words in a string")
SET N (1)
LOOP
. COMPUTE POS1 = REGEXP(TEXT, "\w*(\W|$)", N, 3)
```

```

. COMPUTE POS2 = REGEXP(TEXT, "\w(\W|$)", N, 3)
. IF (POS1 EQ 0) EXIT LOOP
. WRITE "WORD# " N [SBST(TEXT, POS1, 1+POS2-POS1)]
. COMPUTE N = N + 1
END LOOP
END PROGRAM

```

4) Verify entry of an email address. From the start of the line (^) to the @ must contain one or more (+) alphanumeric characters or `._%-` (`[A-Za-z0-9._%-]`) then one or more alphanumeric or `.-` (`[A-Za-z0-9.-]`), a literal `.\.` followed by two to four (`{2,4}`) alphabetic characters (`[A-Za-z]`).

```

PROGRAM
c
c Check an email address
c
STRING*80 TEXT
SET TEXT ( " " )
LOOP
. DISPLAY TEXTBOX "Enter a valid email address" response RC,TEXT
. IFNOT (RC GT 0) EXIT LOOP
. IFTHEN (REGEXP(TEXT, "^([A-Za-z0-9._%-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4})$" ,1,3) EQ 0)
.   DISPLAY ERRORBOX " is an invalid email address"
.   NEXT LOOP
. ELSE
.   DISPLAY INFOBOX "Thank you"
.   EXIT LOOP
. ENDIF
END LOOP
END PROGRAM

```

REGREP

```

str =
REGREP(string, match_regular_expression, replace_regular_expression, nth, s
tyle)

```

Searches a string for the `nth` occurrence of a substring as specified by the regular expression and replaces that substring as specified by the replace regular expression. The function returns the updated string. If any errors are found, the unmodified string is returned. Please see the previous `REGEXP` function for details on SIR regular expressions.

REPLACE

```

str = REPLACE(original, search, replace, times, offset, anchor)

```

If the search string is found in the original string, occurrences are replaced by the replace string. The number of times the string is replaced, the offset for the next starting position, and an anchor column are also specified. The following example returns "CABCBCBBAA" if `INLINE` is "AABABABBAA".

```
INLINE = REPLACE ( INLINE, 'A', 'C', 3, 2, 0)
```

When anything but 0 is specified as the anchor, the string is only replaced once, at the anchor position. The following example returns "AABCBBABBAA" if `INLINE` is "AABABABBAA".

```
INLINE = REPLACE ( INLINE, 'A', 'C', 3, 2, 4)
```

REVERSE

```
str = REVERSE (str)
```

Reverses a string.

RKEYSIZE

```
num = RKEYSIZE (rtnum)
```

Returns the key length a record type. This is the sum of the key fields of the record type plus the case identifier in case structured databases.

RND

```
num = RND( X [,n ])
```

Returns X rounded to n decimal places. Express n as powers of 10, negative for numbers smaller than 1. Omit n or specify 0 to round to integers. Rounding is done by adding $0.5 \cdot 10^n$ to positive numbers, subtracting $0.5 \cdot 10^n$ from negative then truncating.

RNMFILE

```
num = num = RNMFILE (oldname_str, newname_str)
```

Renames a file from `oldname_str` to `newname_str`. Returns 0 for a success and -1 for fail.

ROWCOUNT

```
num = ROWCOUNT (conid, statid)
```

Client/Server function. Returns a count of rows created by the execute command. When using ODBC, this depends on the ODBC source and may not be available (returns -1).

RRECSEC

```
num = RRECSEC (rtnum)
```

Returns the read security level for a record type.

RVARSEC

```
num = RVARSEC (rtnum, varname_str)
```

Returns the read security level for a variable.

SARG

```
str = SARG( num )
```

Returns string arguments from the parameter list. The argument is the position in the list. If it is greater than the number of parameters in the list, undefined is returned. (Numeric parameters are retrieved with the `NARG` function. `NARG(0)` returns the number of parameters in the list.)

SBST

```
str = SBST( input_str , start_pos , num_chars )
```

Returns a substring of the input string. The second argument, `start_pos`, specifies the position within the input string where the substring begins. The third argument specifies the number of characters to retrieve from the input string. If any of the arguments are undefined or missing, undefined is returned. If the starting position is larger than the length of the input string, undefined is returned.

SCROLLAT

```
num = SCROLLAT(id)
```

Returns current line number of scrollable gui control.

SCROLLTO

```
num = SCROLLTO(id,line)
```

Sets current line number of scrollable gui control.

SEEK

```
num = SEEK(attribute_exp,position)
```

Sets position of currently open file where `position` is the number of characters to move from the current location in the file. If `position` is -1 it moves to EOF; if `position` is -2 then it doesn't move and just returns current position; if `position` is -3 it moves to the start of the file. To move to an absolute position, first move to the start of the file and then to the position.

The return value is the new pointer position. The attribute expression is the attribute associated with this file. To specify this directly (as opposed to specifying a variable name which contains the attribute), enclose the name in quotes.

SERADMIN

```
n = SERADMIN (function_type,server_client_id,password)
```

Client function for PQLServer. Server administration function.

Function types:

- 1 - number of server clients
- 2 - server client id of nth client
- 3 - close server client id (not us) (password)
- 4 - server client id logon time
- 5 - server client id last message time
- 6 - shutdown server
- 7 - shutdown server when no clients (password)

The server client id is required on function types 3,4 and 5. Server client ids are returned by function type 2. Use the nth client number as the client id on function type 2 (0 returns our server client id). Use 0 on function types 1,6 and 7.

Do not use SERADMIN function type 3 to close this client.

SERADMIS

```
str = SERADMIS (function_type,client_id,password)
```

Client function for PQLServer. Server admin functions that return a string value.

Function types:

- 1 - address/name of client id (0 returns our name)

SEREXEC

```
n = SEREXEC (wait_factor)
```

Send to server to execute previous sent commands.

0 Wait means return without waiting;

1 means wait one interval, 2 two intervals, etc. The basic server timeout interval can be set on the `sir.ini` file- the setting is called `server.timeout`. If not set, the default is 5 seconds.

Returns completion code:

-1 - Timeout. Note that the server continues to run the request - it is just taking longer than expected. The `SERTEST` function can be used subsequently to inquire as to status.

0 Completed OK

1 Completed with warnings

2 Completed with errors

Other values can set by user program on server using the `SETRC (RC)` function (note user return codes are returned as positive numbers so should avoid values of 1 or 2).

SERGET

```
str = SERGET (number_of_lines_to_skip)
```

Returns a single line of output. If the number of lines to skip is 0, the next line is returned, otherwise it is the line after the skipped lines.

SERLINES

```
n = (dummy)
```

Returns the number of lines left. If this function is invoked after lines are returned or skipped, it returns the remaining number of lines.

SERLOG

```
n = SERLOG (Server_name, Password)
```

Logs on/off to the server. This first logs off any current connection then, if the name of the server to logon to is not blank, this is used to try to log on to. Returns -1 if the logon fails. A string containing the password is required. If the server is started with UPASS specified then the password must match the server administration password. Note that while specified as a string or string variable, the password is a SIR/XS name and is uppercase if not a non-standard name in curly brackets {}.

SERNOOUT

```
num = SERNOOUT (n)
```

PQLServer side function that controls SERVER NOOUTPUT flag. If flag is set on, then any output directed to standard output is thrown away. The flag is set off initially for each client execution. The setting is still maintained once the program that uses this function ends.

n = 0 returns setting 1 - On 0 Off

n = 1 sets no output on

n = -1 sets no output off

SERSEND

```
n = SERSEND (string)
```

Sends string to the server. The string is a line of input.

SERSENDB

```
n = SERSENDB (buffer_name)
```

Sends contents of buffer to the server.

SERTEST

```
n = SERTEST (wait_factor)
```

Use if haven't waited for SEREXEC to complete or had a timeout on the execution. Returns same completion codes as SEREXEC

SERWRITE

```
num = SERWRITE (string)
```

PQLServer side function that writes lines to output regardless of setting of SERVER NOOUTPUT flag

SETAKL

```
n = SETAKL(time,password)
```

Sets the client autokill limit in minutes. If a client is idle for the given number of minutes then they will be automatically disconnected by master. If master has been started with a password, this must match the quoted password, otherwise any name can be used.

SETDFC

```
n = SETDFC(time,password)
```

Sets the master difference file copy interval in minutes. If master has been started with a password, this must match the quoted password, otherwise any name can be used.

SETDIR

```
n = SETDIR(directory_name)
```

Sets the default directory.

SETPOS

```
n = SETPOS(id,pos)
```

Sets the position of a multi-line gui control.

SETRANGE

```
n = SETRANGE(id,min,max)
```

Sets the range of a gui spin/slider/progress control.

Sets the maximum number of characters allowed in an edit or text field (ignoring the minimum parameter).

SETRC

```
n = SETRC( numeric_return_code )
```

Sets the return code SIR/XS sends to the operating system when it finishes.

SGET

```
str = SGET( varname_str )
```

Returns the value of the specified string variable. The argument is a string variable, quoted string constant or string expression whose value is the name of a common, record or program variable.

SGLOBAL

```
str = SGLOBAL( varname_str )
```

Returns the string value of a global variable. The argument is a string variable, constant or expression that specifies a global variable name. If it is not the name of a defined global variable, undefined is returned. Use the `NGLOBAL` function for numeric global variables.

SIGN

```
num = SIGN( num_X , num_Y )
```

Transfers the sign (positive or negative) of `num_Y` to the absolute value of `num_X`. Zero is positive.

SIN

```
num = SIN( radnum )
```

Returns the trigonometric sine of `radnum`, where `radnum` is specified in radians.

SIRUSER

```
name = SIRUSER( name )
```

Sets/returns the current SIR/XS user name. This is written to the database journal to identify the person who entered particular sets of database updates. Specify a name (up to 32 characters). No translation (upper/lower case, etc.) is done. To simply return the current user name, pass either a missing name or a zero length name.

`USER = SIRUSER ('')` returns existing name to string variable `USER`
`USER = SIRUSER ('Mr. J. Smith')` sets the name and returns it to string variable `USER`.

SMAX

```
str = SMAX ( rtnum , varname_str )
```

Returns the maximum valid string value for the specified variable. If the record number (`rtnum`) is negative, the function applies to a summary variable; if `rtnum` is one more than the maximum record count (i.e. `NRECS(0)+1`) then this applies to a standard variable.

SMIN

```
str = SMIN ( rtnum , varname_str )
```

Returns the minimum valid string value for the specified variable. If the record number (`rtnum`) is negative, the function applies to a summary variable; if `rtnum` is one more than the maximum record count (i.e. `NRECS(0)+1`) then this applies to a standard variable.

SPREAD

```
str = SPREAD ( input_str )
```

Returns a string with a blank inserted between each character of the input string.

SPUT

```
str = SPUT (varname_str, str )
```

Stores the value of string argument `str` in the specified string variable. The variable name argument is a string variable name, quoted string constant or string expression whose value is the name of a common, record or program variable. The value returned by the function is the value stored in the variable (possibly undefined, missing, etc.). If the variable refers to a common or record variable, the Retrieval must be in `UPDATE` mode.

SQRT

```
num = SQRT( X )
```

Returns the square root of `X`. Missing is returned for negative values.

SRCH

```
num = SRCH( varX , varY , Z )
```

Returns the location of the value `Z` in the table of values `VarX` to `VarY`, where `VarX` and `VarY` are local numeric variables defined in the program from `VarX` to `VarY`. These

cannot be arrays or string variables. The values in the variables must be in ascending order.

For example, if Z matched the fourth value in the table, SRCH returns 4. If no match is found, a negative value is returned. The value indicates the correct position for Z for in the table. For example:

```
SET INCOME1 TO INCOME8 (1,1.5,2.2,2.5,3,3.1, 3.5, 4)
COMPUTE LOC = SRCH (INCOME1,INCOME8,SALARY)
```

If SALARY has the value 3.5, the function returns 7, because the seventh variable has the value 3.5. If SALARY has the value 2, the function returns the value -3 indicating that the value is not present, and that the correct place in the list is in the third position.

SREAD

```
str = SREAD(prompt)
```

Pops up a box on the screen with the specified prompt and returns a string from the user. The maximum input is 4094 characters; long strings are scrolled horizontally.

SRST

```
num = SRST (strX ,strY)
```

Returns the column number within strX that matches strY. If strY is delimited by characters other than letters or numbers, SRST returns a positive number, otherwise SRST returns a negative value. If strY is not a substring of strX, SRST returns a zero value. The length of strX must be greater than or equal to the length of strY. For example:

```
POS1 = SRST ('BUBBLE GUM','GUM') results in: POS1 = 8
POS2 = SRST ('ANITA TINKLE',' ') results in: POS3 = -6
```

STATTYPE

```
num = STATTYPE ( rtnum , varname_str )
```

Indicates if the variable is a control or observation variable. If the record number (rtnum) is negative, the function applies to a summary variable; if rtnum is one more than the maximum record count (i.e. NRECS(0)+1) then this applies to a standard variable.

0 = not a control or observation variable

1 = observation var

2 = control var

STDEV

```
num = STDEV(X1 , X , . . . . , Xn)
```

Returns the standard deviation for the values in the list that are not missing or undefined. If fewer than 2 values are not missing or undefined, a value of undefined is returned.

STDEVR

```
num = STDEVR( varname )
```

Returns the standard deviation for the values of the specified variable encountered during a `PROCESS REC` or `PROCESS ROWS` loop that are not missing or undefined. If fewer than 2 values are not missing or undefined, a value of undefined is returned.

STDNAME

```
name = STDNAME( name )
```

Checks if name is standard and puts curly brackets around non-standard names. The function ignores leading and trailing spaces and any trailing characters over legal name length. If the name has leading or trailing quotes or curly brackets, these are stripped off. A standard name starts with an uppercase letter and contains only uppercase letters, digits or the four characters \$, #, @, and _. If it is a standard name, it is returned left justified. (Note that this function does *NOT* translate lower case letters to uppercase. Any lowercase means that the name is non-standard.

If it is a non-standard name, it is wrapped in curly brackets and returned left justified. If a name has embedded curly brackets, undefined is returned.

SUBDIR

```
str = SUBDIR (dir_str,sub_str)
```

Concatenates a subdirectory name to a directory path in correct system specific manner e.g.: `DIR = SUBDIR(CURDIR(0), "data")` returns a string like `C:\SIR_XS\data\` under windows or `/usr/SIR/XS/data/` under unix.

SUBSTR

```
str = SUBSTR(string,start,len)
```

Same as `SBST` function. Returns a null string if the starting position is outside the length of the string.

SUM

```
num = SUM(X1 , X ,....., Xn)
```

Returns the sum of the values in the list that are not missing or undefined. The maximum number of variables allowed in the argument list is 128. Arguments must be numeric.

SUMR

```
num = SUMR( X )
```

Returns the sum of the values of X encountered during a PROCESS REC or PROCESS ROWS loop that are not missing or undefined.

SVVAL

```
str = SVVAL ( rtnum , varname_str , n )
```

Returns the value of the nth valid value of a categorical string variable. If the record number (rtnum) is negative, the function applies to a summary variable; if rtnum is one more than the maximum record count (i.e. NRECS(0)+1) then this applies to a standard variable.

SYSTEM

```
num = SYSTEM( X )
```

Extracts a wide variety of information from an executing VisualPQL program set. Some return undefined if not in a RETRIEVAL.

Number	Return Value
SYSTEM(1)	The CPU time elapsed since beginning of run
SYSTEM(2)	A platform number, for example: 3 Sun Risc Solaris 7 Compaq Tru64 Unix 8 Compaq Alpha OpenVMS 10 IBM AIX 12 HP 9000 HP-UX 13 Silicon Graphics IRIX 26 Intel Linux 27 MS Windows 28 Macintosh
SYSTEM(3)	The update level for the current case/record. This is the update level at which this was last written. If it has previously been updated in the current update run, it is 1 greater than the database update level (system(23)).
SYSTEM(4)	1 if current case is available for processing, returns a 0 if current case is not available for processing
SYSTEM(5)	1 if current case has been modified, returns a 0 if current case has not been modified
SYSTEM(6)	1 if current record is available for processing, returns a 0 if current record is not available for processing
SYSTEM(7)	1 if current record has been modified, returns a 0 if current record has not been modified

- SYSTEM(8) If current case is available for processing, returns the number of records of all types belonging to the current case
- SYSTEM(9) The current output file page number
- SYSTEM(10) The lines remaining on current output file page
- SYSTEM(11) The total number or errors in the session
- SYSTEM(12) The number of errors during the current task
- SYSTEM(13) The number of warnings during the current task
- SYSTEM(14) 1 if last CASE IS block was executed, returns a 0 if last CASE IS block was not executed
- SYSTEM(15) 1 if last CASE IS block created a case, returns a 0 if last CASE IS block did not create a case
- SYSTEM(16) 1 if last RECORD IS block was executed, returns a 0 if last RECORD IS block was not executed
- SYSTEM(17) 1 if last RECORD IS block created a record, returns a 0 if last RECORD IS block did not create a record
- SYSTEM(18) The current row block number
- SYSTEM(19) The current row block position. Can be used to save a row position and retrieve the data with an OLD ROW IS AT (block,pos)
- SYSTEM(20) The number of cases (CIRs) created during current run
- SYSTEM(21) The number of cases (CIRs) updated during current run
- SYSTEM(22) The number of cases (CIRs) deleted during current run
- SYSTEM(23) The database update level
- SYSTEM(24) The number of cases in database
- SYSTEM(25) The number of data records in database
- SYSTEM(26) The line width of current output page
- SYSTEM(27) 1 if the last ROW IS block was executed, returns a 0 if the last ROW IS block was not executed
- SYSTEM(28) 1 if the last ROW IS block created a row, returns a 0 if the last ROW IS block did not create a row
- SYSTEM(29) 1 if the current row is available for processing, returns a 0 if the current row is not available for processing
- SYSTEM(30) 1 if the current row was modified, returns a 0 if the current row was not modified
- SYSTEM(31) The row ordinal of the current row
- SYSTEM(32) The number of rows in the table
- SYSTEM(33) Not used
- SYSTEM(34) The amount of table space, in SIR double words, used for the VisualPOL

execution stack

- SYSTEM(35) The amount of table space, in SIR double words, used for the program schema maps
- SYSTEM(36) 1 if the current record is available. A 0 (zero) is returned if access to the current record is denied for concurrent operations because the record is locked by another process with a non-compatible lock type
- SYSTEM(37) 1 if the current CIR is available. A 0 (zero) is returned if access to the current CIR is denied because the CIR is locked by another process with a non-compatible lock type
- SYSTEM(38) 1 if the session is a concurrent session using Master. A 0 (zero) is returned if this is a normal, single-user session
- SYSTEM(39) The ordinal number of the default database. No connected database returns 0
- SYSTEM(40) Indicates the number of connected databases. This returns the size of the connected database table that may include entries for disconnected databases since the position number associated with a particular connected database never changes.
- SYSTEM(41) The default string size
- SYSTEM(42) The Editor Type setting number
- SYSTEM(43) The Error Limit setting (num)
- SYSTEM(44) Encryption on for database (1 on, 0 off)
- SYSTEM(45) The current user has DBA rights (1 yes ,0 no)
- SYSTEM(46) The Page Length setting (num)
- SYSTEM(47) The Page Width setting (num)
- SYSTEM(48) The Loading Factor setting (num)-real
- SYSTEM(49) The Sort Number (SORTN) setting (num)
- SYSTEM(50) The Sort option (Obsolete)
- SYSTEM(51) The Warning Limit setting (num)
- SYSTEM(52) The Number of Attribute settings
- SYSTEM(53) The Number of Global variables set
- SYSTEM(54) The number of database data files. 0=standard
- SYSTEM(55) The Century split year
- SYSTEM(56) The number of Buffers defined
- SYSTEM(57) The number of families in the default procfile
- SYSTEM(58) Printback (1 on, 0 off)
- SYSTEM(59) Printback dorepeat (1 on, 0 off)
- SYSTEM(60) Printback calls (1 on, 0 off)

SYSTEM(61) Printback task stats (1 on, 0 off)
SYSTEM(62) Printback remarks (1 on, 0 off)
SYSTEM(63) Printback skipped commands (1 on, 0 off)
SYSTEM(64) Printback user created attributes (1 on, 0 off)
SYSTEM(65) Printback quiet (1 on, 0 off)
SYSTEM(66) Master Backup Interval
SYSTEM(67) Backup Count
SYSTEM(68) Number of Master clients
SYSTEM(69) Number of Master attached databases
SYSTEM(70) Password on default member? (1 Yes, 0 No)
SYSTEM(71) Default member type (1 :T; 2 :E; 3 :P; 4 :O; 5 :V; 6 :M)
SYSTEM(72) Default member public (1 Yes, 0 No)
SYSTEM(73) Length in bytes of default member
SYSTEM(74) Creation date of default member
SYSTEM(75) Creation time of default member
SYSTEM(76) Modification date of default member
SYSTEM(77) Modification time of default member
SYSTEM(78) Family password on default family? (1 Yes, 0 No)
SYSTEM(79) Lines in default member
SYSTEM(80) Status of window paging. Paging on returns 1

TABINDN

```
str = TABINDN (fn,tn,in)
```

Returns the index name of nth index. See TABINDS.

TABINDS

```
num = TABINDS(fn,tn)
```

Returns the number of indexes on nth table.

TABINDT

```
str = TABINDT (fn,tn,in,vn)
```

Returns the variable name and sort sequence of nth variable on index.

TABINDU

```
num = TABINDU (fn,tn,in)
```

Returns whether nth index is unique 0 - Not unique, 1 - Unique.

TABINDV

```
num = TABINDV (fn,tn,in)
```

Returns the number of variables in nth index.

TABNAME

```
str = TABNAME(fn,tn)
```

Returns the name of nth table. TFTABS(fn) returns number of tables on nth tabfile.

TABRECS

```
n = TABRECS(fn,tn)
```

Returns the number of rows on the nth table. TFTABS(fn) returns number of tables on nth tabfile.

TABVARS

```
num = TABVARS(fn,tn)
```

Returns the number of variables in nth table.

TABVINFN

```
num = TABVINFN(fn,tn,vn,n)
```

Returns various numeric data about the nth variable in nth table. The type of data is set by the fourth parameter as follows:

- 1 = Count of value labels
- 2 = leading zero
- 3 = print this column
- 4 = null not allowed
- 5 = ON if value labels printed
- 6 = set break variable
- 7 = option G on break
- 8 = option C on break
- 9 = option P on break
- 10 = var label as col heading
- 11 = unique flag
- 12 = subtotal title to the left
- 13 = count of ranges
- 14 = SIR data type

TABVINFS

```
str = TABVINFN(fn,tn,vn,n)
```

Returns various string data about the nth variable in nth table. The type of data is set by the fourth parameter as follows:

```
1 = variable label
2 = LNEG
3 = LPOS
4 = NULL
5 = ZERO
6 = TNEG
7 = TPOS
8 = SEPARATE
9 = date/time format (COL should have date/time type)
10= break string
```

TABVNAME

```
str = TABVNAME(fn,tn,vn)
```

Returns the variable name for vnth var on tnth table.

TABVRANG

```
str = TABVRANG(fn,tn,vn,rn)
```

Returns a string representation of the value(s) for the rnth range for vnth var on tnth table. String starts with keyword VALID or MISSING to indicate the type of range. Then string may have two values separated by :. Also may contain keywords BLANK, LOWEST and HIGHEST.

TABVTYPE

```
str = TABVTYPE(fn,tn,vn)
```

Returns the variable type for vnth var on tnth table.

TABVVALI

```
n = TABVVALI(fn,tn,vn,expr)
```

Validates the value in the expression (numeric or string) against the vnth var on tnth table. Returns a code indicating whether a value is allowed in a variable. The codes are:

```
0 = Valid value
Negative = Error detected
2 = Violation of specified valid values or ranges
```

3 - n Missing value 0 to n
4 = Missing value 1
5 = Missing value 2, etc.

TABVVLAB

```
str = TABVVLAB(fn,tn,vn,vln)
```

Returns the label for the vl^{nth} value label for vnth var on tnth table.

TABVVVAL

```
str = TABVVVAL(fn,tn,vn,vln)
```

Returns the value for the vl^{nth} value label for vnth var on tnth table.

TAN

```
num = TAN( X )
```

Returns the trigonometric tangent of X, where x is in radians. If X is an odd integral multiple of $\pi/2$ (e.g., $\pi/2$, $3\pi/2$, $5\pi/2$, etc.), the value of undefined is returned.

TANH

```
num = TANH( X )
```

Returns the hyperbolic tangent of X.

TFACCESS

```
str = TFACCESS(fn)
```

Returns the access type a(uto),r(ead) w(rite) of nth tabfile.

TFATTR

```
str = TFATTR(fn)
```

Returns the internal attribute name of nth tabfile. This is the same as the tabfile name.

TFCOUNT

```
num = TFCOUNT(dum)
```

Returns the number of connected tabfiles.

TFFILE

```
str = TFFILE(fn)
```

Returns the filename of nth tabfile.

TFGRNAME

```
str = TFGRNAME(fn)
```

Returns the group name of nth tabfile.

TFGRPW

```
str = TFGRPW(fn)
```

Returns the group password of nth tabfile.

TFJNNAME

```
str = TFJNNAME(fn)
```

Returns the journal name of nth tabfile.

TFNAME

```
str = TFNAME(fn)
```

Returns the name of nth tabfile.

TFTABS

```
num = TFTABS(fn)
```

Returns the number of tables on nth tabfile.

TFUSNAME

```
str = TFUSNAME(fn)
```

Returns the user name of nth tabfile.

TFUSPW

```
str = TFUSPW(fn)
```

Returns the user password of nth tabfile.

TIME

```
num = TIME( X )
```

Returns an integer that is the number of seconds from midnight. The input argument, *X*, is an integer in the range 0 - 235959; the first two digits are hours, the next two are minutes and the last two are seconds. For example, to calculate the number of seconds from midnight to 8:30 AM.

```
SLEEPSEC = TIME(083000)
```

TIMEC

```
str = TIMEC( X , time_format)
```

Converts an integer, *X*, into a time formatted string. See time formats for a complete description. Values of *X* that are out of range are returned as undefined. For example:

```
WAKESTR = TIMEC(ALARM, 'HH:MM:SS')
```

TIMEMAP

```
str = TIMEMAP ( rtnum, varname_str )
```

Returns a string with the time format (map) of the specified time variable. If the variable is not a time variable, undefined is returned. For example, `TIMESTR` equals "HH:MM:SS" if time variable `INTIME` has that time format. If the record number (*rtnum*) is negative, the function applies to a summary variable; if *rtnum* is one more than the maximum record count (i.e. `NRECS(0)+1`) then this applies to a standard variable.

```
TIMESTR = TIMEMAP (1, 'INTIME')
```

TODAY

```
num = TODAY( dummy )
```

Returns the "date integer" representation of the current date. The argument is a dummy numeric argument (specify 0).

TRIM

```
str = TRIM(string_expression)
```

Deletes trailing blanks from the string expression.

TRIML

```
str = TRIML(string_expression)
```

Deletes leading blanks from the string expression.

TRIMLR

```
str = TRIMLR(string_expression)
```

Deletes leading and trailing blanks from the string expression.

TRIMR

```
str = TRIMR(A)
```

See `TRIM` function.

TRUNC

```
num = (X [,n])
```

See `AINT` function.

TSTODT

```
date = TSTODT (timestamp)
```

Takes a real*8 timestamp (produced by `DTTOTS` and returns the date integer component.

TSTOTM

```
time = TSTOTM (timestamp)
```

Takes a real*8 timestamp (produced by `DTTOTS` and returns the time integer component.

TWRITE

```
str = TWRITE(string_expression)
```

Writes the specified string expression to the scrolled output window. This function can be useful when a program is run with an alternate output file (i.e., with the interactive `SET OUTPUT` command) and can only be used during an interactive session.

UPDLEVEL

```
num = UPDLEVEL (0)
```

Returns the current database update level. (Same as `SYSTEM(23)`.)

UPGET

```
str = UPGET (key_str)
```

Gets string value (User Preference) identified by `key_str` (from `sir.ini` file)
e.g. `COMPUTE TITLE= UPGET('SIR.TITLE')`

UPPER

```
str = UPPER(string_expression)
```

Changes lowercase letters to uppercase.

UPSET

```
num = UPSET (key_str, val_str)
```

Sets string value (User Preference) identified by key_str (in sir.ini file). Returns zero for success, -1 for failure.

e.g. COMPUTE rc= UPSET('SIR.TITLE', 'SIR/XS')

VALIDATE

```
num = VALIDATE (rtnum, varname_str ,value)
```

Returns a code indicating whether a value is allowed in a variable. If the record number (rtnum) is negative, the function applies to a summary variable. The codes are:

```
0 = Valid value
1 = Wrong data type/Not valid value
2 = Violation of specified valid values or ranges
3 = Missing value 0 (Undefined or system missing value)
4 = Missing value 1
5 = Missing value 2
6 = Missing value 3
```

VALLAB

```
str = VALLAB( varname )
```

Returns a character string containing the value label for the current value of the specified variable. If there is no label defined for the value, a zero length string is returned. The argument is a variable name, not a constant or expression.

VALLABSC

```
str = VALLABSC (rtnum, varname_str, value)
```

Returns the value label (up to 78 characters) for the specified value of a variable. If the record number (rtnum) is negative, the function applies to a summary variable; if rtnum is one more than the maximum record count (i.e. NRECS(0)+1) then this applies to a standard variable.

The value can be numeric or string.

For example, suppose the fourth value label for a variable DIVISION in record type 2 has value 10, label 'Head Office' then:

```
THISPOS = VALLABSC (2, 'DIVISION',10)
```

returns 'Head Office'.

VALLABSN

```
str = VALLABSN ( rtnum, varname_str , n)
```

Returns the nth value label for a variable (up to 78 characters). If the record number (rtnum) is negative, the function applies to a summary variable; if rtnum is one more than the maximum record count (i.e. NRECS(0)+1) then this applies to a standard variable. Note that the number of labels is returned by NVALLAB.

For example, suppose the fourth value label for a variable `DIVISION` in record type 2 has value 10, label 'Head Office' then:

```
THISPOS = VALLABSN (2, 'DIVISION',4)
```

returns 'Head Office'.

VALLABSP

```
n = VALLABSP ( rtnum, varname_str , value)
```

Returns the position (nth) of the specified value associated with value labels for a variable. If the record number (rtnum) is negative, the function applies to a summary variable; if rtnum is one more than the maximum record count (i.e. NRECS(0)+1) then this applies to a standard variable.

The value can be numeric or string.

For example, suppose the fourth value label for a variable `DIVISION` in record type 2 has value 10, label 'Head Office' then:

```
POS = VALLABSP (2, 'DIVISION',10)
```

returns 4.

VALLABSV

```
str = VALLABSV ( rtnum, varname_str , n)
```

Returns the nth value associated with value labels for a variable. If the record number (rtnum) is negative, the function applies to a summary variable; if rtnum is one more than the maximum record count (i.e. NRECS(0)+1) then this applies to a standard variable. Note that the number of labels is returned by NVALLAB.

For example, suppose the fourth value label for a variable `DIVISION` in record type 2 has value 10, label 'Head Office' then:

```
THISPOS = VALLABSV (2, 'DIVISION',4)
```

returns '10'.

VARDOSN

```
str = VARDOSN ( rtnum, varname_str,line_no )
```

Returns the nth line of documentation for a variable. Use the `NVARDOC` function to find total number of lines. (Note that variable documentation does not apply to summary variables. Simply use comment lines in programs to document.) In the following example, if `MARSTAT` in record type 1 has a single line of variable documentation of "Current marital status of employee", then `COLDESC` is set to that string value."

```
COLDESC = VARDOSN (1, 'MARSTAT', 1)
```

VARGET

```
str = VARGET (expression)
```

Returns a string representation of the value in the variable named in the expression. This works on all variable types. It converts catvar, date, time, integer and real to a string according to the format of the specified variable as per the `VFORMAT` function. Specify a string variable or expression that contains the name of another variable. For example:

```
INTEGER*1 INT1
STRING*8 STR1 STR2
COMPUTE INT1 = 1; STR1 = 'INT1'
COMPUTE STR2 = VARGET (STR1)
WRITE STR1 STR2
```

```
Output is:   INT1          1
```

VARLAB

```
str = VARLAB( varname )
```

Returns the variable label (up to 78 characters) for a variable. If there is no label defined for the variable, the variable name is returned. The argument is a variable name, not a constant or expression. If the function is compiled in a record block, a record variable can be specified. If the function is compiled in a case block, a common variable can be specified. If the function is outside and record or case block, then specify a local, summary variable.

VARLABSC

```
str = VARLABSC ( rtnum, varname_str )
```

Returns the label (up to 78 characters) for a variable. If there is no label defined for the variable, the variable name is returned. The first argument is the record number where the variable occurs or 0 for common variables. Use a negative value to specify a local, summary variable. The second argument is a constant (in quotes) or an expression that resolves to a variable name. In the following example, if MARSTAT in record type 1 has a variable label of "Marital Status", then COLHEAD equals "Marital Status".

```
COLHEAD = VARLABSC (1, 'MARSTAT')
```

VARNAME

```
string = VARNAME (rtnum,varnum)
```

Returns the name of the specified variable. The first argument is the record number where the variable occurs or 0 for common variables. Use a negative value to specify a local, summary variable. The second argument is the number of the variable (not including common vars). Variables are numbered in the order they are defined in the record schema definition. If the name is a non-standard name, it is returned enclosed in curly brackets {}.

VARNAMEC

```
string = VARNAMEC (rtnum,varnum)
```

Returns the name of the specified variable. The first argument is the record number where the variable occurs; the second argument is the number of the variable (including common vars). Variables are numbered in the order they are defined in the record schema definition.

VARLENG

```
num = VARLENG (rtnum,varname)
```

Returns the data length of the specified variable. The first argument is the record number where the variable occurs; the second argument is the name of the variable.

VARPOSIT

```
num = VARPOSIT (rtnum,varname)
```

Returns the position in the data record of the specified variable. The first argument is the record number where the variable occurs; the second argument is the name of the variable.

VARPUT

```
num = VARPUT(var_exp,string)
```

Places the value from the second argument into the first argument converting from a string to a numeric value if necessary. `Var_exp` contains the name of a variable. If the variable is a numeric variable, the string is converted to the appropriate numeric value and stored in the variable.

Returns the same numeric values as `VALIDATE`.

```
0 = Valid value
1 = Wrong data type/Not valid value
2 = Violation of specified valid values or ranges
3 = Missing value 0 (Undefined or system missing value)
4 = Missing value 1
5 = Missing value 2
6 = Missing value 3
```

For example:

```
COMPUTE RES = VARPUT ('BIRTHDAY', '12/31/50')
```

VARTYPE

```
num = VARTYPE( varname_str )
```

Returns an integer code representing the type of the specified variable. The argument is a string variable, quoted string constant or string expression whose value is the name of a common, record or local variable. Codes are:

```
0          if variable is string
1          if variable is numeric
undefined if variable is undefined
```

VFORMAT

```
str = VFORMAT ( rtnum , varname_str )
```

Returns a string representing the format of the specified variable (for example, A20, F9.2, D'MMIDDYY', etc.) If the record number (`rtnum`) is negative, the function applies to a summary variable; if `rtnum` is one more than the maximum record count (i.e. `NRECS(0)+1`) then this applies to a standard variable.

In the following example, if variable `NEWSAL` in record type 3 is a four digit integer, then `XFMT` equals "I4".

```
XFMT = VFORMAT ( 3, 'NEWSAL' )
```

VTYPE

```
num = VTYPE ( rtnum , varname_str )
```

Returns an integer code representing the type of the variable, where:

```
1 = string
2 = categorical
3 = date
4 = time
5 = integer
6 = real (single precision)
7 = real (double precision)
8 = scaled variable
```

If the record number (rtnum) is negative, the function applies to a summary variable; if rtnum is one more than the maximum record count (i.e. NRECS(0)+1) then this applies to a standard variable.

WACCESS

```
num = WACCESS ( dummy )
```

Returns the write security access level of the current user (the level corresponding to the write security password entered).

WINCNT

```
num = WINCNT ( 0 )
```

Returns the number of lines in the output window

WINLIN

```
str = WINLIN (n)
```

Returns the nth line from the output window

WINMOVE

```
num = WINMOVE (x,y,w,h)
```

Moves and resizes the main window. The window is placed at horizontal position x, vertical position y, size width w by height h. The horizontal units are 1/4 of the average width for the font being used. The vertical units are 1/8 font height. If w or h is zero the window is minimized; if w or h < 0, the window is restored to its original position (before being minimised).

WINPOS

```
num = WINPOS (line,pos,len)
```

Moves to line in output window and highlights from pos to len on that line

WINSELL

```
num = WINSELL(n)
```

Returns the line selected. 0 returns the current cursor line. 1 returns beginning line of selection. 2 returns ending line of selection.

WINSELP

```
num = WINSELP(n)
```

Returns the position in the line selected. 0 returns the current cursor position. 1 returns beginning position of selection. 2 returns ending position of selection.

WRECSEC

```
num = WRECSEC ( rtnum )
```

Returns the write security level for a record type.

WVARSEC

```
num = WVARSEC ( rtnum , varname_str )
```

Returns the write security level for a variable.

YESNO

```
num = YESNO( strX )
```

Displays a question box with the specified text and returns response. 1 indicates Yes; 0 indicates No.

The VisualPQL Debugger

Errors may be encountered in developing VisualPQL programs. Syntax may be incorrectly specified. Syntax errors are detected by the VisualPQL compiler and reported with Error and Warning Messages during compilation.

Errors may also occur during program execution that prevent successful running of the program. Typically these errors might be files not being found, when no provision has been made to handle the error or subscript references outside the bounds of the allowable range.

The program may compile and run but not perform as expected. Errors in programming logic are typically referred to as program **bugs**. The VisualPQL GUI debugger can assist in locating program bugs.

Error Messages

Error messages can be issued at compile time or at run time. Error messages have a number and explanatory text. These are printed at the point where the error has occurred. Errors can be related back to the original source line through line numbers. As the source lines are read, these are assigned an input number. This is a two part number such as 2.3 or 3.7. The first part refers to the *level* of the source input. If source is read directly, it is level 1, if it is included through a `CALL`, it is level 2 and so on. Various text inclusion commands increase this level, some of which are internal. For example, any text `RUN` through the menu system starts at level 2.

Each line is then assigned a sequential input line number within that level. The lines that cause any text inclusion are also assigned numbers. Some text inclusion commands are generated internally. Lines generated through a `DO REPEAT` do not have numbers since they are not part of the input text.

The VisualPQL program lines are also assigned an internal address referred to as the *Stack Location*. This is where that command is located during the execution of the program and is referred to by any run time error message. This number ascends but has gaps because most commands take multiple stack locations.

For a full listing of the input use `PRINT BACK ON`. This gives a listing such as:

```

1.1          CALL TEST.ERROR:T
2.1      00001 program
2.2      00001 compute a = 1
2.3          call inctext
3.1      00003 a = 1
3.2      00005 a = 1
3.3      00007 a = 1
2.4          do repeat x = 1 2 3
2.5          compute a = x
2.6          end repeat
(REPEAT) 00009 compute a = 1
(REPEAT) 00011 compute a = 2
(REPEAT) 00013 compute a = 3
2.7      00015 end program

```

Debug Mode

If the `DEBUG` keyword is specified on the `PROGRAM`, `SUBROUTINE` or `RETRIEVAL` command, the routine is compiled in *debug* mode. and the debuggers can be used with the routine. This assigns an internal line number to each command or part of a command that can be executed and stores the source text of the routine as part of the executable.

Compile Errors

If an error is found during a compilation, the line where the error is detected is listed together with an explanatory message. The line has both the input line number and the stack location listed.

Run Time Errors

If an error is detected during the execution of a program, an error message is displayed. This lists the error together with some information to enable you to locate the code that caused the problem. This is referred to as a "Traceback of error condition". It lists the routine, a line number and a stack location.

The routine is "MAIN" if the main program is being executed or is the name of a sub-routine that is being executed.

If the routine has been compiled in debug mode, then an internal line number is assigned to executable parts of the code and this line number is displayed during a debug run. It is *not* the input sequence number. If the routine has been compiled in debug mode, the source text of the line causing the problem is also displayed.

The stack location is always displayed and is the stack location listed on the `PRINT BACK`. Use this to locate the line causing the problem.

Overview to the VisualPQL GUI Debugger

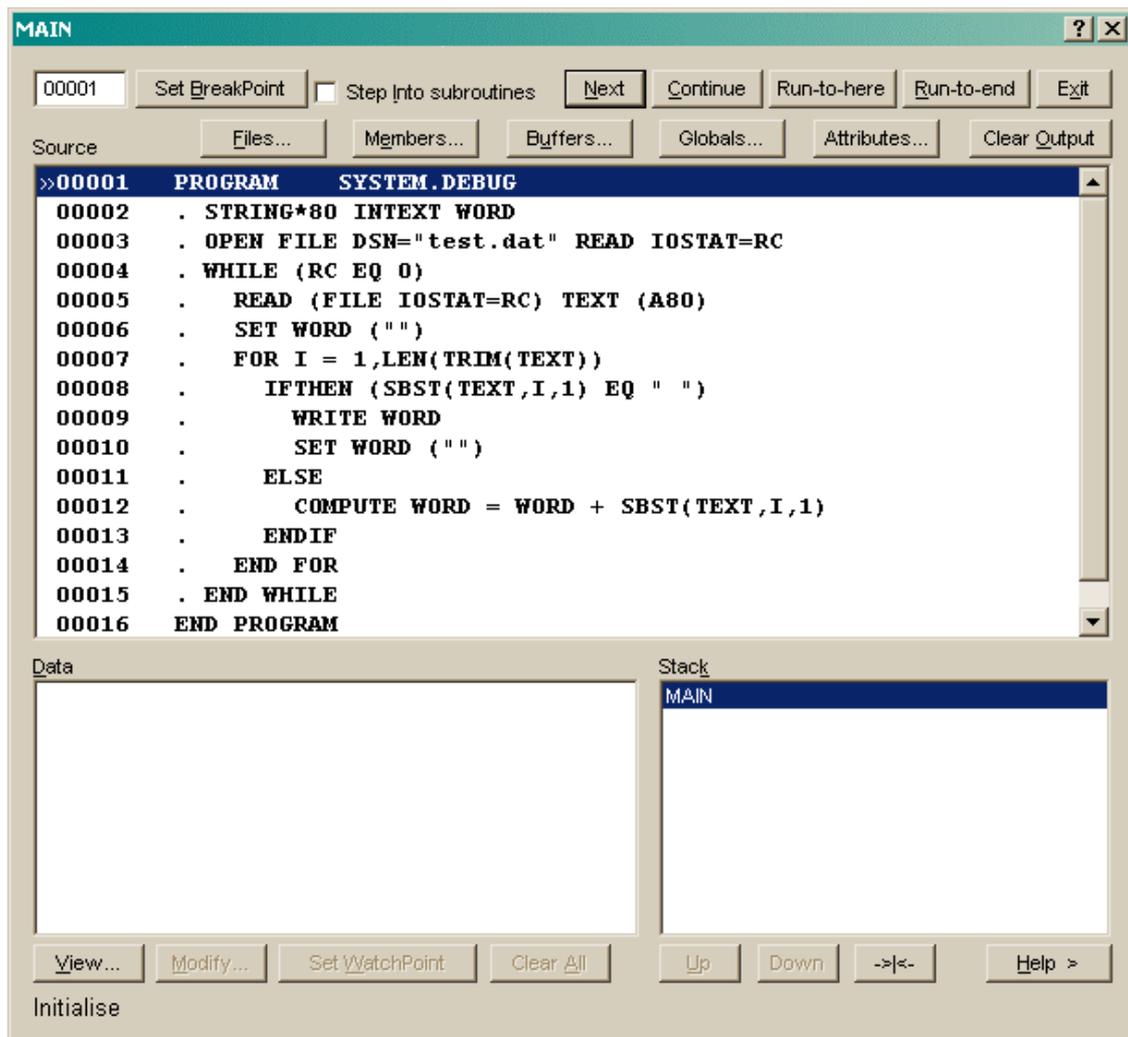
A debugger provides a means of following the execution path through the program statements. Execution can be suspended at any point to examine the values of program variables. In this way the developer can determine where an incorrect branch is taken or if a variable has the expected value.

The VisualPQL debugger is itself a VisualPQL program that uses particular debug functions to interact with the executing system. If you wanted to, you could use these functions to develop a different interface to the debugger but you do not need to use these functions simply to debug your own programs. To avoid confusion, these functions are not documented here, please contact SIR support for details, if you need them.

Using the debugger requires additional memory beyond that needed by the program. As a VisualPQL program is compiled, source commands are converted into operating codes that are the executable program and these are held in the command stack. When the `DEBUG` option is specified on the `RETRIEVAL`, `PROGRAM` or `SUBROUTINE` commands, the compiler keeps the text of each command along with the corresponding operating code in the stack (which is why the debugger requires much more memory). Each command in the stack is assigned a *command number* that is used for internal reference and is used on several commands that display the contents of the stack. These numbers are not editor line numbers or the line numbers in a source code listing.

The following discusses the VisualPQL debugger and the options that are available.

To start the debugger, first compile a program with the `DEBUG` keyword then select **Debug...** from the **Program** menu. Select the compiled object (`SYSTEM.DEBUG:0` is the default) from the member list.



The VisualPQL Debugger

Source

The currently selected line number is displayed in the top left. This is not necessarily the line that is about to be executed but it is the line that is effected by the **Set.BreakPoint** and **Run-to-here** buttons.

The **Set BreakPoint** button sets or clear a breakpoint on the selected line. A breakpoint is a command where execution stops *before* that command is executed.

The **Step Into Subroutines** box if checked and subroutines are compiled with debug, means that the source of the subroutines are displayed when they are executed.

Press **N**ext to execute next command and moves the selected line to the next executable line.

Press **C**ontinue to run until a breakpoint or watchpoint is hit or the program ends.

Run-to-here executes all lines until the selected line is reached.

Run-to-end executes the program ignoring all breakpoints and watchpoints.

Exit stops debugging, stops the program execution and closes the debugger.

The **F**iles, **M**embers and **B**uffers buttons let you open, edit files members and buffers.

The **G**lobals and **A**tttributes buttons let you view and modify global variables and file attributes.

Clear O**utput** clears the main window output area.

The Program source code lines are displayed in the large listing area. Note that the line that is about to be executed is marked with a » character and the selected line is highlighted.

Data

The Bottom left of the debugger deals with program variables. Selected variables and their values are displayed in the list. By default, no variables are selected for display.

Press **V**iew to select variables of interest or to change the order that they are displayed. The most recently selected are displayed first.

Press **M**odify to change the value of a variable.

Press **S**et W**atchpoint** to set a watchpoint on or off on the selected variable. A watchpoint means that execution stops on the command *after* the command that changes the value in that variable. An active watchpoint is indicated with a ☒

Press **C**lear A**ll** to remove all watchpoints.

Double click on a variable to modify its value.

When a variable is selected from the list, its value is displayed at the bottom of the screen. If it is a string variable then blanks are shown with a dot (·).

Stack

The bottom right of the debugger shows the program stack. This indicates the levels of called subroutines that have been traversed to get to the current point and lets you view and navigate through levels of subroutine source. The routine currently being executed is shown at the top of the list.

Press **Up** to move to the source that called the currently displayed source.

Press **Down** to return one level to the source of the called subroutine (after pressing up).

Press **->|<-** to return to the subroutine source and line about to be executed.

You can double click on a routine to view its source.

How to debug a program

A small program could be debugged by viewing all variables and pressing Next through each line of the program until the problem is detected. View the values of variables and expressions to determine why the program is behaving unexpectedly.

In a larger program, you probably need to set breakpoints or watchpoints to stop execution and return to the debugger when a particular line is about to be executed or when the value of a given variable is changed. After setting a breakpoint on a source line or a watchpoint on a variable, press continue to execute the program. The following example is a small program that is easily debugged but is worked through in detail to show a debugging process.

This example program is meant to extract first name, last name and middle initial from a string containing a full name. The problem is that last name is not being calculated properly.

```

SUBROUTINE FMLNAME (NAME) RETURNING (FNAME,MINIT,LNAME) REPLACE
NODATABASE DEBUG DYNAMIC
. STRING*50 NAME
. STRING FNAME MINIT LNAME
. INTEGER FSPACE LSPACE
. SET FNAME MINIT LNAME ("" )
. COMPUTE FSPACE = ABS(SRST(NAME," "))
. COMPUTE LSPACE = LEN(NAME) + 1 - ABS(SRST(REVERSE(NAME)," "))
. COMPUTE FNAME = SBST(NAME,1,FSPACE-1)
. COMPUTE LNAME = SBST(NAME,LSPACE+1,LEN(NAME)-LSPACE)
. IF (LSPACE NE FSPACE) COMPUTE MINIT = SBST(NAME,FSPACE+1,1)
END SUBROUTINE

RETRIEVAL DEBUG
. STRING FNAME MINIT LNAME
. PROCESS CASES
.   PROCESS RECORD 1
.     EXECUTE SUBROUTINE FMLNAME (NAME) RETURNING (FNAME,MINIT,LNAME)
.     WRITE NAME
.     WRITE FNAME " / " MINIT " / " LNAME
.   END RECORD
. END CASE
END RETRIEVAL

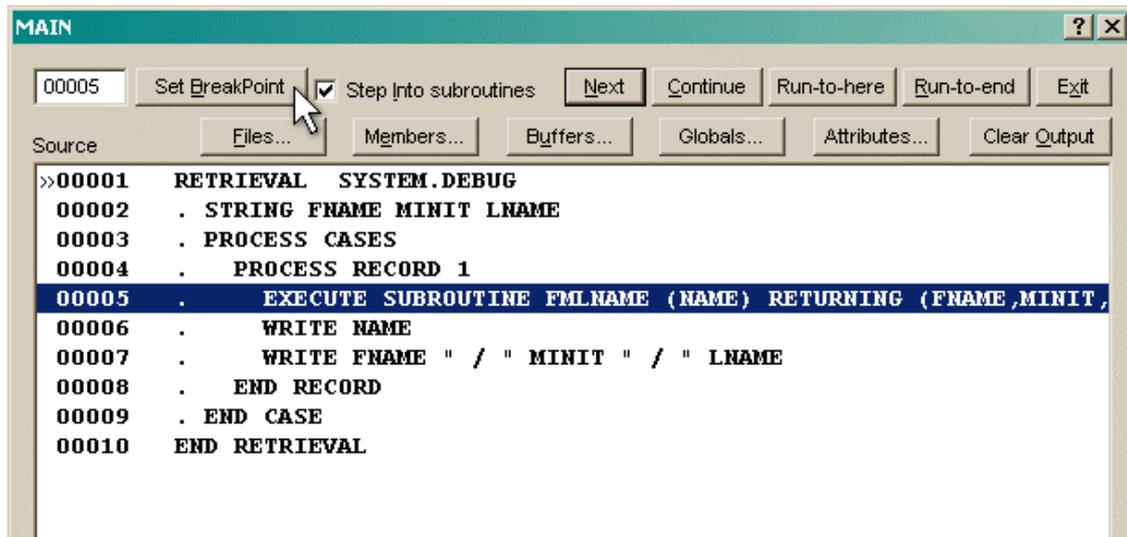
```

```

Start retrieval execution
John D Jones
John / D / *
James A Arblaster
James / A / *
Mary Black
Mary / B / *
...

```

The problem is in extracting components from the variable NAME in record 1 so start by setting a breakpoint near the start of that record block.



Setting a BreakPoint

Then press continue. Execution stops on that line.

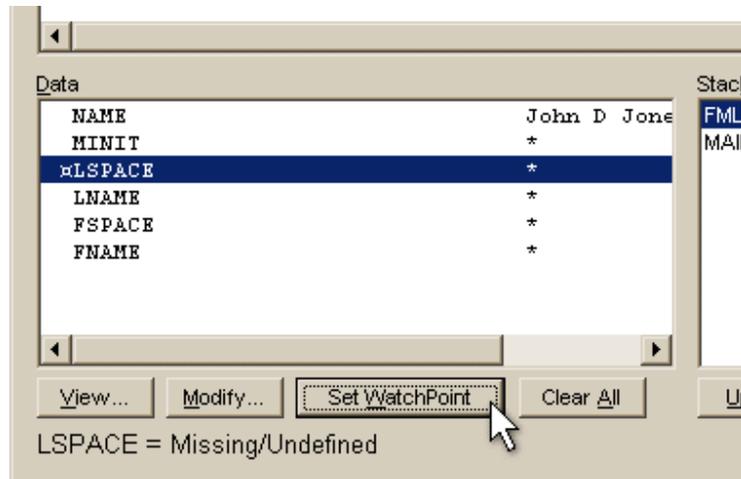
Check the step into subroutine box so that you are able to debug the called subroutine and press next. You should now be looking at the subroutine source.

We know the computation of last name (and middle initial) is not working so put a break on

```
. COMPUTE LNAME = SBST(NAME,LSPACE+1,LEN(NAME)-LSPACE)
```

In the Data section Press View, select all variables and press OK. Looking at the data section we see NAME is *John D Jones*, LSPACE (the last occurrence of blank in the name string) is 25 where we were expecting 7.

At this point we could work out the problem but for sake of the example, put a watchpoint on LSPACE so we can find the next time it is modified and clear the breakpoint on LNAME (select it and press set breakpoint).



Setting a WatchPoint

Press continue and execution stops at the breakpoint at the start of the record 1 block in the main routine.

Press continue again and execution stops after the line that has set the value of LSPACE:

```
. COMPUTE LSPACE = LEN(NAME) + 1 - ABS(SRST(REVERSE(NAME), " "))
```

Select NAME from the data list and you see its value displayed at the bottom of the screen padded with blanks to 25 - so the last blank is at 25.

Repeat the process but press next rather than continue when executing the subroutine. Use the modify button in the data section to remove trailing blanks from the name and step through the computes to see that the program is working correctly.

So the fix is to remove the trailing blanks from NAME when calculating the positions of the blanks.

```
...
. COMPUTE NAME = TRIM(NAME)
. COMPUTE FSPACE = ABS(SRST(NAME, " "))
. COMPUTE LSPACE = LEN(NAME) + 1 - ABS(SRST(REVERSE(NAME), " "))
...
```

Exit the debugger, make the program change and run it again.

- OPERATOR	75	ARRAYS	93
\$ PICTURE	130	ARRDIMN FUNCTION	328
* ASTERISK	82	ARRDIMST FUNCTION.....	328
* I/O FORMAT	124, 131	ARRDIMSZ FUNCTION.....	328
* OPERATOR	75	ARSIN FUNCTION.....	328
** OPERATOR.....	75	ASIN FUNCTION	328
, PICTURE.....	130	AT	275
. PERIOD.....	19	ATAN FUNCTION.....	329
. PICTURE.....	130	ATTRIBUTE NAME	329
/ NEW LINE.....	129	ATTRNAME FUNCTION	329
/ OPERATOR	75	AUTO	263, 267, 275
SUFFIX	92	AUTOKILL LIMIT.....	344, 371
:V SUFFIX.....	92	AUTOSET	78
; SEMI-COLON	19	AVERAGE VALUE FUNCTION.....	351
VERTICAL BAR.....	19	B FORMAT.....	124, 131
+ OPERATOR	75	BACKUP	160
=	96	BEEP	238
A FORMAT	124, 131	BEGIN	102
A PICTURE	125	BINARY	117
ABS FUNCTION	327	BINARY FORMATS	125
ABUTTON.....	280	BIND STATEMENT.....	194
ACCEPT RECORD IF.....	269	BINDPARM FUNCTION	329
ACOS FUNCTION.....	327	BLANK	62, 87
ACROSS RECORD FUNCTIONS.....	305	BLANK REPLACEMENT	342
ADDITION FUNCTION.....	375	BORDERS	217
AFTER	101, 157, 187	BRANCH FUNCTION	329
AINT FUNCTION	327	BRANCHD FUNCTION.....	329
ALL	84, 146, 234	BRANCHN FUNCTION.....	329
ALL, GET VARS	81	BROWSE DIRECTORY	238
ALOG FUNCTION.....	327	BROWSE FILE.....	238
ALOG10 FUNCTION	327	BUFFER NAME	329
AMOD FUNCTION.....	327	BUFNAME FUNCTION.....	329
AND	97	BUTTON	219, 234
APPDIR FUNCTION	327	C PICTURE.....	125
APPEND.....	117	IF 276	
APPEND ITEM.....	236	CALL SCREEN	275
APPEND LINE	236	CAPITAL FUNCTION	329
APPLICATION DIRECTORY.....	327	CAPITALISATION FUNCTION	329
ARC COSINE FUNCTION	327	CASE IS	140
ARC SINE FUNCTION.....	328	CASE PROCESSING.....	146
ARCOS FUNCTION.....	328	CASELOCK	170
ARCTANGENT FUNCTION	329	CASELOCK FUNCTION	330
ARGUMENT LIST FUNCTIONS	304	CAT VAR FUNCTION	330
ARRAY	23, 53	CAT VARS	57

CATEGORICAL VARIABLES	57	COMPILATION ERRORS.....	393
CATINT FUNCTION.....	330	COMPUTE.....	79
CATSTR FUNCTION	330	CONCATENATING STRINGS	75
CDATE FUNCTION.....	330	CONCURRENT VISUALPQL.....	170
CENTER FUNCTION.....	330	CONNECT DATABASE, PQL	135
CGI WRITE	127	CONNECT ODBC.....	192
CGIBUFPN FUNCTION	331	CONNECT TABFILE	178
CGIBUFSV FUNCTION	331	CONSTANTS	74
CGIVARPN FUNCTION.....	331	CONTINUATION LINES.....	19
CGIVARSV FUNCTION.....	331	CONTROL FLOW.....	94
CHANGE CIR LOCK TYPE	330, 331	CONTROL VARS	58
CHAR FUNCTION.....	331	COS FUNCTION	334
CHECK	220, 234	COSINE FUNCTION.....	334
CHECK ITEM	236	COUNT	139, 146, 185, 334
CHECK MENUITEM	213	CREATE BUFFER	292
CHOICE	221, 234	CRWARN.....	38, 56
CIRLOCK.....	38	CRYPTKEY	334
CIRLOCK FUNCTION	331	CTIME FUNCTION	334
CLEAR.....	263, 267, 280	CURDIR FUNCTION	334
CLEAR ALL	236	CURRENT DIRECTORY	334
CLEAR BUFFER	291	D FORMAT	124, 130
CLEAR DCONTROL.....	244	D PICTURE	125
CLEAR OUTPUT WINDOW	203	DATA	259
CLEAR SELECT ITEM.....	236	DATA FORMAT FUNCTION.....	390
CLIENT AUTOKILL	344, 371	DATA TYPE FUNCTION	390
CLIPAPP FUNCTION	331	DATABASE ACCESS	28
CLIPGET FUNCTION	332	DATABASE ACCESS OVERVIEW	132
CLIPLINE FUNCTION.....	332	DATABASE FUNCTIONS	316
CLIPSET FUNCTION.....	332	DATABASE IS	137
CLOSE.....	120	DATABASE NAME FUNCTION ...	336, 337
CLOSE TABLE	177	DATABASE TYPE FUNCTION.....	337
CLOSETABLE	185	DATABASE VARIABLES	77
FILES.....	120	DATABASE, PQL CONNECT	135
CNT FUNCTION	332	DATABASE, PQL DISCONNECT.....	136
CNTR FUNCTION.....	332	DATE.....	22, 59, 124, 131
COLCOUNT FUNCTION.....	332	DATE FUNCTIONS	306, 330, 334, 335
COLLABEL FUNCTION	332	DATE VARIABLES	59
COLLEN FUNCTION.....	333	DATEC FUNCTION.....	334
COLNAME FUNCTION	333	DATEMAP FUNCTION.....	335
COLTYPE FUNCTION.....	333	DATET FUNCTION	335
COLVALN FUNCTION	333	DBINDN FUNCTION	336
COLVALS FUNCTION	333	DBINDR FUNCTION	336
COMBO	223	DBINDS FUNCTION	336
COMMA FUNCTION.....	333	DBINDT FUNCTION	336
FIELD	272	DBINDU FUNCTION	336
COMPARE FLOATING POINT	341	DBINDV FUNCTION.....	336

DBNAME FUNCTION	337	DISPLAY SAVE BOX	239
DBTYPE FUNCTION	337	DISPLAY TEXT BOX	238
DEBUG	38	DISPLAY WDL	300
DEBUGGER	393	DISPLAY YES/NO BOX	238
VISUALPQL	393	DISPLAY YES/NO/CANCEL BOX	238
ARRAY	53	DITEM FUNCTION	338
DECRYPT FUNCTION	337	DITEMCOL FUNCTION	338
DEDIT	240	DITEMH FUNCTION	338
DEDIT MESSAGE	244	DITEMID FUNCTION	338
DEFAULT FAMILY NAME	337	DITEMROW FUNCTION	338
DEFAULT MEMBER NAME	337	DITEMS FUNCTION	338
DEFAULT TABFILE NAME	337	DITEMSEL FUNCTION	338
DEFFAM FUNCTION	337	DITEMSID FUNCTION	339
DEFINE PROCEDURE VARIABLES	93	DITEMTXT FUNCTION	339
DEFMEM FUNCTION	337	DITEMTYP FUNCTION	339
DEFTFN FUNCTION	337	DITEMW FUNCTION	339
DELDIR FUNCTION	337	DRAG AND DROP	211
DELETE	118, 120, 267, 280	DROPPFILE	211
DELETE BUFFER	293	DSN	118
DELETE CASE	141	DSN FUNCTION	339
DELETE CLIENT	338	DTTOTS FUNCTION	336
DELETE DIRECTORY	337	DYNAMIC	41, 45, 118
DELETE FILE	338	E FORMAT	124, 130
DELETE LINE IN BUFFER	294	EDIT	222, 234
DELETE PROCEDURE FILE MEMBER ..	121	EDIT BUFFER	295
DELETE RECORD	152	EDIT FUNCTION	339
DELETE ROW	180	EDITIN	273
DELFILE FUNCTION	338	EDITOR NAME FUNCTION	339
DELMCLID FUNCTION	338	EDITOUT	273
DGLOBAL FUNCTION	338	ARRAY	23
DIALOG	215	ELSE	87, 106
DIALOG DESIGN	239	ELSEIF	106
DIFFERENCE FILE COPY FUNCTION ...	371	ELSEIFNOT	106
DIMENSION	53	ENABLE ITEM	236
ARRAY	53	ENABLE MENUITEM	212
DIMENSIONS	23	ENABLE TIMER	235
DISABLE ITEM	236	ENCRYPT FUNCTION	339
DISABLE MENUITEM	212	ENCRYPTION KEY FUNCTION	334
DISABLE TIMER	235	END BEGIN	102
DISCONNECT DATABASE, PQL	136	END CASE	142
DISCONNECT ODBC	192	END DATABASE IS	137
DISPLAY ERROR BOX	238	END DIALOG	215
DISPLAY INFO BOX	238	END FOR	104
DISPLAY OK/CANCEL BOX	238	END IF	106
DISPLAY OPEN BOX	238	END INITIAL	210, 234
DISPLAY POPUP LIST	214	END JOURNAL RECORD	153

END LOOP	107	EXIT SUBPROCEDURE.....	113
END MENU	205	EXIT SUBROUTINE	49
END MESSAGE	211, 234	EXIT UNTIL	109
END PROCESS CASE	142	EXIT WHILE	111
END PROCESS JOURNAL	153	EXP FUNCTION.....	340
END PROCESS RECORD	153	EXPLICIT DECLARATIONS	51
END PROCESS ROW	181	EXPONENT FUNCTION	340
END PROGRAM	43	EXPRESSIONS	74
END RECORD	153	EXTERN FUNCTION.....	340
END RETRIEVAL	43	EXTERNAL VARIABLE BLOCK	91
END ROW	181	EXTERNALS	93
END SCREEN	270	EXTERNS FUNCTION.....	340
END SUBPROCEDURE.....	113	F FORMAT	124, 130
END SUBROUTINE.....	43	FAILFLD.....	251
END UNTIL.....	109	FAILMESS	252
END WHILE.....	111	FAILSCR.....	252
END WINDOW	198	FAMILY NAME FUNCTION	341
ENDMSG	38	FBUTTON	282
ENVIRONMENT VARIABLE FUNCTION	344	IF 282	
EQ	96	FDISPLAY.....	278
ERROR	118, 123, 127	FEQ FUNCTION.....	341
ERROR BOX.....	238	IF 274	
ERROR FUNCTION.....	340	FIELDIN.....	252
ERROR MESSAGE	354	FIELDOUT	252
ERROR MESSAGES	118, 287	FILE BROWSE.....	238, 341
ERRORS IN PROGRAMMING	393	FILE DETAILS.....	342
EVALUATE.....	80	FILE EXISTS	341
EXPRESSIONS	76	FILE I/O OVERVIEW.....	115
EXCLUDE.....	93, 284	FILE NAME.....	341
EXECUTE	38	FILE OUTPUT.....	341
EXECUTE DBMS	44	FILECNT FUNCTION	341
EXECUTE STATEMENT.....	194	FILEID	117, 123, 127
EXECUTE SUBPROCEDURE	114	FILEIN FUNCTION	341
EXECUTE SUBROUTINE	45	FILEIS FUNCTION	341
EXISTS FUNCTION.....	340	FILEN FUNCTION	341
EXIT	103, 280	FILENAME FUNCTION	339
EXIT BEGIN	102	FILEOUT FUNCTION	341
EXIT CASE.....	143	FILES IN DIRECTORY	341
EXIT FOR.....	104	FILESTAT FUNCTION	342
EXIT IF	106	FILETIME FUNCTION.....	342
EXIT JOURNAL IS	167	FILL FUNCTION	342
EXIT LOOP	107	FIND ITEM FUNCTION	343
EXIT MESSAGE.....	211	FIRST	280
EXIT PROCESS JOURNAL.....	168	FLOATING POINT	64
EXIT RECORD.....	154	FLOATING POINT COMPARE	341
EXIT ROW	182	FOCUS.....	235

FOCUS ITEM.....	236	GETRSTEP FUNCTION	347
FOR	104	GETTXT FUNCTION	347
FORM	263	GETTXTH FUNCTION	347
FORMAT FUNCTION.....	343	GLOBAL NAME FUNCTION.....	348
FILES	124	GLOBAL VARIABLE FUNCTIONS.....	307
FRACTION	147	GLOBALN FUNCTION	347
FROM	158, 187	GLOBALS FUNCTION	348
FST FUNCTION	343	GRID	245
FSTR FUNCTION	343	GT	96
FUNCTIONS.....	75	HELP	235, 276
VISUALPQL	75	HELP FUNCTION	348
GE	96	HEX FORMAT	125
GENERATE.....	284	HI 87	
GET	38	HIDE ITEM.....	236
GET INI FILE TEXT	385	HIGHEST	87
GET LINE FROM BUFFER.....	296	HYPERBOLIC TANGENT FUNCTION ...	382
GET VARS	56, 81	I FORMAT.....	124, 130
GETAKL FUNCTION.....	344	FORMATS.....	124, 130
GETBTNH FUNCTION	343	I/O SPECIFICATIONS	124, 130
GETCHCH FUNCTION	343	IB FORMAT	125
GETCHKH FUNCTION	344	ICHR FUNCTION	348
GETDFC FUNCTION	344	IDSTATUS FUNCTION.....	348
GETENV FUNCTION.....	344	IF 99	
GETERR FUNCTION	344	IFNOT.....	99
GETFLT FUNCTION	344	IFNOTTHEN.....	106
GETFOCUS FUNCTION.....	344	IFTHEN.....	106
GETICLK FUNCTION	345	IMAGE.....	225
GETIFLT FUNCTION	345	IMPLICIT VARIABLES.....	22, 56
GETIINT FUNCTION.....	345	INCLUDE.....	93, 284
GETINT FUNCTION	345	INCLUDE EXTERNAL VARIABLE.....	92
GETITXT FUNCTION	345	INCREMENT	146
GETLBLH FUNCTION.....	345	INDEX VARIABLE FUNCTION.....	336
GETLTXT FUNCTION.....	345	INDEXED BY	157, 185, 189
GETMAXCH FUNCTION	345	INFORMATION BOX	238
GETMCADD FUNCTION	345	INITIAL	210, 234, 267
GETMCHK FUNCTION	346	INPUT FORMATS	124
GETMCLID FUNCTION.....	346	INPUT FUNCTION	356, 374
GETMCLST FUNCTION	346	SUBROUTINE	38
GETMCON FUNCTION	346	INSERT DCONTROL.....	242
GETMDBN FUNCTION	346	INSERT ITEM	236
GETMSEL FUNCTION	346	INSERT LINE INTO BUFFER	297
GETNITEM FUNCTION.....	346	INSERT TEXT.....	236
GETNLINE FUNCTION	346	INTEGER	22, 61
GETNSEL FUNCTION	347	INTEGER FUNCTION.....	327
GETPOS FUNCTION	347	INTEGER RANGES	22
GETRADH FUNCTION	347	INTEGER VARIABLES.....	61

GRAPHIC INTERFACE PQL.....	32	LOG10 FUNCTION.....	350
FILES.....	27	LOGARITHM FUNCTION.....	327
ODBC PQL.....	31	VALUES.....	98
PQL FUNCTIONS.....	34	LOGICAL OPERATIONS.....	96
PQL ODBC.....	31	LOGICAL VALUES.....	98
PROGRAM FLOW.....	25	LOOKUP.....	172
TABFILE ACCESS.....	30	LOOP.....	107
IOSTAT.....	118, 123, 178, 179	LOWER FUNCTION.....	350
IOSTAT =.....	128	LOWEST.....	87
JOUFLAG FUNCTION.....	349	LRECL.....	118
JOURNAL FILE.....	163	LST FUNCTION.....	350
JOURNAL RECORD IS.....	166	LSTR FUNCTION.....	351
JOURNAL TYPE.....	164	LT.....	96
JOURNAL USER.....	165	MAIN ROUTINE.....	36
JOURNALS IN VISUALPQL.....	161	MAKEDIR FUNCTION.....	351
JULC FUNCTION.....	349	MASTER.....	170
DATE FUNCTIONS.....	349	MASTER DIFFERENCE FILE COPY FUNC	344
JULN FUNCTION.....	349	FUNCTIONS.....	303
JUMP.....	100	MATHEMATICAL FUNCTIONS.....	303
KEEP CIR.....	141	MAX FUNCTION.....	351
KEYBOARD INPUT.....	123	MAXR FUNCTION.....	351
KEYNAME FUNCTION.....	349	MAXRECS FUNCTION.....	351
KEYORDER FUNCTION.....	349	MEAN FUNCTION.....	351
L PICTURE.....	125	MEANR FUNCTION.....	351
LABEL.....	226	MEMBER.....	119, 123, 128
LABELS.....	70, 100	MEMBER COUNT FUNCTION.....	352
LAST.....	280	MEMBER INFORMATION FUNCTION ..	352
LAST VALUE FUNCTION.....	350, 351	MEMBER NAME FUNCTION.....	352
LE.....	96	MENU.....	205
LEN FUNCTION.....	350	MENUITEM.....	206
LG10 FUNCTION.....	350	MENUSEP.....	207
LIBRARY.....	39	MESSAGE.....	211, 234
LINE.....	227	MIN FUNCTION.....	353
LINES FUNCTION.....	350	MINR FUNCTION.....	353
LIST.....	147, 228, 234	MISNUM FUNCTION.....	353
LISTS OF VARIABLES.....	52	MISS FUNCTION.....	353
LN FUNCTION.....	350	MISSCHAR.....	39
LO.....	87	MISSING.....	82, 87
LOADING.....	39	MISSING FUNCTION.....	353
LOADMAP.....	39	MISSING VALUE FUNCTION.....	353
LOCAL VARIABLES.....	22, 50, 77	MISSING VALUES.....	23, 62, 73
LOCK.....	39, 140, 147, 151, 157	MKEYSIZE FUNCTION.....	353
LOCK =.....	170	MOD FUNCTION.....	354
LOCK TYPES.....	38	MODE.....	178
LOG BASE 10 FUNCTION.....	327	MODIFY DCONTROL.....	243
LOG FUNCTION.....	350	MODIFY DCONTROL FONT.....	243

MOVE VARS	81	NOFCASES FUNCTION	356
MRECSIZE FUNCTION	354	NOPROCS	40
MSGTXT FUNCTION	354	NOSAVE	91
NAMES	20	NOSIMPLE	93
VISUALPQL	20	NOT	96
NARG FUNCTION	354	NOTUPDLOG	42
NATURAL LOGARITHM FUNCTION	327	NOUPDATE.....	268
NBRANCH FUNCTION.....	354	NOUPDLOG	42
NE	96	NOVARMAP	42
NEW CASE IS	140	NOW FUNCTION	356
NEW LINE.....	129	NPUT FUNCTION	356
NEW RECORD IS	151	NREAD FUNCTION	356
NEW ROW IS.....	189	NRECS FUNCTION	356
NEXT.....	108, 280	NSUBDIR FUNCTION	356
NEXT CASE	144	NUMBER OF INDEXES FUNCTION	336
NEXT FOR	104	NUMBER TO STRING FUNCTION	343
NEXT LOOP	107	NUMBERS	74
NEXT MESSAGE	211	NUMBR FUNCTION.....	357
NEXT PROCESS HEADER	169	NUMCASES FUNCTION.....	357
NEXT PROCESS JOURNAL	169	NUMERIC CONSTANTS.....	74
NEXT RECORD	155	NUMRECS FUNCTION.....	357
NEXT ROW	183	NVALID FUNCTION	357
NEXT UNTIL	109	NVALLAB FUNCTION	357
NEXT WHILE	111	NVARDOC FUNCTION	357
NEXTROW FUNCTION	354	NVARS FUNCTION.....	358
NGET FUNCTION	355	NVARSC FUNCTION	358
NGLOBAL FUNCTION	355	NVVAL FUNCTION	358
NKEYS FUNCTION.....	355	OBSERVATION VARS	63
NLABELS FUNCTION.....	355	ODBC STATEMENT.....	194
NLEVELS.....	49	ODBCCOLS FUNCTION	358
NMAX FUNCTION.....	355	ODBTABS FUNCTION	358
NMIN FUNCTION.....	355	OK/CANCEL BOX.....	238
NMISSING	82	OLD CASE IS.....	140
NOARRAYMSG.....	39	OLD RECORD IS.....	151
NOARRAYS	93	OLD ROW IS	189
NOAUTO	263, 267	ONCALL	276
NOAUTOCASE.....	39, 112	ONETIME.....	157, 186
NOCLEAR.....	263, 267	OPEN	117
NOCRWARN	38	OPEN FILE BOX	238
NODATA	259	OPEN MESSAGES.....	118
NODATABASE.....	40, 263	OPEN TABLE	176
NOECHO.....	274	FILES.....	117
NOENDMSG.....	38	OPERATORS	75
NOEOL	128	OPERATORS IN EXPRESSIONS	74
NOEXECUTE.....	38	OR	97
NOEXTERNALS	93	OUTPUT CLEAR WINDOW	203

OUTPUT FILE NAME FUNCTION	358	LABEL	257
OUTPUT SAVE	204	LABELS	260
OUTPUT SPECIFICATION	129	LAST	255
OUTPUT WINDOW	201	MESSAGE	262
PQLFORMS COMMANDS	249	MESSAGES	252
PACK FUNCTION	358	NEXT	255
PAD FUNCTION	359	NOLABELS	260
PAGEDOWN	281	NOPROMPT	260
PAGELFN FUNCTION	359	PAGING	254
PAGENO	268	PQLFORMS	247
PAGENO FUNCTION	359	PREVIOUS	255
PAGES	268	PROMPT	257, 260
PAGESIZE	263, 267, 271	RESET	255
PAGEUP	281	ROWS	258
PAGEWID FUNCTION	359	SCREEN CO-ORDINATES	258
PARENTHESES	74, 97	UPDATING RECORDS	255
PATTERN FUNCTION	359	VALUE LABELS	260
PERFORM PROCS	46	WIDTH	259
PERIOD	19	WRITE	255
PFORMAT FUNCTION	359	PQLSERVER FUNCTIONS	369
PICTURE CLAUSE	125, 129	PRECEDENCE IN EXPRESSIONS	76
PICTURE FUNCTION	360	PREFIX, GET VARS	81
POPUP HELP	238	PREPARE STATEMENT	194
POSTYPE	218	PRESET	23, 82
PQL CONNECT DATABASE	135	ARRAY	82
PQL CONNECT TABFILE	178	PREVIOUS	280
PQL DISCONNECT DATABASE	136	PREVIOUS CASE	145
PQL DISCONNECT TABFILE	179	PREVIOUS RECORD	156
PQL ESCAPE	47	PREVIOUS ROW	184
PQL EXIT DBMS	48	PRINT FILENAME	239
PQLFILE	263	PROCEDURE VARIABLES	93
AT	258	PROCESS CASES	146
CLEAR	255	PROCESS JOURNAL	163
COLUMNS	258	PROCESS RECORD	157
CO-ORDINATES	258	PROCESS ROWS	185
DATA	257	PROCFILE FUNCTION	361
DELETE	256	PROCNAME FUNCTION	361
EDITING FIELDS	254	PROCS	40
ERROR	252, 262	PROGRAM	37
ERROR MESSAGE	262	PROGRAM STATUS FUNCTION	376
EXIT	254	PROGRAM VARIABLES	50
FIELD ELEMENTS	257	PROGRAMMING ERRORS	393
FIRST	255	PROGRESS	40, 231
FONT	261	PROGRESS FUNCTION	361
GENERAL CLAUSES	257	PROMPT	276, 283
HELP	252	PROMPT TEXT	123

PUBLIC.....	40, 91	REMOVE DCONTROL	243
PUT LINE TO BUFFER	298	REMOVE ITEM	236
PUT VARS	84	RENAME FILE FUNCTION.....	367
QUOTATION MARKS	74	REPEAT SYMBOL	82
QUOTED STRINGS	74	MEMBERS	119
RACCESS FUNCTION	361	REPLACE.....	40, 91, 119, 128
RADIO	229, 234	REPLACE FUNCTION	366
RAND FUNCTION	362	REPLACING BLANKS.....	342
RANDOM NUMBER FUNCTION.....	362	REQUIRED FIELDS.....	269
RANF FUNCTION	362	REREAD	126
RANGES	71	RESET	280
RANGES INTEGER.....	22	RESTORE CIR	148
RB FORMAT	125	RESTORE REC	159
READ	119, 122, 268	RETRIEVAL.....	37
FILES.....	122	RETRY RECORD	159
READONLY	274	RETURN	49
REAL.....	22, 64	RETURN CODE FUNCTION.....	372
REAL4 FUNCTION	362	RETURNING	40, 45
RECDOC FUNCTION	362	SUBROUTINE	40
RECDOCN FUNCTION	362	REVERSE.....	147, 157, 164, 186
RECLEVEL	149	REVERSE FUNCTION	367
RECLEVEL FUNCTION.....	362	RKEYSIZE FUNCTION.....	367
RELOCK	40, 170	RND FUNCTION	367
RELOCK FUNCTION	363	RNMFILE FUNCTION	367
RENAME FUNCTION.....	363	ROUNDING FUNCTION	367
RECNUM FUNCTION.....	363	ROW IS	189
RECODE	86	ROW PROCESSING.....	185
RECORD BLOCKS	29	ROWCOUNT FUNCTION.....	367
RECORD COUNTER FUNCTION.....	334	RRECSEC FUNCTION	367
RECORD DOCUMENTATION.....	362	RT_ERROR	112
RECORD INDEXED FUNCTION.....	336	RUN TIME ERROR.....	112
RECORD IS	151	RVARSEC FUNCTION	368
RECORD LABEL	362	S PICTURE	125
RECORD NAME	151, 157	SAMPLE	147, 186
RECORD NUMBER.....	151, 157	SARG FUNCTION	368
RECSIZE FUNCTION	363	SAVE	40
ARRAY.....	54	SAVE FILE BOX	239
REDEFINE ARRAY	54	SBST FUNCTION	368
REGEXP FUNCTION.....	363	SCALED VARS	65
REGREP FUNCTION	366	SCHEMA FUNCTIONS	316
REGULAR EXPRESSIONS.....	363, 366	SCREEN.....	265
REJECT RECORD IF.....	269	SCREENS	285
OPERATORS	96	SCROLLAT FUNCTION.....	368
RELATIONAL OPERATORS	96	SCROLLTO FUNCTION.....	368
REMAINDER FUNCTION	327	SECURITY	178
REMOVE ALL	236	SEED	41, 147, 186

SEEK FUNCTION	368	SMISSING	82
SELECT	268	SORT	54
SELECT ALL	236	ARRAY	54
SELECT DCONTROL	243	SPIN	224
SELECT ITEM	236	SPREAD FUNCTION	373
SERADMIN FUNCTION	369	SPUT FUNCTION.....	373
SERADMIS FUNCTION	369	SQRT FUNCTION	373
SEREXEC FUNCTION	369	SQUARE ROOT FUNCTION.....	373
SERGET FUNCTION	370	SRCH FUNCTION	373
SERLINES FUNCTION	370	SREAD FUNCTION	374
SERLOG FUNCTION	370	SRST FUNCTION.....	374
SERNOOUT FUNCTION	370	STANDARD DEVIATION FUNCTION... 374,	375
SERSEND FUNCTION	370	START CASE.....	146
SERSENDB FUNCTION.....	370	STATEMENT	194
SERTEST FUNCTION.....	371	STATEMENT LABELS	100
SERVER.....	192	STATIC	41, 45
SERWRITE FUNCTION	371	STATTYPE FUNCTION	374
SESSION FUNCTIONS	313	STATUS LINE.....	200
SET.....	23, 89	STDEV FUNCTION	374
SET DIALOG TITLE.....	236	STDEVR FUNCTION	375
SET IMAGE.....	236	STDNAME FUNCTION.....	375
SET INI FILE TEXT	386	STDOUT WRITE	127
SET ITEM.....	236	STOP	103
SET ITEM FONT	236	STRING	22, 66
SETAKL FUNCTION	371	STRING EDIT FUNCTION	339
SETDFC FUNCTION	371	STRING FUNCTIONS	308
SETDIR FUNCTION	371	STRING LENGTH	56
SETPOS FUNCTION.....	371	STRING LENGTH FUNCTION.....	350
SETRANGE FUNCTION	371	STRING SEARCH FUNCTION.....	373
SETRC FUNCTION	372	STRING TO NUMBER FUNCTION	357
SETTING VARIABLES	72	STRINGS.....	74
SGET FUNCTION	372	SUBDIR FUNCTION.....	375
SGLOBAL FUNCTION	372	SUBPROCEDURE	112
SHOW ITEM.....	236	SUBROUTINE	37, 264
SHOWMISS	41	ARRAY.....	23
SIGN FUNCTION	372	SUBSCRIPT, ARRAYS.....	23
SIMPLE.....	93	ARRAY	53
SIMPLE VARIABLES	51	SUBSCRIPTS	23, 53
SIN FUNCTION	372	SUBSTR FUNCTION	375
SINE FUNCTION	372	SUBSTRING FUNCTION	368
SIRCGLHTM	127	SUBSTRING SEARCH FUNCTION	374
SIRUSER FUNCTION	372	SUFFIX, GET VARS	81
SKIPPING	146	SUM FUNCTION.....	375
SLIDER.....	230	SUMFILE	41
SMAX FUNCTION	373	SUMMARY VARIABLES.....	50
SMIN FUNCTION	373		

SUMR FUNCTION	375	TIME FORMAT	22
SVVAL FUNCTION	376	TIME FUNCTION	383
SWAP ITEM	236	TIME FUNCTIONS	306, 334
SYSTEM FUNCTION	376	TIME VARIABLES	67
SYSTEM MISSING VALUES	73	TIMEC FUNCTION	384
SYSTEM()	139, 149	TIMEMAP FUNCTION	384
T POSITIONAL FORMAT	124, 131	TIMER	235
TABFILE	41	TIMESTAMP FUNCTION	336, 385
TABFILE CONNECT	178	TIP BOX	238
TABINDN FUNCTION	379	TITLE	268
TABINDS FUNCTION	379	TO LISTS	52
TABINDT FUNCTION	379	TODAY FUNCTION	384
TABINDU FUNCTION	380	TREE	233
TABINDV FUNCTION	380	FUNCTIONS	302
TABLE FUNCTIONS	379	TRIGONOMETRIC FUNCTIONS	301
TABNAME FUNCTION	380	TRIM FUNCTION	384
TABRECS FUNCTION	380	TRIML FUNCTION	384
TABVARS FUNCTION	380	TRIMLR FUNCTION	384
TABVINFN FUNCTION	380	TRIMR FUNCTION	385
TABVINFS FUNCTION	381	TRUNC FUNCTION	385
TABVNAME FUNCTION	381	TRUNCATION FUNCTION	327, 385
TABVRANG FUNCTION	381	TSTODT FUNCTION	385
TABVTYPE FUNCTION	381	TSTOTM FUNCTION	385
TABVVALI FUNCTION	381	TUPDATE	41, 186, 189
TABVVLAB FUNCTION	382	TWRITE FUNCTION	385
TABVVVAL FUNCTION	382	TYPE	274
TAN FUNCTION	382	TYPE OF DATABASE FUNCTION	337
TANGENT FUNCTION	382	U PICTURE	125
TANH FUNCTION	382	UNCHECK ITEM	236
TBARITEM	208	UNCHECK MENUITEM	213
TBARSEP	209	UNDEFINED	73, 87
TEXT	232, 234	UNIQUE INDEX FUNCTION	336
TEXT BOX	238	UNTIL	109, 158, 187
TFACCESS FUNCTION	382	UPDATE	41, 264, 268
TFATTR FUNCTION	382	UPDATE LEVEL	164
TFCOUNT FUNCTION	382	UPDATE LEVEL FUNCTION	385
TFFILE FUNCTION	383	UPDATE LOG	42
TFGRNAME FUNCTION	383	UPDLEVEL FUNCTION	385
TFGRPW FUNCTION	383	UPGET FUNCTION	385
TFJNNAME FUNCTION	383	UPPER FUNCTION	385
TFNAME FUNCTION	383	UPSET FUNCTION	386
TFTABS FUNCTION	383	UPSTAT	42
TFUSNAME FUNCTION	383	VALID RANGES	71
TFUSPW FUNCTION	383	VALID VALUES	68
THRU	86, 158, 187	VALIDATE FUNCTION	386
TIME	22, 67, 124, 131	VALLAB	69

VALLAB FUNCTION	386	VFORMAT FUNCTION.....	390
VALLABSC FUNCTION	386	VIA.....	158, 187
VALLABSN FUNCTION	387	VTYPE FUNCTION	390
VALLABSP FUNCTION.....	387	WACCESS FUNCTION	391
VALLABSV FUNCTION	387	WAIT	110
VALUE EXPRESSIONS	74	ARRAY	39
VALUE LABEL FUNCTION	386	WHILE	111
VALUE LABELS	69	WIDTH	277
VAR LABEL	70	WINCNT FUNCTION.....	391
VAR RANGES	71	WINDOW	198
VARDOCN FUNCTION	388	WINDOW CLEAR	203
VARGET FUNCTION	388	WINDOW OUTPUT	201
VARIABLE CONVERSION	86	WINDOW SAVE	204
VARIABLE LABEL FUNCTION	388	WINDOW STATUS	200
VARIABLE LISTS	52	WINDOW TITLE.....	199
VARIABLE REFERENCES	77	WINLIN FUNCTION.....	391
VARIABLE TYPE FUNCTION.....	390	WINMOVE FUNCTION	391
VARIABLES.....	22, 50	WINPOS FUNCTION	391
VARIABLES IN INDEX FUNCTION.....	336	WINSELL FUNCTION	392
VARLAB	70	WINSELP FUNCTION	392
VARLAB FUNCTION	388	WITH.....	158, 187
VARLABSC FUNCTION.....	388	WRECSEC FUNCTION.....	392
VARLENG FUNCTION	389	WRITE	119, 127, 269, 280
VARMAP	42, 50, 91	WVARSEC FUNCTION.....	392
VARNAME FUNCTION	389	X POSITIONAL FORMAT	124, 131
VARNAMEC FUNCTION.....	389	XOR	97
VARPOSIT FUNCTION.....	389	YES/NO BOX.....	238
VARPUT FUNCTION	389	YES/NO/CANCEL BOX	238
VARS.....	285	YESNO FUNCTION	392
VARTYPE FUNCTION	390		