BOS DIGITEC

tcVISION 7

Scripting Language

Technical Documentation

Last Review: September 25, 2023
tcScript7 Code Rev. 1595, tcREXX Code Rev. 487

1 Introduction....5
2 Instructions....6
  2.1 The ADDRESS Instruction....6
  2.2 ARG Instruction....9
  2.3 CALL Instruction....10
  2.4 DO/END Instruction....10
  2.5 DROP Instruction....11
  2.6 EXIT Instruction....11
  2.7 IF/THEN/ELSE Instruction....12
  2.8 ITERATE Instruction....12
  2.9 LEAVE Instruction....12
  2.10 NOP Instruction....13
  2.11 NUMERIC Instruction....13
  2.12 PARSE Instruction....13
  2.13 PROCEDURE Instruction....14
  2.14 RETURN Instruction....15
  2.15 SAY Instruction....15
  2.16 SELECT/WHEN/OTHERWISE Instruction....15
  2.17 UPPER Instruction....16
3 Operators....17
  3.1 Arithmetic Operators....17
  3.2 Assignment Operators....17
  3.3 Comparative Operators....17
  3.4 Concatenation Operators....18
  3.5 Logical Operators....18
4 Built-in Functions....19
  4.1 General....19
  4.2 Precision and Normalization....19
  4.3 Standard Parameter Names....19
  4.4 Possible System Dependencies....20
  4.5 Blanks vs. Spaces....21
  4.6 A2E( )....22
  4.7 ABBREV( )....22
  4.8 ABS( )....22
  4.9 ADDRESS( )....22
  4.10 AGENTCHECK()....23
  4.11 ARG( )....23
  4.12 B2C( )....24
  4.13 B2X( )....24
  4.14 BITAND( )....25
  4.15 BITCHG( )....25
  4.16 BITCLR( )....26
  4.17 BITCOMP( )....26
  4.18 BITOR( )....26
  4.19 BITSET( )....27
  4.20 BITTST( )....27
  4.21 BITXOR( )....27
  4.22 C2B( )....27
  4.23 C2D( )....28
  4.24 C2X( )....28
  4.25 CALLADA( )....29
  4.26 CALLDB( )....30
  4.27 CALLDB2CMD( )....36
  4.28 CENTER( ) CENTRE( )....36
  4.29 CHANGESTR( )....37

4.30 CHARIN( ) 37
4.31 CHAROUT( ) 38
4.32 CHARS( ) 38
4.33 CMD( ) 38
4.34 COMPARE( ) 39
4.35 CONDITION( ) 39
4.36 COPIES( ) 39
4.37 COUNTSTR( ) 40
4.38 CP_TRANSLATE( ) 40
4.39 D2C( ) 40
4.40 D2X( ) 41
4.41 DATATYPE( ) 41
4.42 DATE( ) 42
4.43 DELSTR( ) 44
4.44 DELWORD( ) 44
4.45 DIGITS( ) 45
4.46 E1141_A1252( ) 45
4.47 E2A( ) 45
4.48 ENCEDC 45
4.49 FILEGET( ) 46
4.50 FILEPUT( ) 46
4.51 FIND( ) 47
4.52 FORM( ) 47
4.53 FORMAT( ) 47
4.54 GETPID() 48
4.55 IMPORT() 48
4.56 INDEX( ) 49
4.57 INSERT( ) 49
4.58 JOBID() 49
4.59 JOBNAME() 50
4.60 JUSTIFY( ) 50
4.61 LASTPOS( ) 50
4.62 LEFT( ) 51
4.63 LENGTH( ) 51
4.64 LINEIN( ) 51
4.65 LINEOUT( ) 52
4.66 LINES() 53
4.67 LOWER( ) 53
4.68 LS_NOTE( ) 53
4.69 MAX( ) 54
4.70 MIN() 54
4.71 OVERLAY( ) 54
4.72 POS( ) 55
4.73 PROCESSCANCEL( ) 55
4.74 PROCESSINFO() 56
4.75 PROCESSINSTANCE() 57
4.76 PROCESSLIST() 57
4.77 PROCESSSTART() 58
4.78 PROCESSSTOP() 58
4.79 RANDOM( ) 59
4.80 REVERSE( ) 60
4.81 RIGHT( ) 60
4.82 SCRIPTCANCEL( ) 60
4.83 SCRIPTINFO() 61
4.84 SCRIPTINSTANCE() 62

4.85 SCRIPTLIST() 62
4.86 SCRIPTSTART() 64
4.87 SCRIPTSTOP() 64
4.88 SIGN( ) 64
4.89 SLEEP( ) 65
4.90 SOCKETCALL( ) 65
4.91 SOURCELINE( ) 71
4.92 SPACE( ) 71
4.93 STORAGEINFO( ) 72
4.94 STREAM( ) 72
4.95 STRIP( ) 75
4.96 SUBSTR( ) 76
4.97 SUBWORD( ) 76
4.98 SYMBOL( ) 77
4.99 TIME( ) 77
4.100 TRANSLATE( ) 78
4.101 TRIM( ) 79
4.102 TRUNC( ) 79
4.103 TSTP( ) 80
4.104 TSTPDIFF( ) 80
4.105 TSTPUTC( ) 80
4.106 UNLINK( ) 80
4.107 UPPER( ) 80
4.108 UTF8A2E( ) 81
4.109 UTF8CHAR( ) 81
4.110 UTF8LEFT( ) 81
4.111 UTF8LENGTH( ) 82
4.112 UTF8RIGHT( ) 82
4.113 UTF8SUBSTR( ) 82
4.114 VALUE( ) 83
4.115 VERIFY( ) 83
4.116 WORD( ) 84
4.117 WORDINDEX( ) 84
4.118 WORDLENGTH( ) 85
4.119 WORDPOS( ) 85
4.120 WORDS( ) 85
4.121 WTO( ) 86
4.122 X2B( ) 86
4.123 X2C( ) 86
4.124 X2D( ) 86
4.125 XRANGE( ) 87
4.126 XWTO( ) 87
5 Copyright 89

1 Introduction

This manual provides an overview of instructions, variables, and functions used in the tcVISION script language. The syntax of the language is based on the industry standard REXX. The intention of this manual is **not** to give an overview about the language, the syntax, and structure. This manual will only discuss functions that are relevant to the usage with tcVISION (tcSCRIPT).

It is important to understand that the tcSCRIPT functionality will be provided under z/OS, Windows, and all supported UNIX derivates. Instructions and functions discussed in this document may have different implications under the different environments or may not even be supported at all.

The description of parameters and variables required for an implementation of tcSCRIPT is available in manual “tcVISION – tcSCRIPT – Technical Documentation”.

2 Instructions

2.1 The ADDRESS Instruction

```
ADDRESS [ environment [ command ] [ redirection ] ] ;
[ [ VALUE ] expression [ redirection ] ] ;

and redirection has one of the forms:
WITH INPUT in_redir [ OUTPUT out_redir ] [ ERROR out_redir] ;
WITH INPUT in_redir [ ERROR out_redir ] [ OUTPUT out_redir] ;
WITH OUTPUT out_redir [ INPUT in_redir ] [ ERROR out_redir] ;
WITH OUTPUT out_redir [ ERROR out_redir ] [ INPUT in_redir] ;
WITH ERROR out_redir [ INPUT in_redir ] [ OUTPUT out_redir] ;
WITH ERROR out_redir [ OUTPUT out_redir ] [ INPUT in_redir] ;
in_redir is defined as:
NORMAL ;
[ STREAM | STEM ] symbol ;
and out_redir is defined as:
NORMAL ;
[ APPEND | REPLACE ] [ STREAM | STEM ] symbol ;
```

The ADDRESS instruction monitors in which external environment commands can be sent. If both environment and command are specified, the given command will be executed in the given environment. The effect is the same as issuing an expression to be executed as a command, except that the environment in which it is to be executed can be explicitly specified in the ADDRESS clause. In this case, the special variable RC will be set as usual, and the ERROR or FAILURE conditions might be raised as for normal commands.

In other words, all normal commands are ADDRESS statements with a suppressed keyword and environment.

The environment term must be a symbol or a string. If it is a symbol, its "name" is used, i.e. no variable value is exchanged. The command and expression terms can be any REXX expression.

Examples:
```
SYSTEM='PATH'
ADDRESS SYSTEM "echo Hello World"
```
is equivalent to a plain

```
ADDRESS SYSTEM "echo Hello World"
or
ADDRESS "SYSTEM" "echo Hello World"
```
for the external command 'Echo'.

A symbol specified as an environment name is not case-sensitive, whereas a string must match the case. Built-in environments are always uppercase.

The second syntax form of ADDRESS is a special case of the first form with command omitted. If the first token after ADDRESS is VALUE, the rest of the clause is taken to be an expression naming the environment that is to be made the current environment. Using VALUE makes it possible to circumvent the restriction that the name of the new environment must be a symbol or literal string.

However, it is not possible to combine both VALUE and command in a single clause.

Examples of the ADDRESS instruction:

```
ADDRESS system
ADDRESS SYSTEM 'copy' inputfile outputfile
ADDRESS VALUE newenvironment
ADDRESS (oldenvironment)
```

The first of these sets SYSTEM as the current environment (automatically changed to uppercase characters).
The second performs the command ''copy'' in the environment SYSTEM using the values of the symbols *inputfile* and *outputfile* as parameters. Note that this will not set SYSTEM as the current environment.
The third example sets the content of *newenvironment* as the current environment.
The fourth example is equivalent to the third example, but is not allowed by ANSI. Again, this kind of ADDRESS statement style should be avoided and the VALUE version should be used instead.

Example: The VALUE sub-keyword

```
ADDRESS NEWENVIRONMENT
ADDRESS VALUE NEWENVIRONMENT
```

The first of these examples sets the current default environment to NEWENVIRONMENT, while the second sets it to the value of the symbol NEWENVIRONMENT.

Only one type of environment is supported:

```
SYSTEM
```

This is the default environment that is selected at startup. The standard operating system command line interpreter will be loaded to execute the commands. It is also possible to execute so-called Shells or any other program that the command line interpreter can find and load.

ADDRESS Redirections

Some examples show the usage of ADDRESS redirections.

```
ADDRESS SYSTEM "sort" WITH INPUT STEM values. OUTPUT STEM values.
ADDRESS SYSTEM "myprogram" WITH INPUT STEM someinput OUTPUT STREAM
program.out ERROR STEM errors.
```

The first example executes the command 'sort'. The input for the command is read from the stem values (note the trailing period) and the output is sent back to the same stem after the command terminates.

A program called 'myprogram' is called in the second example. The input is fetched from the stem *someinput* and the standard output of the program is redirected to the output stream PROGRAM.OUT (note it is uppercase using standard REXX rules). Any generated error messages via the standard error stream are redirected to the stem *errors*.

Note: For PROGRAM.OUT, a symbol instead of character strings must be used.

The powerful possibilities of the redirection command are obvious. The disadvantage is the loss of a direct overview of what happens after a permanent redirection command has been executed.

The following is an overview of all redirection rules:

- Every environment has its own default redirection set.
- Every redirection set consists of three independent redirection streams, standard input (INPUT), standard output (OUTPUT), and standard error (ERROR). Users with some experiences with UNIX, DOS, and Windows know the redirection commands of the command line interpreter that can redirect each of the streams. This is nearly the same.
- Each redirection stream starts with the program-startup streams given to REXX when invoking the interpreter. These can be reset to the startup default by specifying the argument NORMAL for each redirection stream.
- The sequence of the redirection streams is irrelevant.
- Each stream can only be specified once per statement.
- Redirections can be intermixed. This means that both the OUTPUT and the ERROR redirection may point to the same "target". The data from the different channels will be put to the assigned "target" as they arrive.
- Redirections from and to the same source/destination try to keep the data consistent. If the INPUT/OUTPUT pair or the INPUT/ERROR pair points to the same destination, the content of the input or output channel is buffered so that writing to the output will not overwrite the input.
- All redirection streams are referenced by name (e.g. INPUT), a redirection processor (e.g. STREAM), and a destination symbol (e.g. OUT_FILE) following the rules to the redirection processor. This means that a period (.) after a symbol name for a stem must be specified or any symbol for the rest of the processors, in which case the content of the symbol is used as for normal variables.
- Both OUTPUT and ERROR streams can replace or append the data to the destination. Simply append either APPEND or REPLACE immediately after the OUTPUT or ERROR keywords. REPLACE is the default.
- The destination is checked or cleared prior to the execution of the command.
- ANSI defines two redirection processors: STEM and STREAM.
- The processor STEM uses the content of the symbol *destination.0* to access the count of the currently accessible lines and d*estination* is the given destination name. *Destination.0* must be filled with a whole, non-negative number in terms of the DATATYPE built-in function. Each of n lines can be addressed by appending the whole numbers one to n to the stem. Example: `STEM data`. is given, *DATA.0* contains 3. This indicates three content lines. They are the contents of the symbols *DATA.1* and *DATA.2* and *DATA.3*.
- The processor STREAM uses the content of the symbol destination to use a stream as known in the STREAM built-in function. The usage is nearly equivalent to the commands

LINEIN or LINEOUT for accessing the contents of the file. An empty variable (content set to the empty string) as the content of the destination is allowed and indicates the default input, output, or error streams given to the REXX program. This is equivalent to the NORMAL keyword.

- On INPUT all the data in the input stream is read up to either the end of the input data or until the called process terminates. The latter one may be determined after feeding up the input stream of the called process with unused data. Thus, there is no way to say whether data is used or not. This is not a problem with STEMs. All file-related sequential access objects may have lost data between two calls. Imagine an input file (STREAM) with three lines:

```
First line
DELIMITER
Second line
```

  Furthermore two processes p1 and p2 called WITH INPUT STREAM file with (file containing the three lines above). p1 reads lines up until a line containing DELIMITER and p2 processes the rest. It is very likely that the second process will not fetch any line because the stream may be processed by REXX and REXX may have put one or more lines ahead into the feeder pipe to the process. This might or might not happen. It all depends on the implementation.

  In short: INPUT may or may not use the entire input.

- Both OUTPUT and ERROR objects are checked for being properly set up just before the command starts. REPLACE is implemented as a deletion just before the command starts. Note that ANSI does not force STEM lines to be dropped in case of a replacement. A big stem with thousands of lines will still exist after a replacement operation if the called command does not produce any output. Just *destination.0* is set to 0.

2.2 ARG Instruction

```
ARG [ template ] ;
```

The ARG instruction parses the argument strings passed to a program or an internal routine and assigns them to variables. Parsing will be performed in uppercase mode. This clause is equivalent to:

`PARSE UPPER ARG [` *template* `] ;`

For more information, see the PARSE instruction. Note that this is the only situation where a multi string template is relevant.

The similarity between ARG and PARSE UPPER ARG has one exception. Suppose the PARSE UPPER ARG has an absolute positional pattern as the first element in the template, like:

```
pos = 5
parse upper arg =(pos) var
```

This is not equivalent to the ARG instruction (`arg = (pos) var`) because the ARG instruction would become an assignment. A simple trick to avoid this problem is just to put a placeholder period (.) in front of the pattern, thus the equal sign (=) is no longer the second token in the new ARG instruction.

2.3 CALL Instruction

```
CALL = routine [ parameter ][, [ parameter ] ... ] ;
or
CALL { ON | OFF } condition [ NAME label ] ;
condition is defined as
[ ERROR | FAILURE | HALT | NOTREADY ] ;
```

The CALL instruction invokes a subroutine, named by *routin*e, which can be internal, built-in, or external. The three function repositories are searched in that order for *routine*. The token *routine* is either a literal string or a symbol.
If *routine* is a literal string, the pool of internal subroutines is not searched.
Each *parameter* is evaluated from left to right, and passed to the subroutine as an argument. A *parameter* might be left out (e.g. an empty argument), which is not the same as passing a null string as argument.

Note:
For the CALL instruction the interference with line continuation must be watched. If there are trailing commas, it might be interpreted as line continuation. If a CALL instruction uses line continuation between two parameters, two commas are needed: one to separate the parameters and one to denote line continuation.

The second use of CALL offers the possibility to catch different conditions. OFF deactivates the catching of the specified condition, ON activates it. If the key word NAME and a *label* has been specified for ON, the routine specified by *label* (internal or built-in) is called when the corresponding *condition* is valid.

2.4 DO/END Instruction

```
DO [ repetitor ] [ conditional ] ;
[ clauses ]
END [ symbol ] ;
repetitor is defined as:
 symbol = expri [ TO exprt ] [ BY exprb ] [ FOR exprf ]
or
 exprr
or
 FOREVER
conditional is defined as:
{ WHILE exprw | UNTIL expru }
```

The DO/END instruction is the instruction used for looping and grouping several statements into one block. This is a multi-clause instruction.

The simplest case is when there is no *repetitor* or *conditiona*l, in which case it works like BEGIN/END in Pascal or {...} in C, meaning it groups one or more tcSCRIPT clauses into one conceptual statement.
The *repetitor* sub-clause controls the control variable of the loop or the number of repetitions. The *exprr* sub-clause may specify a certain number of repetitions or FOREVER may be used to go on looping forever.
If the control variable *symbo*l is specified, it will get the initial value *expri* at the start of the loop. At the start of each iteration including the first, it will be checked whether it has reached the value specified by *exprt*. At the end of each iteration the value *exprb* is added to the control variable. The loop will terminate after *exprf* iterations at most. All these expressions are evaluated only once before the loop is entered for the first iteration.
UNTIL or WHILE may also be specified which take a Boolean expression. WHILE is checked before each iteration, immediately after the maximum number of iterations has been performed. UNTIL is checked after each iteration, immediately before the control variable is incremented. It is not possible to specify both UNTIL and WHILE in the same DO instruction.
The FOREVER keyword is only needed when there is no *conditional*. Actually, it has the same effect as DO WHILE.
The sub-clauses TO, BY, and FOR may come in any order and their expressions are evaluated in the order in which they occur. However, the initial assignment must always come first.

2.5 DROP Instruction

`DROP` *symbol* `[` *symbol* `... ] ;`

The DROP instruction uninitializes the named *variabl*es, i.e. recreates the same state they were in at the startup of the program. The list of variable names is processed strictly from left to right. The sequence can be important if one variable is “tail” of another. In the following example, the two DROP instructions are not equivalent:

```
tail = 'a'
drop tail head.tail /* drops 'TAILBAR' and 'HEAD.TAIL' */
tail = 'a'
drop head.tail tail /* drops 'HEAD.a' and 'TAIL'
```

2.6 EXIT Instruction

`EXIT [` *expr* `] ;`

Terminates the tcSCRIPT program and optionally returns the expression *expr* to the caller. If specified, *expr* must be a positive integer. If *expr* is omitted, nothing will be returned to the caller. The EXIT instruction behaves differently in a "program" than in an external subroutine. In a "program" it returns control to the caller e.g. the operating system command interpreter. While for an external routine it returns control to the calling script, independent of the level of nesting inside the external routine being terminated.

2.7 IF/THEN/ELSE Instruction

```
IF expr [;] THEN [;] statement
[ ELSE [;] statement ]
```

This is a simple if-construct. First, the expression *expr* is evaluated and its value must be either 0 or 1 (*true* or *false*). Then, the statement following either THEN or ELSE is executed depending on whether *expr* was 1 or 0.
There must be a statement after THEN and ELSE. It is not allowed to just put a null-clause (e.g. a comment or a label) there. If the THEN or ELSE part is empty, use the NOP instruction. Also note that you cannot put more than one statement directly after THEN or ELSE. They must be packaged in a DO-END pair to make them a single, conceptual statement.
After THEN, after ELSE, and before THEN one or more clause delimiters (new lines or semicolons) can be used, but these are not required. The ELSE part is not required either, in which case no code is executed if *expr* is false (evaluates to 0). Note that there must also be a statement separator before ELSE because the statement must be terminated. This also applies to the statement after ELSE.

2.8 ITERATE Instruction

```
ITERATE [ symbol ] ;
```

The ITERATE instruction will iterate the innermost active loop in which the ITERATE instruction is located. If *symbol* is specified, it will iterate the innermost active loop with *symbol* as control variable. The simple DO/END statement without a *repetitor* and *conditional* is not affected by ITERATE. All active multi clause structures (DO, SELECT, and IF) within the loop being iterated are terminated.
ITERATE immediately transfers control to the END statement of the affected loop so that the next (if any) iteration of the loop can be started. It only affects loops on the current procedural level. All actions normally associated with the end of an iteration are performed.

Note that *symbol* must be specified as a literal. Compound variables are not supported.

2.9 LEAVE Instruction

```
LEAVE [ symbol ] ;
```

This statement terminates the innermost active loop. If *symbol* is specified, it terminates the innermost active loop with *symbol* as control variable. The instruction is identical to ITERATE, except that LEAVE terminates the loop while ITERATE lets the loop start on the next normal iteration. No actions normally associated with the normal end of an iteration of a loop are performed for a LEAVE instruction.

2.10 NOP Instruction

```
NOP ;
```

The NOP instruction is the "no operation" statement. It does nothing. Actually that is not entirely true, because the NOP instruction is a "real" statement (and a placeholder), as opposed to null clauses.

2.11 NUMERIC Instruction

```
NUMERIC DIGITS [ expr ] ;
FORM [ SCIENTIFIC | ENGINEERING | [ VALUE ] expr ] ;
```

tcSCRIPT uses floating point arithmetic of arbitrary precision that operates on strings representing the numbers. The NUMERIC statement is used to control most aspects of arithmetic operations. It has two distinct forms: DIGITS and FORM; which one to choose is given by the second token in the instruction:

DIGITS
Is used to set the number of significant digits in arithmetic operations. The initial value is 9, which is also the default value if *expr* is not specified. Large values for DIGITS may slow down some arithmetic operations considerably. If specified, *expr* must be a positive integer.
FORM
Is used to set the form in which exponential numbers are written. It can be set to either SCIENTIFIC or ENGINEERING. SCIENTIFIC uses a mantissa in the range 1.000... to 9.999..., and an exponent that can be any integer. ENGINEERING uses a mantissa in the range 1.000... to 999.999..., and an exponent that is divisible by 3. The initial and default setting is SCIENTIFIC. Following the keyword FORM may be the keywords SCIENTIFIC and ENGINEERING or the keyword VALUE. In this case the rest of the statement is considered an expression which will evaluate to either SCIENTIFIC or ENGINEERING. However, if the first token of the expression following VALUE is neither a symbol nor literal string, the VALUE keyword can be omitted.
The setting of FORM never affects the decision about whether to choose exponential form or normal floating point form. It only affects the appearance of the exponential form once that form has been selected.

2.12 PARSE Instruction

```
PARSE [ UPPER ] type [ template ] ;
type is defined as:
{ ARG | LINEIN | SOURCE | VERSION  |
  VALUE [ expr ] WITH VAR symbol  }
```

The PARSE instruction takes one or more source strings and then parses them using the *template* for directions. The process of parsing is one where parts of a source string are extracted and stored in variables. Which parts exactly is determined by the patterns.

Which strings are to be the source of the parsing is defined by the *type* sub-clause that can be any of:

ARG
The data to use as the source during the parsing is the argument strings given at the invocation of this procedure level. Note that this is the only case where the source may consist of multiple strings.
LINEIN
Makes the PARSE instruction read a line from the standard input stream as if the LINEIN() built-in function had been called. It uses the contents of that line (after stripping off end-of-line characters if necessary) as the source for the parsing. This may raise the NOTREADY condition if problems occur during the read.
SOURCE
The source string for the parsing is a string containing information about the invocation of the program. This information will not change during the execution of a tcSCRIPT script. The format of the string is:
system invocation filename
The first space-separated word *(system*) is a single word describing the platform on which the system is running. This is often the name of the operating system. The second word describes how the script was invoked.
VALUE expr WITH
This form evaluates *expr* and uses the result of that evaluation as the source string to be parsed. The token WITH may not occur inside *expr*, because it is a reserved keyword in this context.
VAR symbol
This form uses the current value of the named variable *symbol* (after tail-substitution) as the source string to be parsed. The variable may be any variable symbol. If the variable is uninitialized, a NOTREADY condition will be raised.
VERSION.
This form resembles SOURCE. It will display information about tcSCRIPT.

The following example shows a typical output:

```
PARSE VERSION info
SAY info

tcSCRIPT 5.0.1161 (STR), 32-bit, tcREXX 1.1.266
```

2.13 PROCEDURE Instruction

```
PROCEDURE [ EXPOSE [ varref [ varref ... ] ] ] ;
varref is defined as
{ symbol | ( symbol ) }
```

The PROCEDURE instruction is used by tcSCRIPT subroutines in order to control how variables are shared among routines. The simplest use is without any parameters. In this case all future references to variables in that subroutine refer to local variables. If there is no PROCEDURE instruction in a subroutine, all variable references in that subroutine refer to variables in the calling routine's name space.

If the EXPOSE keyword is specified to any references to the variables in the list following EXPOSE, refer to local variables that can also be used by the calling program. Variables not defined in the EXPOSE list but used in the subroutine are not available to the calling program.

2.14 RETURN Instruction

```
RETURN [ expr ] ;
```

The RETURN instruction is used to terminate the current procedure level and return control to a level above. When RETURN is executed inside one or more nesting constructs, e.g. DO, IF, WHEN, or OTHERWISE, the nesting constructs are terminated too.
Optionally, an expression can be specified as an argument to the RETURN instruction, and the string resulting from evaluating this expression will be the return value from the procedure level terminated to the caller procedure level. Only a single value can be returned. When RETURN is executed with no argument, no return value is returned to the caller and an error is raised if the subroutine was invoked as a function.

2.15 SAY Instruction

```
SAY [ expr ]
```

The SAY instruction can be used to display any character string (including line feed) on the screen. If no argument is used, only a line feed is displayed.
expr can be anything that can be interpreted as a string (a character string, the result of a function, the content of a variable, etc.).

2.16 SELECT/WHEN/OTHERWISE Instruction

```
SELECT ; whenpart [ whenpart ... ] [ OTHERWISE [;]
[ statement ... ] ] END ;
whenpart is defined as:
WHEN expr [;] THEN [;] statement
```

This instruction is used for general purpose nested IF structures. Although it has certain similarities with CASE in Pascal and switch in C, it is very different from these in some respects. An example of the general use of the SELECT instruction is:

```
select
when expr1 then statement1
when expr2 then do
 statement2a
 statement2b
end
when expr3 then statement3
otherwise
 owstatement1
 owstatement2
end
```

When the SELECT instruction is executed, the next statement after the SELECT statement must be a WHEN statement. The expression immediately following the WHEN token is evaluated and must result in a valid Boolean value. If it is *true* (i.e. 1), the statement following the THEN token matching the WHEN is executed, and afterwards control is transferred to the instruction following the END token matching the SELECT instruction. If the expression of the first WHEN is *false* (i.e. 0), the next statement must be either another WHEN or an OTHERWISE statement. In

the first case, the process explained above is iterated. In the second case, the clauses following the OTHERWISE up to the END statement are interpreted.
By the nature of the SELECT instruction, the WHENs are tested in the sequence in which they occur in the source. If more than one WHEN has an expression that evaluates to true, the first one encountered is selected. If the programmer wants to associate more than one statement with a WHEN statement, a DO/END pair must be used to enclose the statements. Zero, one, or more statements may be put after the OTHERWISE without having to enclose them in a DO/END pair.
The clause delimiter is optional after OTHERWISE and before and after THEN.

2.17 UPPER Instruction

```
UPPER symbol [ symbol [ symbol [...] ] ] ;
```

The UPPER instruction is used to translate the contents of one or more variables to uppercase.
The variables are translated in sequence from left to right.
Each symbol is separated by one or more blanks.
Only simple and compound symbols can be specified. Specification of a stem variable results in an error.

3 Operators

An operator represents an operation to be carried out between two terms, such as a division. tcSCRIPT supports five types of operators:
Arithmetic, *Assignment*, *Comparative*, *Concatenation*, and *Logical* Operators.

Each is described in detail in the following sections.

3.1 Arithmetic Operators

Arithmetic operators can be applied to numeric constants and tcSCRIPT variables that evaluate to valid numbers. The following operators are listed in decreasing order of precedence:

-	Unary prefix. Same as 0 – number (example -3).
+	Unary prefix. Same as 0 + number (example +3).
******	Power
*****	Multiply
/	Divide
%	Integer divide. Divide and return the integer part of the division
//	Remainder divide. Divide and return the remainder of the division
+	Addition
-	Subtraction

3.2 Assignment Operators

Assignment operators are a means to change the value of a variable. tcSCRIPT only has one assignment operator.

=	Assign the value on the right side of the "=" to the variable on the left.

3.3 Comparative Operators

The comparative operators compare two terms and return the logical value **1** if the result of the comparison is *true*, or **0** if the result of the comparison is *false*. The comparative operators will ignore leading or trailing blanks for string comparisons and leading zeros for numeric comparisons. Numeric comparisons are made if both terms to be compared are valid numbers, otherwise a string comparison is done. String comparisons are case-sensitive, and the shorter of the two strings is padded with blanks.
The following comparative operators are supported for operations on fields with different field lengths (non-strict operators).

=	Equal
\=, ^=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<>, ><	Greater than or less than; same as not equal

The following comparative operators are supported for operations on fields with identical field lengths (strict operators).

==	Strictly equal
\==, ^==	Strictly not equal
>>	Strictly greater than
<<	Strictly less than
>>=	Strictly greater than or equal
<<=	Strictly less than or equal

3.4 Concatenation Operators

The concatenation operators combine two strings to form one by appending the second string to the right side of the first. The concatenation operators are:

(blank) Concatenation of strings with one space between them.

(abuttal) Concatenation of strings with no intervening space.

|| Concatenation of strings with no intervening space.

Examples:

```
var1 = 'Hello';var2 = 'World'
Say var1 var2 -> results in 'Hello World'
Say a || b -> results in 'HelloWorld'
Say var1'World' -> results in 'HelloWorld'
```

3.5 Logical Operators

Logical operators work with the strings 1 and 0, usually as a result of a comparative operator. These operators also only result in logical TRUE; 1 or logical FALSE; 0.

& AND returns 1 if both terms are 1.

```
(4 > 2) & (a = a)   /* true,  so result is 1  */
(2 > 4) & (a = a)   /* false, so result is 0  */
```

| INCLUSIVE OR returns 1 if either term is 1.

```
(4 > 2) | (5 = 3)   /* at least one is true, so result is 1 */
(2 > 4) | (5 = 3)   /* neither one is true,  so result is 0 */
```

&& EXCLUSIVE OR returns 1 if either term is 1 but NOT both terms.

```
(4 > 2) && (5 = 3)  /* only one is true,    so result is 1 */
(4 > 2) && (5 = 5)  /* both are true,       so result is 0 */
(2 > 4) && (5 = 3)  /* neither one is true, so result is 0 */
```

\ LOGICAL NOT reverses the result; 0 becomes 1 and 1 becomes 0.

```
\ 0                 /* opposite of 0,    so result is 1 */
\ (4 > 2)           /* opposite of true, so result is 0 */
```

4 Built-in Functions

4.1 General

Below follows an in-depth description of all the functions in the library of built-in functions (BIF).

BIFs are executed either as part of a CALL instruction or in an assignment statement. If the BIF is part of a CALL instruction, no parenthesis must be used. If the BIF is part of an assignment statement, the parenthesis must be used.

Certain functions can only be executed with CALL. This really depends on the environment (Operating System) in which tcSCRIPT is used.

If at all possible, this document will mention the dependencies.

This section is an introduction to the built-in functions. It describes common behavior, parameter conventions, concepts, and lists possible system-dependent parts.

The syntax of each one is listed. For each of the syntax diagrams, the parts written in *italic* names the parameters. Terms enclosed in [square brackets] denote optional elements. The `courier` font is used to denote that something should be written as is and also to mark output from the computer.

4.2 Precision and Normalization

The BIF library uses its own internal precision for whole numbers which may range from -999999999 to +999999999. For most functions neither parameters nor return values will be affected by any setting of NUMERIC. In the few cases where this does not apply it is explicitly stated in the description of the function. In general, only parameters that are required to be whole numbers are used in the internal precision, while numbers not required to be whole numbers are normalized according to the setting of NUMERIC before use. However, if a parameter is a numeric expression, that expression will be calculated and normalized under the settings of NUMERIC before it is given to the function as a parameter.

4.3 Standard Parameter Names

In the descriptions of the BIFs several generic names are used for parameters to indicate something about the type and use of that parameter, e.g. valid range. To avoid repeating the same information for the majority of the functions, some common "rules" for the standard parameter names are stated here. These rules implicitly apply for the rest of this chapter.

Note that the following list does not try to classify any general "datatypes" but provides a binding between the sub-datatypes of strings and the methodology used when naming parameters.

- *Length*
  is a non-negative whole number within the internal precision of the BIFs. Whether it denotes a length in characters or in words depends on the context.
- *String*
  can be any normal character string including the null string. There are no further requirements for this parameter. Sometimes a string is called a "packed string" to explicitly show that it usually contains more than the normal printable characters.
- *Option*
  is used in some of the functions to choose a particular action, e.g. in DATE() to set the format in which the date is returned. Everything except for the first character will be ignored and case does not matter. Thus, the string should not have any leading space.
- *Start*
  is a positive whole number and denotes a start position, e.g. in a string. Whether it refers to characters or words depends on the context. The first position is always numbered 1, unless explicitly stated otherwise in the documentation. When return values denote positions, the number 0 is generally used to denote a non-existent position.
- *Padchar*
  must be a string – exactly one character long. That character is used for padding.
- *Streamid*
  is a string that identifies a REXX stream. The actual contents and format of such a string depends on the implementation.
- *Number*
  is any valid number and will be normalized according to the settings of NUMERIC before it is used by the function.

Sometimes these names have a number appended. This is only to separate several parameters of the same type, e.g. *string*1, *string2,* etc. They still follow the rules listed above. There are several parameters in the BIFs that do not easily fall into the categories above. These are given other names and their type and functionality will be described together with the functions in which they occur.

4.4 Possible System Dependencies

Some of the functions in the BIF library are more or less system or implementation dependent. The functionality of these may vary, so be advised to use defensive programming and be prepared for any side effects they might have. These functions include:

- `ADDRESS()`
  is dependent on your operating system since there is no standard for naming environments.
- `BITAND()`, `BITOR()` and `BITXOR()`
  are dependent on the character set of your machine. In general, seemingly identical parameters will return very different results on ASCII and EBCDIC machines. Results will be identical if the parameter was given to these functions as a binary or hexadecimal literal.
- `CALLDB2CMD()` will only work on a mainframe.
- `C2X()`, `C2D()`, `D2C()`, and `X2C()`
  will be affected by the character set of your computer because they convert to or from characters. If **`C2X()`** and **`C2D()`** get their first parameter as a binary or hexadecimal literal, the result will be unaffected by the machine type. Also note that the functions **`B2X(), X2B(), X2D(), and D2X()`** are not affected by the character set because they do not use character representation.

- `CHARIN()`, `CHAROUT()`, `CHARS()`, `LINEIN()`, `LINEOUT()`, `LINES()`, and `STREAM()`
  are the interfaces to the file system. They might have system-dependent peculiarities.
- `CONDITION()`
  depends on the I/O system and execution of commands. Although the general operation of this function will be fairly equal among systems, the details may differ.
- `DATATYPE()` and `TRANSLATE()`
  If your implementation supports national character sets, the operation of these two functions will depend on the selected language.
- `DELWORD()`, `SUBWORD()`, `WORD()`, `WORDINDEX()`, `WORDLENGTH()`, `WORDPOS()`, and `WORDS()`
  requires the concept of a "word" which is defined as non-blank characters separated by blanks. However, the interpretation of what a blank character is depends on the implementation.
- `RANDOM()`
  will differ from machine to machine, because the algorithm is implementation-dependent. If you set the *seed*, you can safely assume that the same interpreter under the same operating system and on the same hardware platform will return a reproducible sequence. But if you change to another interpreter, another machine or even just another version of the operating system, the same *seed* might not give the same pseudo-random sequence.
- `TIME()`
  will differ somewhat on different machines, because it is dependent on the underlying operating system.
- `VALUE()`
  will be dependent on implementation and operating system if it is called with its third parameter specified.
- `XRANGE()`
  will return a string whose contents will be dependent on the character set used by your computer.

4.5 Blanks vs. Spaces

The description in this document differentiates between "blanks" and the <space> character. A blank is any character that might be used as "whitespace" to separate text into groups of characters. The <space> character is only one of several possible blanks. When this text says "blank", it means any character from a set of characters that is used to separate visual characters into words. When this text says <space>, it means one particular blank that is generally bound to the space bar on a normal computer keyboard.

The <space> character is treated as blank. However, additional characters might be interpreted as blanks. They are <tab> (horizontal tabulator), <ff> (formfeed), <vt> (vertical tabulator), <nl> (newline), and <cr> (carriage return). The interpretation of what is blank will vary between machines, operating systems, and interpreters. If support for national character sets is used, it will even depend on the selected language.

4.6 A2E( )

```
A2E(expression)
```

Converts the result of *expression* from ASCII to EBCDIC.

Example:

```
IF A2E(substr(FLDA,1,1)) = FIELD_EBCDIC
```

4.7 ABBREV( )

```
ABBREV(long,short[,length])
```

Returns 1 if the string *short* is strictly equal to the initial first part of the string *lon*g and otherwise returns 0. If *length* is unspecified, no minimum restrictions for the length of *short* apply, hence the null string is an abbreviation of any string.

This function is case-sensitive. Leading and trailing spaces are not stripped off before the two strings are compared.

Examples:

```
ABBREV('Testtext','Test')                 1
ABBREV('Testtext','Test',4)               0 /* Too short */
ABBREV('Testtext','test')                 0 /* Different case */
```

4.8 ABS( )

```
ABS(number)
```

Returns the absolute value of the *numbe*r, which can be any valid number. The result will be normalized according to the current setting of NUMERIC.

Examples:

```
ABS(-13)                                  13
ABS(123)                                  123
```

4.9 ADDRESS( )

```
ADDRESS()
```

Returns the current default environment to which commands are sent. The default environment is SYSTEM.

NOTE: ADDRESS() can only be used with a CALL instruction.

Examples:

```
CALL ADDRESS SYSTEM
```

4.10 AGENTCHECK()

```
AGENTCHECK( [waittime] )
```

Generates a string with the status of the connection to the agent. The following statuses are possible:

READY
The script has an error-free connection to the agent.
TERMINATE
The script has a healthy connection to the agent, but the Agent sent the script termination signal.
ERROR
The script has no connection to the agent.
UNKNOWN
The script does not know any agent and therefore cannot run a connection test.

If *waittime* is specified and is not 0, the *waittime* function waits seconds before returning to the caller, unless a state other than READY occurs first.

This function only makes sense in a control script. There you can react to the loss of the agent connection in 'CDC_ExitControl'.

Example:
```
CDC_ExitControl:

do forever
      /* make something meaningfull */
      ...
      /* Agent connection still OK ? */
      rc = AGENTCHECK(1)
      say 'AGENTCHECK:' rc
      if rc <> "READY" then leave
end

RETURN '0, No Error'
```

4.11 ARG( )

```
ARG([argno[,option]])
```

Returns information about the arguments of the current procedure level. For subroutines and functions, it will refer to the arguments with which they were called. For the "main" program, it will refer to the arguments used when the program was called.

A call without parameter returns the number of passed arguments.

argno must be a positive whole number. If only *argno* is specified, the argument specified will be returned. The first argument is numbered 1. If *argno* refers to an unspecified argument (either omitted or *argno* is greater than the number of arguments), a null string is returned.
If *option* is specified as well, the return value will be 1 or 0 depending on the value of *option* and whether the numbered parameter was specified or not. *option* can be:

[O] - (Omitted)
Returns 1 if the numbered argument was omitted or unspecified. Otherwise 0 is returned.

[E] - (Existing)
Returns 1 if the numbered argument was specified. Otherwise 0 is returned.

Examples:

```
CALL Testfunction 'This' 'is', 'a',, 'test',,
```

ARG produces the following results for “Testfunction”:

```
ARG()                   4           /* Last parameter omitted */
ARG(1)                  'This is'
ARG(2)                  'a'
ARG(3)                  ''
ARG(9)                  ''          /* Ninth parameter doesn't exist*/
ARG(2,'E')              1
ARG(2,'O')              0
ARG(3,'E')              0           /* Third parameter omitted */
ARG(9,'O')              1
```

4.12 B2C( )

```
B2C(binstring)
```

Converts a string of binary digits (0 or 1) into the corresponding character representation. The conversion is the same as though the argument string had been specified as a literal binary string (e.g. '1010'B). Blanks are permitted in the string, but only at byte boundaries. This function is particularly useful for creating strings that are to be used as bit masks.

Examples:

```
B2C('00110010')         '2'
B2C('01100010')         'b'
```

4.13 B2X( )

```
B2X(binstring)
```

The specified parameter is converted from a binary string into a hexadecimal string. *Binstring* can only contain the binary digits 0 and 1. To increase readability blanks may be included in

binstring to group the digits. Each group must have a multiple of four binary digits, except for the first group. If the number of binary digits in the first group is not a multiple of four, that group is padded at the left with up to three leading zeros to make it a multiple of four. Blanks can only occur between binary digits, not as leading or trailing characters.
Each group of four binary digits is translated into one hexadecimal digit in the output string. There will be no extra blanks in the result and the upper six hexadecimal values A through F are uppercase.

Examples:

```
B2X('0100 01110001 0001')          '4711'
B2X('11 1101 01111111')            '3D7F'
B2X('0010010 0011')                '123'
```

4.14 BITAND( )

```
BITAND(string1[,[string2][,padchar]])
```

Returns the result from byte wise applying the operator AND to the characters in the two strings *string1* and *string2*. Note that this is not the logical AND operation, but the bit wise AND operation. *String2* defaults to a null string. The two strings are left justified. The first characters in both strings will be AND'ed, then the second characters and so forth.
The *padchar* character defines the behavior of this function when the two strings do not have equal length. If it is undefined, the remaining part of the longer string is appended to the result after all characters in the shorter string have been processed. If *padchar* is defined, the shorter string is padded on the right up to the length of the longer string, using *padchar*.
When using this function on character strings, e.g. to uppercase or lowercase a string, the result depends on the character set used. To lowercase a string in EBCDIC, use BITAND() with a *padchar* value of x'bf'. To do the same in ASCII, use BITOR() with a *padchar* value of x'20'.

Examples:

```
BITAND('123456'x,'F0F0'x)          '103056'x
BITAND('hello world',,'df'x)       'HELLO WORLD' /* For ASCII */
BITAND('123456'x,'4321'x,'f0'x)    '022006'x
```

4.15 BITCHG( )

```
BITCHG(string,bit)
```

Changes the state of the specified *bit* in the argument *string*. Bit numbers are defined so that bit 0 is the low-order bit of the rightmost byte of the string.

Examples:

```
BITCHG('1111'x,4)                        '1101'x
BITCHG('1234'x,10)                       '1634'x
```

4.16 BITCLR( )

```
BITCLR(string,bit)
```

Clears (sets to zero) the specified *bit* in the argument *strin*g. Bit numbers are defined so that bit 0 is the low-order bit of the rightmost byte of the string.

Examples:

```
BITCLR('1111'x,4)                          '1101'x
BITCLR('1234'x,9)                          '1034'x
```

4.17 BITCOMP( )

```
BITCOMP(string1,string2,bit[,pad])
```

Compares the argument strings bit wise starting at bit number 0. The returned value is the bit number of the first bit in which the strings differ, or -1 if the strings are identical.

Examples:

```
BITCOMP('7F'x,'FF'x)               7
BITCOMP('FF'x,'FF'x)               -1
BITCOMP('FFF'x,'FF'x,'F'x)         -1
BITCOMP('11000011'b, '11010011'b,) 4
```

4.18 BITOR( )

```
BITOR(string1[,[string2][,padchar]])
```

Returns the result from byte wise applying the operator OR to the characters in the two strings *string1* and *strin*g2. Note that this is not the logical OR operation but the bit wise OR operation. *String2* defaults to a null string. The two strings are left-justified. The first characters in both strings will be OR'ed, then the second characters and so forth.
The *padchar* character defines the behavior of this function if the two strings are not of equal length. If it is undefined, the remaining part of the longer string is appended to the result after all characters in the shorter string have been processed. If *padchar* is defined, the shorter string is padded on the right up to the length of the longer string using *padcha*r.
When using this function on character strings, e.g. to uppercase or lowercase a string, the result depends on the character set used.

Examples:

```
BITOR('13'x)                       '13'x
BITOR('25'x,'12'x)                 '37'x
BITOR('25'x,'1234'x)               '3734'x
BITOR('25'x,'1234'x,'F0'x)         '37F4'x
BITOR('3333'x,,'5A'x)              '7B7B'x
BITOR('HeLlo',,'20'x)              'hello' /* ASCII */
```

4.19 BITSET( )

```
BITSET(string,bit)
```

Sets the specified *bit* in the argument *string* to 1. Bit numbers are defined so that bit 0 is the low-order bit of the rightmost byte of the string.

Examples:

```
BITSET('0000'x,2)                   '0004'x
BOTSET('1234'x,10)                  '1634'x
```

4.20 BITTST( )

```
BITTST(string,bit)
```

The Boolean return indicates the state of the specified bit in the argument string.
Bit numbers are defined so that bit 0 is the low-order bit of the rightmost byte to the string.

Examples:

```
BITTST('1234'x,3)                   0
BITTST('1234'x,4)                   1
```

4.21 BITXOR( )

```
BITXOR(string1[,[string2][,padchar]])
```

Works like BITAND(), except that the logical function XOR (exclusive OR) is used instead of AND. For more information see BITAND().

Examples:

```
BITXOR('123456'x,'F0F0'x)           'E2C456'x
BITXOR('Hello World',,'20'x)        'hELLO wORLD' /* For ASCII */
BITXOR('123456'x,'4321'x,'0f'x)     '511559'x
```

4.22 C2B( )

```
C2B(string)
```

Converts the supplied *string* into the equivalent string of binary digits.

Examples:

```
C2B('world')                  '0111011101101111011100100110110001100100'
```

4.23 C2D( )

```
C2D(string[,length])
```

Returns a whole number that is the decimal representation of the string *string*, interpreted as a binary number. If *length* (must be a non-negative whole number) is specified, it denotes the number of characters in *string* to be converted, and *string* is interpreted as a representation of a complement of two of a binary number consisting of the *length* right-justified characters in *string*. If *length* is not specified, *string* is interpreted as an unsigned number. A value of 0 for *length* gives a 0 as the function result. If *length* is larger than the length of *string*, *string* is extended on the left with '00'x.
If *length* is too short, only the *length* right-justified characters in *string* are considered. It should be noted that this will not only change the value of the number in general, but it might even change the sign.

Attention:
This function is highly dependent on the character set used by the computer.

If it is not possible to express the final result as a whole number under the current settings of NUMERIC DIGITS, an error is reported. The number to be returned will not be stored in the internal representation of the BIF library, so size restrictions on whole numbers generally applying to BIFs do not apply in this case.

Examples:

```
C2D('abc')                    6382179 /* For ASCII machines */
C2D('123'x)                   291
C2D('123'x,1)                 35
C2D('123'x,2)                 291
C2D('0123'x,3)                291
C2D('ffff'x,2)                -1
C2D('ffff'x)                  65535
C2D('ffff'x,3)                65535
C2D('fff9'x,2)                -16
C2D('ff80'x,2                 -128
C2D('ff80'x,0)                0
```

4.24 C2X( )

```
C2X(string)
```

Returns a string of hexadecimal digits representing the character string *string*. Converting is done byte wise, the hex values A through F are in uppercase, and there are no blank characters in the result. Leading zeros are not stripped off in the result. Note that the behavior of this function depends on the character set your computer is running (e.g. ASCII or EBCDIC).

Examples:

```
C2X('48CF'x)                  48CF
C2X('abc')                    616263 /* For ASCII machines */
C2X('101 0110 1110 'b)        056E'
```

4.25 CALLADA( )

```
CALLADA( function, InputStem, OutputStem)
```

This function can be used to access Adabas on mainframe systems (z/OS).

Function

This parameter is not used.

In the following table **'ADAI'** is assumed to the *InputStem* and **'ADAO'** is assumed to be the *OutputStem*.

Parameter	Input or Output	Description
ADAI.MODULENAME	I	Name of the Adabas Load Module
ADAI.SVC	I	Optional specification of Adabas SVC
ADAI.COMMAND	I	Command
ADAI_COMMAND_ID	I/O	Command ID
ADAI.RETURNCODE	I/O	Return Code
ADAI.DB_ID	E	DbId of selected database
ADAI.FILE_NR	I	File Nr of selected table
ADAI.ISN	I/O	Refer to Adabas documentation
ADAI.ISN_LOWER_LIMIT	I/O	Refer to Adabas documentation
ADAI.ISN_QUANTITY	I/O	Refer to Adabas documentation
ADAI.COMMANDOPT1	I	Refer to Adabas documentation
ADAI.COMMANDOPT2	I	Refer to Adabas documentation
ADAI.ADDITIONS1	I	Refer to Adabas documentation
ADAI.ADDITIONS2	I/O	Refer to Adabas documentation
ADAI.ADDITIONS3	I	Refer to Adabas documentation
ADAI.ADDITIONS4	I	Refer to Adabas documentation
ADAI.ADDITIONS5	I	Refer to Adabas documentation
ADAI.RB	I/O	Record Buffer
ADAI.FB	I/O	Format Buffer
ADAI.SB	I/O	Search Buffer
ADAI.VB	I/O	Value Buffer
ADAI.IB	I/O	ISN Buffer

Parameters indicated by **I** are processed via the *InputStem*. Parameters indicated by **O** are processed via the *OutputStem*.

Example:

```
ADAIN.MODULENAME  =     'TVSADALN'
ADAIN.COMMAND           =     'LF'
ADAIN.RB                =     'S'
ADAIN.DB_ID            =     1
ADAIN.FILE_NR          =     1

RC = CALLADA('CDM' ,ADAIN,ADAOUT)
SAY RC
SAY ADAOUT.RETURNCODE
SAY ADAOUT.COMMAND_ID
SAY ADAOUT.ISN
SAY ADAOUT.ISN_LOWER_LIMIT
SAY ADAOUT.QUANTITY
SAY ADAOUT.ADDITIONS2 C2X(ADAOUT.ADDITIONS2)
SAY 'RB LENGTH' C2X(SUBSTR(ADAIN.RB,1,16)
SAY ADAIN.RB
```

4.26 CALLDB( )

```
CALLDB(function, InputStem, OutputStem)
```

CALLDB is a function that can be used to access external data sources via connection types of ODBC, DRDA, DB2, EXASOL, or Oracle (OCI). Based on the *function* used, parameters are available as part of the *InputStem* and *OutputStem*.

Functions:

CONNECT	This function connects to a data source using one of the above connection types.
DISCONNECT	This function disconnects from one of the above connection types.
EXECUTE	This function executes a defined SQL statement and returns a result set as part of the OutputStem.
FETCH	This function fetches row by row from the result set.

Based on the connection type, different parameters are available for the *InputStem* and *OutputStem*.

An input stem of **'DBI'** and an output stem of **'DQ1'** is assumed in the tables below.

Connection type: ODBC
This connection type connects directly to an ODBC data source on a Windows or UNIX/Linux machine. UNIX/Linux ODBC is only possible if an ODBC Manager such as UNIXODBC is installed.

Parameter	Input or Output	Used with function	Meaning
`DBI.InterfaceType`	I	CONNECT	Specifies the connection type `DBI.InterfaceType='ODBC'`
`DBI.ConnectString`	I	CONNECT	Specifies the string that must be used to connect to the data source `DBI.Connectstring = 'DSN=… '` Additional information can be found in the manual 'tcVISION tcSCRIPT*.
`DBI.Token`	I	EXECUTE FETCH	Specifies the communication

		DISCONNECT	token that will be returned to the script after a successful CONNECT. Use this token to use the correct connection "channel" for functions following the CONNECT.
`DBI.MaxRecords`	I	EXECUTE	Limits the number of records in the result set `DBI.MaxRecords = 5`
`DBI.SQL_Command`	I	EXECUTE	Specifies the SQL statement that should be passed to ODBC `DBI.SQL_Command = 'SELECT … '`
`DBI.RequestId`	I	FETCH	Multiple requests can be executed for every connection. Specify the request id that you want to use. `DBI.RequestId = 1`
`DQ1.NativeErrorCode`	O	CONNECT EXECUTE DISCONNECT	Contains the code returned by the DBMS system accessed by the ODBC driver. Refer to the corresponding documentation of the DBMS. `SAY 'DBMS-Error' DQ1.NativeErrorCode`
`DQ1.State`	O	CONNECT EXECUTE DISCONNECT	Contains the status code. This is usually related to the SQL statement and may indicate that the statement contains errors. Refer to the corresponding DBMS documentation. `SAY 'SQL STATE' DQ1.State`
`DQ1.ErrorMessage`	O	CONNECT EXECUTE DISCONNECT	Contains possible error messages `SAY DQ1.ErrorMessage`
`DQ1.FieldCount`	O	EXECUTE	Contains the number of fields in the result set. Can be used as an index (I)
`DQ1.F.Name.I`	O	EXECUTE	Contains the name of the field(I)
`DQ1.F.Type.I`	O	EXECUTE	Contains the field type of field(I)
`DQ1.F.Null.I`	O	EXECUTE	Contains NULL if field(I) does not exist or may contain low values (X'00')
`DQ1.F.Length.I`	O	EXECUTE	Contains the length of field(I)
`DQ1.F.Scale.I`	O	EXECUTE	Contains the number of decimal digits of field(I)
`DQ1.V.Null.I`	O	FETCH	Contains NULL if field(I) does not exist
`DQ1.V.Value.I`	O	FETCH	Contains value of field(I)

Connection type: DRDA
This connection type connects directly to a Db2 z/OS, LUW database using the DRDA protocol. The Db2 may reside on any of the supported platforms.

Parameter	**Input or Output**	**Used with function**	**Meaning**
`DBI.InterfaceType`	I	CONNECT	Specifies the connection type `DBI.InterfaceType='DRDA' or`

			`DBI.InterfaceType='DB2'`
`DBI.ConnectString`	I	CONNECT	Specifies the string that must be used to connect to the data source. Refer to manual 'tcVISION tcSCRIPT' for detailed information. `DBI.Connectstring = 'HOST=… '`
`DBI.Token`	I	EXECUTE FETCH DISCONNECT	Specifies the communication token that will be returned to the script after a successful CONNECT. Use this token to use the correct connection "channel" for functions following the CONNECT.
`DBI.MaxRecords`	I	EXECUTE	Limits the number of records in the result set `DBI.MaxRecords = 5`
`DBI.SQL_Command`	I	EXECUTE	Specifies the SQL statement that should be passed to Db2 `DBI.SQL_Command = 'SELECT … '`
`DBI.RequestId`	I	**FETCH**	Multiple requests can be executed for every connection. Specify the request id that you want to use. `DBI.RequestId = 1`
`DQ1.NativeErrorCode`	O	CONNECT EXECUTE DISCONNECT	Contains the code returned by the Db2 system. Refer to the corresponding IBM documen-tation. `SAY 'DBMS-Error' DQ1.NativeErrorCode`
`DQ1.State`	O	CONNECT EXECUTE DISCONNECT	Contains the status code. This is normally related to the SQL statement and may indicate that the statement contains errors. Refer to the corresponding IBM documentation. `SAY 'SQL STATE' DQ1.State`
`DQ1.ErrorMessage`	O	CONNECT EXECUTE DISCONNECT	Contains possible error messages `SAY DQ1.ErrorMessage`
`DQ1.FieldCount`	O	EXECUTE	Contains the number of fields in the result set. Can be used as an index (I).
`DQ1.F.Name.I`	O	EXECUTE	Contains the name of the field(I)
`DQ1.F.Type.I`	O	EXECUTE	Contains the field type of field(I)
`DQ1.F.Null.I`	O	EXECUTE	Contains NULL if field(I) does not exist or may contain low values (X'00')
`DQ1.F.Length.I`	O	EXECUTE	Contains the length of field(I)
`DQ1.F.Scale.I`	O	EXECUTE	Contains the number of decimal digits of field(I)
`DQ1.V.Null.I`	O	FETCH	Contains NULL if field(I) does not exist
`DQ1.V.Value.I`	O	FETCH	Contains value of field(I)

Connection type: EXASOL

This connection type connects directly to a EXASOL database using the CLI interface.

Parameter	Input or Output	Used with function	Meaning
`DBI.InterfaceType`	I	CONNECT	Specifies the connection type `DBI.InterfaceType='EXASOL'`
`DBI.ConnectString`	I	CONNECT	Specifies the string that must be used to connect to the data source. Refer to manual 'tcVISION tcSCRIPT' for detailed information. `DBI.Connectstring = 'SERVER=… '`
`DBI.Token`	I	EXECUTE FETCH DISCONNECT	Specifies the communication token that will be returned to the script after a successful CONNECT. Use this token to use the correct connection "channel" for functions following the CONNECT.
`DBI.MaxRecords`	I	EXECUTE	Limits the number of records in the result set `DBI.MaxRecords = 5`
`DBI.SQL_Command`	I	EXECUTE	Specifies the SQL statement that should be passed to EXASOL `DBI.SQL_Command = 'SELECT … '`
`DBI.RequestId`	I	**FETCH**	Multiple requests can be executed for every connection. Specify the request id that you want to use. `DBI.RequestId = 1`
`DQ1.NativeErrorCode`	O	CONNECT EXECUTE DISCONNECT	Contains the code returned by the EXASOL system. Refer to the corresponding EXASOL documentation. `SAY 'DBMS-Error' DQ1.NativeErrorCode`
`DQ1.State`	O	CONNECT EXECUTE DISCONNECT	Contains the status code. Usually this is related to the SQL statement and may indicate that the statement contains errors. Refer to the corresponding EXASOL documentation. `SAY 'SQL STATE' DQ1.State`
`DQ1.ErrorMessage`	O	CONNECT EXECUTE DISCONNECT	Contains possible error messages `SAY DQ1.ErrorMessage`
`DQ1.FieldCount`	O	EXECUTE	Contains the number of fields in the result set. Can be used as an index (I).
`DQ1.F.Name.I`	O	EXECUTE	Contains the name of the field(I)
`DQ1.F.Type.I`	O	EXECUTE	Contains the field type of field(I)
`DQ1.F.Null.I`	O	EXECUTE	Contains NULL if field(I) does not exist or may contain low values (X'00')
`DQ1.F.Length.I`	O	EXECUTE	Contains the length of field(I)
`DQ1.F.Scale.I`	O	EXECUTE	Contains the number of decimal digits of field(I)

`DQ1.V.Null.I`	O	FETCH	Contains NULL if field(I) does not exists
`DQ1.V.Value.I`	O	FETCH	Contains value of field(I)

<u>Connection type: Oracle</u>
This connection type connects directly to an Oracle database using the OCI interface. Refer to manual "tcVISION Workstation Component" for additional information about OCI requirements.

Queries for the system view V$PARAMETER must be granted to the user who will start the connection (Select any Dictionary).

Parameter	**Input or Output**	**Used with function**	**Meaning**
`DBI.InterfaceType`	I	CONNECT	Specifies the connection type `DBI.InterfaceType='ORACLE'`
`DBI.ConnectString`	I	CONNECT	Specifies the string that must be used to connect to the data source. Refer to manual ‘tcVISION tcSCRIPT’ for detailed information. `DBI.Connectstring = 'HOST=… '`
`DBI.Token`	I	EXECUTE FETCH DISCONNECT	Specifies the communication token that will be returned to the script after a successful CONNECT. Use this token to use the correct connection "channel" for functions following the CONNECT.
`DBI.MaxRecords`	I	EXECUTE	Limits the number of records in the result set `DBI.MaxRecords = 5`
`DBI.SQL_Command`	I	EXECUTE	Specifies the SQL statements that should be passed to Oracle `DBI.SQL_Command = 'SELECT … '`
`DBI.RequestId`	I	**FETCH**	Multiple requests can be executed for every connection. Specify the request id that you want to use. `DBI.RequestId = 1`
`DQ1.ErrorMessage`	O	CONNECT EXECUTE DISCONNECT	Contains possible error messages `SAY DQ1.ErrorMessage`
`DQ1.FieldCount`	O	EXECUTE	Contains the number of fields in the result set. Can be used as an index (I)
`DQ1.F.Name.I`	O	EXECUTE	Contains the name of the field(I)
`DQ1.F.Type.I`	O	EXECUTE	Contains the field type of field(I)
`DQ1.F.Null.I`	O	EXECUTE	Contains NULL if field(I) does not exist or may contain low values (X'00')
`DQ1.F.Length.I`	O	EXECUTE	Contains the length of field(I)
`DQ1.F.Scale.I`	O	EXECUTE	Contains the number of decimal digits of field(I)
`DQ1.V.Null.I`	O	FETCH	Contains NULL if field(I) does not exist
`DQ1.V.Value.I`	O	FETCH	Contains value of field(I)

Return codes of the CALLDBF Function:

Return code	Description	Interface type	Function
10	`ODBC_DLL_NOT_FOUND`	ODBC	CONNECT
11	`ODBC_DLL_PROC_NOT_FOUND`	ODBC	CONNECT
12	`ODBC_ERROR`	ODBC	CONNECT EXECUTE FETCH DISCONNECT
13	`ODBC_UNKNOWN_FIELD_TYPE`	ODBC	EXECUTE
16	`DB_EOF`	ODBC ORACLE DRDA DB2 EXASOL	FETCH
18	INVALID_COMMAND	ODBC ORACLE DRDA DB2 EXASOL	neither CONNECT nor EXECUTE FETCH DISCONNECT
19	`INVALID_TOKEN`	ODBC ORACLE DRDA DB2 EXASOL	EXECUTE FETCH DISCONNECT
20	`OUT_OF_MEMORY`	ODBC ORACLE DRDA DB2 EXASOL	CONNECT EXECUTE FETCH DISCONNECT
23	`INVALID_INTERFACE_TYPE`	neither ODBC nor ORACLE DRDA DB2 EXASOL	CONNECT EXECUTE FETCH DISCONNECT
24	`INVALID_DATA_LENGTH`		EXECUTE FETCH DISCONNECT
28	`NO_DATA_AVAILABLE`	ODBC ORACLE DRDA DB2 EXASOL	FETCH
29	`INVALID_CODEPOINT_RECEIVED`	ODBC ORACLE DRDA DB2 EXASOL	internal error
31	`OCI_DLL_NOT_FOUND`	ORACLE	CONNECT
32	`OCI_DLL_PROC_NOT_FOUND`	ORACLE	CONNECT
33	`OCI_DB_VERSION_NOT_SUPPORTED`	ORACLE	CONNECT
34	`OCI_OCI_VERSION_LT_ORADB`	ORACLE	CONNECT
35	`OCI_OCI_ERROR`	ORACLE	CONNECT

			EXECUTE FETCH DISCONNECT

4.27 CALLDB2CMD( )

```
CALLDB2CMD( command, DB2 sub system name))
```

This function executes the command specified as *command* and passes it to the Db2 subsystem specified as second parameter.

The function is only available on z/OS mainframe systems.

The output lines of the command are passed to the script in variable DB2_Out.I. Variable DB2_OutCount contains the number of output lines and can be used as an index.

The function returns two return codes. These are the variables DB2_ReturnCode and DB2_ReasonCode.

Example:

```
  RC = CALLDB2CMD('-DISPLAY LOG','DB8G')
/* Check whether DB2 command has been successful  */
  IF RC <> 0
  THEN DO
    SAY 'Error during execution of DB2 command'
    SAY 'Returncode : ' DB2_ReturnCode
    SAY 'Reasoncode : ' DB2_ReasonCode
    RETURN 165
  END
 /* Extract RBA from Command output  */
  DO I = 1 TO DB2_OutCount
      fpos = INDEX(DB2_Out.I, 'H/O RBA = ')
    IF fpos > 0 THEN DO
        RBA = SUBSTR(DB2_Out.I, fpos + LENGTH('H/O RBA = '))
        SAY 'RBA to start with : ' RBA
    END
 END /* DO I = 1 TO DB2_OutCount
```

4.28 CENTER( ) CENTRE( )

```
CENTER(string, length [, padchar ] )
CENTRE(string, length [, padchar ] )
```

This function has two names to support both American and British spelling. The function centres *string* in the length of *length* characters. If *length* (must be a non-negative whole number) is greater than the length of *string*, *string* is padded with *padchar* or <space> if *padchar* is unspecified. If *length* is smaller than the length of *string*, characters will be removed. If possible, both ends of *string* receive (or lose) the same number of characters.
If an odd number of characters is to be added (or removed), one more character is added to (or removed from) the right end than the left end of *string*.

Examples:

```
CENTER('Hello',9)                '  Hello  '
CENTER('Hello',10)               '  Hello   '
CENTRE('Hello',3)                'ell'
CENTER('Hello',4)                'Hell'
CENTER('Hello',9,'*')            '**Hello**'
```

4.29 CHANGESTR( )

```
CHANGESTR(needle, haystack, newneedle )
```

The purpose of this function is to replace all occurrences of *needle* in the string *haystack* with *newneedl*e. The function returns the changed string.
If *haystack* does not contain *needl*e, the original *haystack* is returned.

Examples:

```
CHANGESTR('a','hello','x')                        'hello'
CHANGESTR('','','x')                              ''
CHANGESTR('e','hello','x')                        'hxllo'
CHANGESTR('0','0','1')                            '1'
CHANGESTR('a','','x')                             ''
CHANGESTR('','hello','world')                     'hello'
CHANGESTR('hello','helloworld','bye')             'byeworld'
CHANGESTR('hellow','hello','x')                   'hello'
CHANGESTR('hello','helloworldhello','x')          'xworldx'
```

4.30 CHARIN( )

```
CHARIN( [streamid] [, [start] [, count] ] )
```

CHARIN returns a string with up to *count* characters read from input stream *streamid*.
The default value for *streamid* is the standard input stream. Format and type of *streamid* depend on the environment. The default value for *count* is 1.
Parameter *start* (positive number) can be specified to define the current position in the file before the read operation is processed. If *start* is not specified, the current position is not changed before the read process. *Start* is only valid for persistent streams. If *start* is specified for transient streams, an error is issued. The first character in the stream has the number 1.
Whether an explicit OPEN must be performed before the read operation depends on the environment.
count specifies the number of characters that should be read. For a value of 0, no characters are read but the actual read position for the data is set and a null string is returned.

Examples:

```
CHARIN(streamid,1,3)          'Thi'  /* the first 3 characters           */
CHARIN(streamid,1,0)          ''     /* now at the beginning of the file */
CHARIN(streamid)              'T'    /* after the last call              */
CHARIN(streamid, ,2)          'hi'   /* after the last call              */
```

4.31 CHAROUT( )

```
CHAROUT( [streamid] [, [string] [, start] ] )
```

Writes the string specified by *string* into file *streamid* after positioning on the character specified by parameter *start*.
The number of remaining characters is returned. If *streamid is not specified,* the default output is used.
The parameter *start* (positive number) can be specified to define the current position in the file before the write operation is processed. If *start* is not specified, the current position is not changed before the write process. *Start* is only valid for persistent streams. If *start* is specified for transient streams, an error is issued. The first character in the stream has the number 1.
During the first OPEN of the stream the write position is at the end of the stream, hence CHAROUT appends the characters at the end of the stream.
If *start i*s specified but not *string*, positioning only takes place at the new position.
If neither *string* nor *start* have been specified, the stream will be closed.

Examples:

```
CHAROUT(streamid,'Bye')       /* Bye is added to the end of the stream   */
CHAROUT(streamid,'Hello',1)   /* writes Hello at the stream beginning    */
CHAROUT(streamid, ,6)         /* positions on character 6                */
CHAROUT(streamid)             /* closes the stream                       */
```

4.32 CHARS( )

```
CHARS( [streamid] )
```

Returns the number of characters that are still available in input stream *streamid*. For persistent streams this is the number of characters at the current read position. If *streamid* is not specified, the default input is assumed.
For some streams it is not possible to determine the number of remaining characters (example STDIN). For those streams, CHARS returns 1 to indicate that data is available or 0 if no data is available. In a Windows environment CHARS always returns 1.

Examples:

```
CHARS(streamid)               42 /* this may depend upon */
CHARS()                       1  /* the stream           */
```

4.33 CMD( )

```
CMD( command )
```

CMD can be used in a mainframe z/OS to issue the console command *command*.

Example:

```
rc = CMD('D A')
```

4.34 COMPARE( )

```
COMPARE(string1,string2[,padchar])
```

This function compares *string1* with *string2* and returns the position (an integer) of the first character that is different. If the two values are identical, 0 is returned. The comparison is case-sensitive, and leading and trailing spaces matter.
If the strings have different lengths, the shorter string is padded with the *padchar* fill character on the right side with the length of the longer string. This is done before the comparison. If no *padchar* is specified, <space> is used.

Examples:

```
COMPARE('HelloWorld','Helloworld')            6
COMPARE('HelloWorld','HelloWorld')            0
COMPARE('Hellooo','Hell')                     5
COMPARE('Hellooo','Hell','o' )                0
```

4.35 CONDITION( )

```
CONDITION( [option] )
```

Returns information about a currently caught condition. The following options are available:

[C] – (Condition name)
Returns the name about the currently caught condition.

[I] - (Instruction)
Either returns 'CALL' or 'SIGNAL', the keyword of the processed instruction when the current condition was caught. This is also the default if *option* is not specified.
[D] - (Description)
Returns a descriptive string related to the currently caught condition.

Examples:

```
CONDITION()                    'SIGNAL'
CONDITION('C')                 'SYNTAX'
CONDITION('I')                 'SIGNAL'
CONDITION('D')                 'Expression 'temp' is not a number'
```

4.36 COPIES( )

```
COPIES(string,copies)
```

Returns a string with *copies* concatenated copies of *string*. *Copies* must be a non-negative whole number. No extra space is added between the copies.

Examples:

```
COPIES('Hello',3)                             'HelloHelloHello'
COPIES('*',16)                                '****************'
COPIES('Hello ',2)                            'Hello Hello '
COPIES('',10000)                            ''
COPIES('Hello',0)                           ''
```

4.37 COUNTSTR( )

```
COUNTSTR(needle,haystack)
```

Returns a count of the number of occurrences of *needle* in *haystack* that do not overlap.

Examples:

```
COUNTSTR('','')                             0
COUNTSTR('e','Hello')                       1
COUNTSTR(0,0)                               1
COUNTSTR('x','Hello')                       0
COUNTSTR('x','')                            0
COUNTSTR('','Hello')                        0
COUNTSTR('Hello','HelloWorld')              1
COUNTSTR('HelloW','Hello'                   0
COUNTSTR('He','HelloHelloHello')            3
```

4.38 CP_TRANSLATE( )

```
CP_TRANSLATE(area,from CCSID, to CCSID)
```

Performs a translation of *area* from code page *from CCSID* to code page *to CCSID*.

Example:

```
CP_TRANSLATE(INPUT_AREA, 1141, 1252)
```

4.39 D2C( )

```
D2C(integer[,length])
```

Returns a (packed) string that is the character representation of *integer* (must be a whole number) and is governed by the settings of NUMERIC, not by the internal precision of the BIFs. If *length* is specified, the returned string will be *length* bytes long. If *length* (must be a non-negative whole number) is not large enough to hold the result, the result is cut on the left. If *length* is larger than the string length, the string on the left is padded with '00'x (for a positive *integer*) or with 'FF'x (for a negative *integer*).
If *length* is not specified, *integer* is interpreted as an unsigned number. If *integer* is negative, it is interpreted as a complement of two and *length* must be specified.

Examples:

```
D2C(0)                          '00'x
D2C(127)                        '7F'x
D2C(127,3)                      '00007F'x
```

```
D2C(128)                            '80'x
D2C(128,3)                          '000080'x
D2C(-128)                           'FF80'x
D2C(-10,3)                          'FFFFF6'x
D2C(543,3)                          '0021f'x
D2C(543,2)                          '021F'x
D2C(543,1)                          '1F'x
```

4.40 D2X( )

```
D2X(integer[,length])
```

Returns a hexadecimal number that is the hexadecimal representation of *intege*r. *Integer* must be a whole number under the current settings of NUMERIC. It is not affected by the precision of the BIFs.
If *length* is not specified, *integer* must be non-negative.
If *length* is specified, the resulting string will have that length. If *length* is larger than the string length, the string on the left is padded with '00'x (for a positive *integer*) or with 'FF'x (for a negative *integer*). If *length* is not large enough to hold *intege*r, the result will be cut on the left.

Examples:

```
D2X(0)                              '0'
D2X(127)                            '7F'
D2X(128)                            '80'
D2X(128,5)                          '00080'x
D2X(-128,5)                         'FFF80'x
D2X(-10,5)                          'FFFF6'x
D2X(129,3)                          '081'x
D2X(129,2)                          '81'x
D2X(129,1)                          '1'x
```

4.41 DATATYPE( )

```
DATATYPE(string[,option])
```

This function identifies the "datatype" of *strin*g. The returned value will be "NUM" if *string* is a valid REXX number. Otherwise "CHAR" is returned. Note that the interpretation of whether *string* is a valid number will depend on the current settings of NUMERIC.
If *option* is specified as well, it will check whether *string* is of a particular data type and return either "1" or "0". The possible values of *option* are:

[A] (Alphanumeric)
Only consists of alphabetic characters (in upper, lower, or mixed case) and decimal digits.
[B] (Binary)
Only consists of the two binary digits 0 and 1. Blanks are only allowed between groups of four binary characters.
[L] (Lower)
Only consists of alphabetic characters in lower case.
[M] (Mixed)
Only consists of alphabetic characters, but the case does not matter (i.e. upper, lower, or mixed.)
[N] (Numeric)

If *string* is a valid number, DATATYPE(*string)* returns NUM.
[S] (Symbolic)
Consists of characters that are valid REXX symbols.
[U] (Upper)
Only consists of uppercase alphabetic characters.
[W] (Whole)
Determines whether *string* is a valid whole number under the current setting of NUMERIC.
[X] (Hexadecimal)
Consists of only hexadecimal digits, i.e. the decimal digits 0-9 and the alphabetic characters A-F in either case (or mixed). Blanks are not allowed.

To check whether a string is suitable as a variable name, the SYMBOL() function should be used. SYMBOL only verifies which characters *string* contains, not the order. Pay attention to lowercase alphabetic characters that are allowed in the tail of a compound symbol, but not in a simple or STEM symbol or in the head of a compound symbol.
Also note that the behavior of the options A, L, M, and U might depend on the language setting.

Examples:

```
DATATYPE(' - 2.34E-2 ')                   'NUM'
DATATYPE('1E123456789')                   'NUM'
DATATYPE('!§$%&/()#@`')                   'CHAR'
DATATYPE('HelloWorld','A')                '1'
DATATYPE('Hello World','A')               '0'
DATATYPE('1100110000110011','B')          '1'
DATATYPE('1100 1100 0011 0011','B')       '1'
DATATYPE('11 00 11 00 00 11 00 11','B')   '0'
DATATYPE('helloworld','L')                '1'
DATATYPE('HelloWorld','M')                '1'
DATATYPE(' -12E2 ','N')                   '1'
DATATYPE('ONE_SYMBOL','S')                '1'
DATATYPE('ONE+SYMBOL','S')                '0'
DATATYPE('Hello World','S')               '0'
DATATYPE('HELLOWORLD','U')                '1'
DATATYPE('ABCDEF578','X')                 '1'
DATATYPE('12.3','W')                      '0'
DATATYPE('12.0','W')                      '1'
```

4.42 DATE( )

```
DATE([option_out [,date [,option_in]]])
```

This function returns information relating to the current date. If the *option_out* character is specified, it will set the format of the return string. The default value for *option_out* is "N".

Possible options are:

[B] (Base)
The number of complete days from January 1st 0001 up to and including yesterday. The value is returned as a whole number. This function uses the Gregorian calendar extended backwards. Date('B') // 7 equals the day of the week, whereas 0 corresponds to Monday and 6 to Sunday.
[C] (Century)

The number of days in this century from January 1st -00 up to and including today. The return value is a positive integer.
[D] (Days)
The number of days in this year from January 1st up to and including today. The return value is a positive integer.
[E] (European)
The date in European format, i.e. "dd/mm/yy". Single digit values are displayed with a leading zero.
[M] (Month)
The unabbreviated name of the current month in English.
[N] (Normal)
Returns the date with the name of the month abbreviated to three letters with only the first letter in uppercase. The format is "dd Mmm yyyy", whereas Mmm is the month abbreviation (in English) and dd is the day of the month without leading zeros.
[O] (Ordered)
Returns the date in the ordered format: "yy/mm/dd"
[S] (Standard)
Returns the date according to the format specified by International Standards Organization Recommendation ISO/R 2014-1971 (E). The format is "yyyymmdd" and each part is padded with a leading zero where appropriate.
[U] (USA)
Returns the date in the format that is used in the USA, i.e. "mm/dd/yy". Each part is padded with a leading zero where appropriate.
[W] (Weekday)
Returns the English unabbreviated name of the current weekday for today. The first letter of the result is in uppercase, the rest in lowercase.
[T] (*time*_t)
Returns the current date/time in UNIX *time_t* format. *time_t* is the number of seconds since January 1st 1970.
[I] – (*ISO timestamp*)
Returns the date/time in ISO timestamp format, i.e. “yyyy-mm-dd-hh.mm.ss.msec” of the operating system. Each part is padded with a leading zero where appropriate. The milliseconds are specified with six digits.
[G] – (*ISO timestamp with UTC time*)
Returns the date/time in ISO timestamp in UTC time format, i.e. “yyyy-mm-dd-hh.mm.ss.msec” in dependence of the operating system. Each part is padded with a leading zero where appropriate. The milliseconds are specified with six digits.

Please note that none of the formats of the DATE() function is affected by the settings of NUMERIC. If there is more than one call to DATE() (and TIME()) in a single clause of tcSCRIPT code, all of them will use the same base data for calculating the date (and time).

Examples:

Assuming that today is March 13 th 2014:
```
DATE('B')                     '735304'
DATE('C')                     '5186'
DATE('D')                     '72'
DATE('E')                     '13/03/14'
DATE('M')                     'March'
DATE('N')                     '13 Mar 2014'
DATE('O')                     '14/03/13'
DATE('S')                     '20140313'
DATE('U')                     '03/13/14'
DATE('W')                     'Thursday'
DATE('T')                     1394665200
DATE('I')                     '2014-13-05-23.00.00.123456'
DATE('G')                     '2014-13-05-21.00.00.123456'
```

If the *date* option is specified, the function provides date conversions. The optional *option_in* specifies the format of *date*. The possible values for *option_in* are: **BDEOUNST.** The default value for *option_in* is N.

Examples:

```
DATE('O','13 Mar 2014')       '14/03/13'
DATE('O','12/24/70','U')      '70/12/24'
```

If the *date* does not include a century in its format, the result is chosen to make the year within 50 years past or 49 years future of the current year.

4.43 DELSTR( )

```
DELSTR(string,start[,length])
```

Returns *strin*g after the substring of length *length* starting at position *start* has been removed. The default value for *length* is the rest of the string. *Start* must be a positive whole number, *length* must be a non-negative whole number. It is not an error if *start* or *length* (or a combination of them) refers to more characters than *string* holds.

Examples:

```
DELSTR('Hello',4)                   'Hel'
DELSTR('Hello',2,3)                 'Ho'
DELSTR('Hello',3,4)                 'He'
DELSTR('Hello',8)                   'Hello'
```

4.44DELWORD( )

```
DELWORD(string,start[,length])
```

Removes *length* words and all blanks between them from *strin*g starting at word number *start*. The default value for *length* is the rest of the string. All consecutive spaces right after the last deleted word – but no spaces before the first deleted word – are removed. Nothing is removed if *length* is zero.
The valid range of *start* is the positive whole number. The first word in *string* is numbered 1. Valid values for *length* are non-negative integers. It is not an error if *start* or *length* (or a combination of them) refers to more words than *string* holds.

Examples:

```
DELWORD('This is a test',4)         'This is a '
DELWORD('This is a test',3,1)       'This is test'
DELWORD('This is a test',3,5)       'This is'
DELWORD('This is a test',1,2)       'a test'
DELWORD('This is    a test',3,1)    'This is    test'
DELWORD('This is a    test',2,2)    'This test'
```

4.45 DIGITS( )

```
DIGITS()
```

Returns the current precision of arithmetic operations. This value is set using the NUMERIC statement.

For more information, refer to the documentation on NUMERIC.

Examples:

```
DIGITS()                            9
```

4.46 E1141_A1252( )

```
E1141_A1252(expression)
```

Converts the result of *expression* from EBCDIC CCSID 1141 to ASCII CCSID 1252.

Example:

```
Result = E1141_A1252(FLDA)
```

4.47 E2A( )

```
E2A(expression)
```

Converts the result of *expression* from EBCDIC to ASCII.

Example:

```
IF E2A(substr(FLDA,1,1) = FIELD_ASCII
```

4.48 ENCEDC

```
ENCDEC( expr )
```

Decrypts the encrypted password given in expr. The encrypted password needs to start with "enc". If the password starts with other characters, expr is returned.

Example:

```
PASSWORD=ENCDEC('encd52e205a92473e5c2c06986139dd03aa')
```

4.49 FILEGET( )

```
FILEGET( connection, dir_or_file )
connection is defined as:
ADDRESS=ip-addr/name,PORT=port[,SSL={Y|N}[,ssldef]]
```

This function opens a TCP/IP listener for the specified address/port and waits for a connection attempt from the FILEPUT function.

When using SSL=Y, an encrypted connection is expected. The required certificate information is to be provided with *ssldef* (refer to SOCKETCALL - SSL_CONNECT function).

If a directory is specified with *dir_or_file* (name ends with \ under Windows, with / under Linux or Unix and with . under z/OS), the file sent by FILEPUT() will be stored in this directory with the original name. If the full name is specified, the file will be created under the given name.

Return values:

0, ...	Call was successful; statistical information is provided
-1, ...	Error during the call; an error message is supplied.

After a successful connect, the function terminates.

Example:
```
rc = fileget('ADDRESS=192.168.0.68,PORT=4444', 'C:\temp\')
say rc
```

4.50 FILEPUT( )

```
FILEPUT( connection, file [,options])
connection is defined as:
ADDRESS=ip-addr/name,PORT=port[,SSL={Y|N}[,ssldef]]
```

This function opens a connection to the specified address/port and sends the contents of the file *file* (under z/OS also VSAM cluster data). The destination address can be a TCP/IP listener opened by the FILEGET() function or a tcVISION agent. If the data is sent to a tcVISION agent, the file is stored in the working directory of the agent.

When using SSL=Y, an encrypted connection is established. The required certificate information is to be provided with *ssldef* (refer to SOCKETCALL - SSL_CONNECT function).

The transfer of the data is binary.

Possible options:

RDW:	When sending variable data from z/OS, RDWs (Record Descriptor Words) are generated and written to the destination file. When sending data with RDWs to z/OS, these are used to convert the data into records with variable record length.

Return values:

0, ...	Call was successful; statistical information is provided
-1, ...	Error during call; an error message is provided

Example:
```
rc = filePUT('ADDRESS=192.168.0.68,PORT=4444', 'VSAM.ARTICLE.CLUSTER','RDW')
say rc
```

4.51 FIND( )

```
FIND( string, phrase [,start] )
```

Determines the number of words in *string* that matches the *phrase*. A requirement is that *phrase* is a subset of *string*. If this is not the case, 0 is returned to indicate that the phrase was not found. A phrase has a significant difference to a substring: A phrase is an accumulation of words that can be separated by any number of blanks.
As an example: "is a" is a subphrase of "This is a phrase". If *start* has been specified, a positioning takes place on the word in *string* from which the search should start. The default for *start* is 1.

Examples:

```
FIND('to be or not to be','or not')          3
FIND('to be or not to be','not   to')        4
FIND('to be or not to be','to be')           1
FIND('to be or not to be','to be',3)         5
```

4.52 FORM( )

```
FORM( )
```

Returns the actual format setting for arithmetic operations. The value is set based on the setting of NUMERIC.

Example:

```
FORM()                  'SCIENTIFIC'
```

4.53 FORMAT( )

```
FORMAT(number[,[before][,[after][,[expp][,[expt]]]]])
```

This function is used to control the format of numbers. The size and format of the number may be requested. The parameter *number* is the number to be formatted and must be a valid number. Note that this number will be normalized according to the current settings of NUMERIC before any conversion or formatting is done.
The *before* and *after* parameters determine how many characters are used before or after the decimal point. Note that *before* does **not** specify the number of digits in the integer part but the size of the field in which the integer part of the number is written. It is also necessary to allocate space in this field for a minus if that is relevant. If the field is not long enough to hold the integer part (including a minus if relevant), an error is reported.
The *after* parameter will dictate the size of the field in which the fractional part of the number is written. The decimal point itself is not a part of that field, but the decimal point will be omitted if the field holding the fractional part is empty. If there are fewer digits in the number than the

size of the field, it is padded with zeros on the right. If there are more digits than fit into the field, the number will be rounded (not truncated) to fit the field.
before must at least be large enough to hold the integer part of *numbe*r. Therefore it can never be less than 1 and never less than 2 for negative numbers. The integer field will have no leading zeros, except when a single zero digit of the integer part of *number* is empty.
The parameter *expp specifies* the size of the field the exponent is written in. This is the size of the numeric part of the exponent, so the "E" and the sign come in addition, i.e. the real length of the exponent is two more than *expp* specifies. If *expp* is zero, it signalizes that exponential form should not be used. *Expp* must be a non-negative whole number. If *expp* is positive, but not large enough to hold the exponent, an error is reported.
Expt is the trigger value that decides when to switch from simple to exponential form. Usually the default precision (NUMERIC DIGITS) is used, but *expt* overrides that. Note that exponential form is used if *expt* is set to zero. If *expt* tries to force exponential form, simple form will still be used even if *expp* is zero. Negative values for *expt* result in an error. Exponential form is used if more digits than *expt* are needed in the integer part or more than twice the *expt* digits are needed in the fractional part.
The *after* number will mean different things in exponential and simple form. If *after* is set to e.g. 3, then in simple form it will force the precision to 0.001, no matter the magnitude of the number. If in exponential form, it will force the number to four digits precision.

Examples:

```
FORMAT(10.23,3,4)                         ' 10.2300'
FORMAT(10.23,3,,3,0)                      ' 1.023E+001'
FORMAT(10.23,2,1)                         ' 10.2'
FORMAT(10.23,2,0)                         ' 10'
FORMAT(10.23,,,,0)                        '1.023E+1'
FORMAT(10.23,,,0)                         '10.23'
FORMAT(10.23,,,0,0)                       '10.23'
```

4.54 GETPID()

```
GETPID()
```

Returns the process id of the currently running process.

Example:

```
mypid = GETPID()
```

4.55 IMPORT()

```
IMPORT( address, [,length] )
```

Creates a string by copying the data specified by the 4 byte address *address*. If parameter *length* is not specified, the copy process is terminated by the first NULL byte. The address must be specified according to the endianess of the machine.
IMPORT can only run on mainframe machines. On all other machines, a NULL string is returned.

Example:

```
IMPORT('0004 0000'x, 9)      'The data'  /* only an example */
```

4.56 INDEX( )

```
INDEX(haystack,needle[,start])
```

Returns the character position of the string *needle* in *haystack*. If *needle* is not found, 0 is returned. The search starts at the first character of haystack *(start* is 1) by default. This can be overridden by giving a different *start* that is a positive whole number.

Examples:

```
INDEX('Hello','el')                 '2'
INDEX('Hello','xy')                 '0'
INDEX('Hello','el',3)               '0'
INDEX('HelloHello ','el',3)         '7'
INDEX('Hello','el',6)               '0'
```

4.57 INSERT( )

```
INSERT(string1,string2[,position[,length[,padchar]]])
```

Returns the result of inserting *string1* into a copy of *string*2. If *position* is specified, it marks the character in *string2* that *string1* is to be inserted after. *Position* must be a non-negative whole number, and it defaults to 0 meaning *string2* is put in front of the first character in *string*1. Padding is performed with *padchar* or space if *padchar* is not specified.
If *length* is specified, *string1* is truncated or padded on the right side to make it exactly *length* characters long before it is inserted. If padding occurs, *padchar* is used - or <space> if *padchar* is undefined.

Examples:

```
INSERT('Hello','World')             'HelloWorld'
INSERT('Hello','World',2)           'WoHellorld'
INSERT('Hello','World',2,10)        'WoHello     rld'
INSERT('Hello','World',2,10,'*')    'WoHello*****rld'
INSERT('Hello','World',2,4)         'WoHellrld'
INSERT('Hello','World',8)           'World   Hello'
INSERT('Hello','World',8,,'*')      'World***Hello'
```

4.58 JOBID()

```
JOBID( )
```

Creates a string that contains the ID of the job. This function can only be executed on mainframe machines. On all other machines, string 'NoJobId' is returned.

Example:

```
JOBID()           'JOB00039'   /* on a mainframe (example)         */
JOBID()           'NoJobId'    /* on a non mainframe machine       */
```

4.59 JOBNAME()

```
JOBNAME( )
```

Creates a string that contains the name of the job. This function can only be executed on mainframe machines. On all other machines string 'NoJobNam' is returned.

Example:

```
JOBNAME()         'TVSM50'     /* on a mainframe (example)         */
JOBNAME()         'NoJobNam'   /* on a non mainframe machine       */
```

4.60 JUSTIFY( )

```
JUSTIFY(string,length[,pad]) (CMS)
```

Formats words separated by blanks in *strin*g by adding *pad* characters between words to align them with both margins. That means an adjustment to width *length (length* must be non-negative). The default *pad* character is a blank. *string* is first normalized as though SPACE(*strin*g) had been executed (meaning multiple blanks are converted to single blanks, and leading and trailing blanks are removed). If *length* is less than the width of the normalized string, the string is truncated on the right and any trailing blank is removed. Extra *pad* characters are added evenly to provide the required length and the blanks between words are replaced with the *pad* character.

Examples:

```
JUSTIFY('The good shepherd',14)         'The good sheph'
JUSTIFY('The good shepherd',8)          'The good'
JUSTIFY('The good shepherd',9)          'The  good'
JUSTIFY('The good shepherd',9,'+')      'The++good'
```

4.61 LASTPOS( )

```
LASTPOS(needle,haystack[,start])
```

Searches the string *haystack* for the string *needl*e and returns the position in *haystack* of the first character in the substring matching *needl*e. The search is started from the right side, so if *needle* occurs several times, the last occurrence is reported. If *start* is specified, the search starts at character number *start* in *haystac*k. Note that the standard only states that the search starts at the *star*t character.

Examples:

```
LASTPOS('be','To be or not to be')             17
LASTPOS('be','to be or not to be',10)          4
LASTPOS('is','to be or not to be')             0
LASTPOS('to','to be or not to be',1)           0
```

4.62 LEFT( )

```
LEFT(string,length[,padchar])
```

Returns the *length* leftmost characters in *strin*g. If *length* (must be a non-negative whole number) is greater than the length of *strin*g, the result is padded on the right with <space> (or *padchar* if that is specified).

Examples:

```
LEFT('Hello world',7)                                    'Hello w'
LEFT('Hello world',3)                                    'Hel'
LEFT('Hello world',13)                                   'Hello world  '
LEFT('Hello world',13,'*')                               'Hello world**'
```

4.63 LENGTH( )

```
LENGTH(string)
```

Returns the number of characters in *strin*g.

Examples:

```
LENGTH('')                                               0
LENGTH('Hello')                                          5
LENGTH('Hello world')                                    11
LENGTH(' Hello world  ')                                 14
```

4.64 LINEIN( )

```
LINEIN([streamid][,[line][,count]])
```

Returns a line read from a file. If only *streamid* is specified, the reading starts at the current read position and continues to the first end-of-line (EOL) mark. Afterwards, the current read position is set to the character behind the EOL mark which terminated the read operation. If the operating system uses special characters for EOL marks, these are not returned as a part of the read string.
The default value for *streamid* is the default input stream. The format and range of the string *streamid* depend on the implementation.

The *line* parameter (must be a positive whole number) might be specified to set the current position in the file to the beginning of line number *line* before the read operation starts. If *line* is unspecified, the current position will not be changed before the read operation. *line* is only valid for persistent streams. For transient streams, an error is reported if *line* is specified. The first line in the stream is numbered 1.
Count specifies the number of lines to read. However, it can only take the values 0 and 1. If it is 1 (which is the default), it will read one line. If it is 0, it will not read any lines, the actual read position for the data is set, and a null string is returned.
It depends on the implementation whether *stream* must be explicitly opened before a read operation can be performed. In many implementations, a read or write operation will implicitly open the stream if not already open.

Examples:

Assuming that the file /tmp/file contains the three lines: *"First lin*e", *Second lin*e", and *"Third lin*e":

```
LINEIN('/tmp/file',1)         'First line'
LINEIN('/tmp/file')           'Second line'
LINEIN('/tmp/file',1,0)       '' /* Sets read position to 1 */
LINEIN('/tmp/file')           'First line'
LINEIN()                      'Hi, there!'      /* example */
```

Example how to read a file on the mainframe. The example reads a member of a partitioned dataset.

```
LINEIN('TCVISISON.V100.MACLIB(TEST)',1)    'First line'
```

4.65 LINEOUT( )

```
LINEOUT([streamid][,[string][,line]])
```

Returns the number of lines remaining after positioning the stream *streamid* to the start of line *line* and writing out *string* as a line of text. If *streamid* is omitted, the default output stream is used. If *line* (must be a positive whole number) is omitted, the stream will not be repositioned before the write. If *string* is omitted, nothing is written to the stream. If *string* is specified, a system-specific action is taken after it has been written to stream to mark a new line.
If *string* is specified but not *lin*e, *string* is written to the stream starting at the current write position. If *line* is specified but not *strin*g, the stream is positioned in the new position.
The *line* parameter is only valid for persistent streams. If *line* is specified for a transient stream, an error is issued. The current write position cannot be used for transient streams.
If neither *line* nor *string* is specified, the standard requires that the current write position is set to the end of the stream and implementation-specific side effects may occur. In practice, this means that an implementation can use this situation to do things like closing the stream or flushing the output.
Note that if *string* should be written to a line in the middle of the stream (i.e. *line* is less than the total number of lines in the stream), the behavior is system- and implementation-specific. Some systems will truncate the stream after the newly written line, others will only truncate if the newly written line has a different length than the old line that it replaced, and yet other systems will overwrite and never truncate.

Examples:

```
LINEOUT(,'First line')
LINEOUT('/tmp/file','Second line',2)
LINEOUT('/tmp/file','Third line')
LINEOUT('/tmp/file','Fourth line',4)
```

4.66 LINES()

```
LINES( [streamid] [,option].)
```

Returns 1 if there is at least one complete line remaining in the named file *streamid*. Returns 0 if no complete lines remain in the file. A complete line is not as complete as the name might indicate. A complete line is zero, one or more characters, followed by an end-of-line (EOL) marker. So if half a line has been read, a "complete" line is still left.
The format and contents of the stream *streamid* depend on system and implementation.
Possible options are:
[**C**] (Count)
Returns the current number of complete lines remaining in the stream.
[**N**] (Normal)
Returns 1 if there is at least one complete line remaining in the file, or 0 if no lines are remaining. This is the default option.

For transient streams, LINES only returns 0 or 1 because repositioning on these streams is not possible. The remaining lines cannot be counted.

Example:

```
LINES('/tmp/file')                  1      /* only a possible value */
LINES('/tmp/file','C')              12     /* only a possible value */
LINES()                             1      /* only a possible value */
```

4.67 LOWER( )

```
LOWER( string )
```

Translates *string* in lowercase characters.

Example:

```
lower('Hello WORLD')                'hello world'
```

4.68 LS_NOTE( )

```
LS_NOTE( stream_name, note)
```

Writes the content of *note* into the logstream *stream_name*.
The exit CDC_ExitDBControl with the content of *note* is called while processing the data from the logstream in order to start the respective processing in the target system.

Example:
```
LS_NOTE('START1.CICSTS.DFHJ01', 'REPRO;MY.VSAM.FILE;MY.REPRO.INPUT.FILE')
```

4.69 MAX( )

```
MAX(number1[,number2]...)
```

Takes the parameters and returns the parameter with the highest numerical value. The parameters may be any valid number. The returned number is normalized according to the current settings of NUMERIC, thus the result need not be strictly equal to any of the parameters.

Examples:

```
MAX(1,7,9,4)                    9
MAX(5)                          5
MAX(-12,.05E2,7)                7
MAX(1,7,09.0,4)                 9.0
```

4.70 MIN()

```
MIN(number[,number]...)
```

Like MAX(), except that the lowest numerical value is returned.

Examples:

```
MIN(1,7,9,4)                    1
MIN(5)                          5
MIN(-12,.05E2,7)                -12
MIN(1,7,09.0E-1,4)              0.90
```

4.71 OVERLAY( )

```
OVERLAY(string1,string2[,[start][,[length][,padchar]]])
```

Returns a copy of *string*2, totally or partially overwritten by *string*1. If these are the only arguments, the overwriting starts at the first character in *string*2. If *start* is specified, the first character in *string1* overwrites character number *start* in *string*2. *start* must be a positive whole number. The default is 1. If the *start* position is longer than the length of *string2, string2* is padded on the right end to make it *star*t-1 characters long before *string1* is added.
If *length* is specified, *string2* will be stripped or padded at the right end to match the specified length. For padding (of both strings) *padchar* is used. <space> is used if *padchar* is unspecified. *Length* must be non-negative and defaults to the length of *string*1.

Examples:

```
OVERLAY('XYZ','original')                 'XYZginal'
OVERLAY('XYZ','original',3)               'orXYZnal'
OVERLAY('XYZ','original',3,5)              'orXYZ  l'
OVERLAY('XYZ','original',3,5,'*')          'orXYZ**l'
OVERLAY('XYZ','original',3,2)              'orXYinal'
OVERLAY('XYZ','original',9)                'originalXYZ'
OVERLAY('XYZ','original',11)               'original  XYZ'
OVERLAY('XYZ','original',11,,'*')          'original**XYZ'
OVERLAY('XYZ','original',11,5,'*')         'original**XYZ**'
```

4.72 POS( )

```
POS(needle,haystack[,start])
```

Seeks an occurrence of the string *needle* in the string *haystack*. If *needle* is not found, 0 is returned. Otherwise the position of the first matching character in *haystack* is returned as positive whole number. If *start* (which must be a positive whole number) is specified, the search for *needle* will start at position *start* in *haystack*.

Examples:

```
POS('be','to be or not to be')             4
POS('to','to be or not to be',10)          14
POS('is','to be or not to be')             0
POS('to','to be or not to be',18)          0
```

4.73 PROCESSCANCEL( )

```
PROCESSCANCEL( connection, name, pid )
connection is defined as:
HOSTNAME=ip-addr/name,PORT=port[,AGENT=ip-addr/name]
        [,USER=userid,PWD=password][,SSL={Y|N}[,ssldef]]
```

The function cancels the process indicated by *name* and *pid* on the corresponding Agent. When using AGENT=, the request is forwarded to the named Agent. When using SSL=Y, an encrypted connection is established. The required certificate information is to be provided with *ssldef* (refer to SOCKETCALL - SSL_CONNECT function).

The return code is either 0 if it has been canceled successfully or the error code of the communication. The value of *pid* can be determined with a previous call of functions PROCESSINSTANCE() or PROCESSLIST().

Return code values:

0	Call successful
<>0	Error during call

Example:

```
rc = PROCESSCANCEL("HOSTNAME=192.168.0.231,PORT=4131", "TEST", pid)
```

4.74 PROCESSINFO()

```
PROCESSINFO(connection, name, pid, stem)
connection is defined as:
HOSTNAME=ip-addr/name,PORT=port[,AGENT=ip-addr/name]
        [,USER=userid,PWD=password][,SSL={Y|N}[,ssldef]]
```

Returns statistical information about the process identified by *name* and *pid* on the specified Agent. When using AGENT=, the request is forwarded to the named Agent. When using SSL=Y, an encrypted connection is established. The required certificate information is to be provided with *ssldef* (refer to SOCKETCALL - SSL_CONNECT function).

The return value is zero if the function has been completed successfully or contains an error code. The content of argument *pid* can be determined using a previous call of function PROCESSINSTANCE() or PROCESSLIST().

0	Call successful
<> 0	Error during call

After a successful completion of the function, the following information is provided in STEM variables:

STEM.ScriptName	Process name
STEM.ScriptID	Process PID
STEM.ScriptStatus	Internal process state
STEM.LC_InCount	Number of input control records
STEM.LB_InCount	Number of input data blocks
STEM.LR_InCount	Number of input data records
STEM.LC_OutCount	Number of output control records
STEM.LB_OutCount	Number of output data blocks
STEM.LR_OutCount	Number of output data records
STEM.LUW_Count	Number of processed LUWs
STEM.LUW_MaxOpenCount	Number of parallel open LUWs
STEM.LUW_UncomittedCount	Number of current open LUWs
STEM.BUBC_Inserted	BatchCompare: Number of detected INSERTs
STEM.BUBC_Deleted	BatchCompare: Number of detected DELETEs
STEM.BUBC_Updated	BatchCompare: Number of detected UPDATEs
STEM.BUBC_Unchanged	BatchCompare: Number of unchanged records
STEM.STAT_MemoryAllocated	Max. allocated storage
STEM.RemoteScriptName	Name of remote process
STEM.RempteScriptID	PID of remote process
STEM.RemoteSysid	Sysid where the remote process is executed
STEM.LastProcessTimestamp	Timestamp of last processed data records

Example:

```
connection = "HOSTNAME=192.168.0.231,PORT=4131"
name = "TEST"
rc = PROCESSINSTANCE(connection, name)
IF rc > 0 THEN DO
  pid = rc
  say 'PID of process =' pid
  rc = PROCESSINFO(connection, name, pid, "List")
  IF rc <> 0 THEN DO
    say 'Error during PROCESSINFO' rc
  END
  ELSE DO
    say "LastProcessTimestamp:" List.LastProcessTimestamp
  END
END
```

4.75 PROCESSINSTANCE()

```
PROCESSINSTANCE(connection, name)
connection is defined as:
HOSTNAME=ip-addr/name,PORT=port[,AGENT=ip-addr/name]
        [,USER=userid,PWD=password][,SSL={Y|N}[,ssldef]]
```

This function checks whether a process with the specified *name* is currently running on the defined Agent. When using AGENT=, the request is forwarded to the named Agent. When using SSL=Y, an encrypted connection is established. The required certificate information is to be provided with *ssldef* (refer to SOCKETCALL - SSL_CONNECT function).

The following return codes are possible:

-1	A communication error has occurred (error message in REXX variable RESULT).
0	The process is not active.
>0	The process is active, the return code is the current process ID (*pid*).

Example:

```
rc = PROCESSINSTANCE("HOSTNAME=192.168.0.231,PORT=4131", "TEST")
IF rc > 0 THEN say 'PID of process =' rc
```

4.76 PROCESSLIST()

```
PROCESSLIST(connection, stem)
connection is defined as:
HOSTNAME=ip-addr/name,PORT=port[,AGENT=ip-addr/name]
        [,USER=userid,PWD=password][,SSL={Y|N}[,ssldef]]
```

Returns a list of all processes currently executed on the Agent. When using AGENT=, the request is forwarded to the named Agent. When using SSL=Y, an encrypted connection is established. The required certificate information is to be provided with *ssldef* (refer to SOCKETCALL - SSL_CONNECT function).

Possible return values are:

0	Call successful
<>0	Error during call

After a successful execution of the function, the number of entries is returned in STEM variable STEM.0. The individual values for every entry are part of STEM variables STEM.n.ProcessName and STEM.n.ProcessID.

Example:

```
rc = PROCESSLIST("HOSTNAME=192.168.0.231,PORT=4131", "List")
IF rc <> 0 THEN DO
      say 'Error during function PROCESSLIST' rc
END
ELSE DO
      SAY "No. of processes:" List.0
      DO i = 1 TO List.0
         SAY "Name:" List.i.ProcessName ", PID:" List.i.ProcessID
      END
END
```

4.77 PROCESSSTART()

```
PROCESSSTART(connection, name [, parameter])
connection is defined as:
HOSTNAME=ip-addr/name,PORT=port[,AGENT=ip-addr/name]
        [,USER=userid,PWD=password][,SSL={Y|N}[,ssldef]]
```

The function starts the process defined with *name* on the specified Agent. When using AGENT=, the request is forwarded to the named Agent. When using SSL=Y, an encrypted connection is established. The required certificate information is to be provided with *ssldef* (refer to SOCKETCALL - SSL_CONNECT function).

To (re)start only a specific process part, the full name of the process part must be specified (e.g. "ADABAS NRT:Processing:Applying"). For correct execution of the start attempt, this process part must have been active at least once during the current agent run. If this is not the case, the start is rejected. It is also checked whether this process part is currently already active. If it is, the start is rejected.

The return code is ZERO if the process has been started successfully. If the process start fails, the return code contains the error code. Parameter of stem PM_I. can be passed to the process.

Example:

```
parm="PM_I.INPUT_SOURCE_NAME=ISNRANGE=00100-00200"
rc = PROCESSSTART("HOSTNAME=192.168.0.231,PORT=4131", "TEST", parm)
IF rc = 0 THEN say 'Process successfully started'
```

4.78 PROCESSSTOP()

```
PROCESSSTOP(connection, name, pid [stopdescription])
connection is defined as:
HOSTNAME=ip-addr/name,PORT=port[,AGENT=ip-addr/name]
        [,USER=userid,PWD=password][,SSL={Y|N}[,ssldef]]
```

The function stops the process specified with *name* and *pid* on the corresponding Agent. When using AGENT=, the request is forwarded to the named Agent. When using SSL=Y, an encrypted connection is established. The required certificate information is to be provided with *ssldef* (refer to SOCKETCALL - SSL_CONNECT function).

The optional parameter *stopdescription* can be used to display the reason for stopping the process. The return code is ZERO if the stop of the process was successful. If it was unsuccessful, the return code contains the error code. The *pid* number can be obtained by functions PROCESSINSTANCE() or PROCESSLIST().

Example:

```
rc = PROCESSSTOP("HOSTNAME=192.168.0.231,PORT=4131", "TEST", pid)
rc = PROCESSSTOP("HOSTNAME=192.168.0.231,PORT=4131", "TEST", pid,
                  "maintenance shutdown")
```

4.79 RANDOM( )

```
RANDOM(max)
RANDOM([min][,[max][,seed]])
```

Returns a pseudo-random whole number. If called with only the first parameter, the first format will be used and the returned number is in the range 0 up to and including the value of the first parameter. Parameter *max* must be a non-negative whole number, not greater than 100000. If the function is called with more than one parameter or with a parameter that is not the first, the second format will be used. Then *min* and *max* must be positive whole numbers, *max* cannot be less than *min* and the difference *max-min* cannot be more than 100000. If one or both of them is unspecified, the default for *min* is 0 and the default for *max* is 999.
If *seed* is specified (must be a positive whole number), it is possible to control which numbers will be generated by the pseudo-random algorithm. If not specified, it will be set to some "random" value at the first call to RANDOM() (typically a function of the time). If *seed* is specified, it influences the result of the current call to RANDOM().
The standard does not require a specific method to be used for generating the pseudo-random numbers, so the reproducibility can only be guaranteed as long as the same implementation on the same machine with the same operating system is used. If any of this differs, a given *seed* may produce a different sequence of pseudo-random numbers.
Note that some numbers might have a slightly increased chance of turning up than others depending on the implementation.
Because of the special syntax, there is a big difference between using RANDOM(25) and RANDOM(25,). The former will give a pseudo-random number in the range 0-25, while the latter will give a pseudo-random number in the range 25-999.

Examples:

```
RANDOM()                     637        /* Between 0 and 999 */
RANDOM(25)                   6          /* Between 0 and 25  */
RANDOM(,25)                  17         /* Between 0 and 25  */
RANDOM(25,50)                34         /* Between 25 and 50 */
RANDOM(,,4711)               843        /* Between 0 and 999,
                                           with specified seed */
```

4.80 REVERSE( )

```
REVERSE(string)
```

Returns a string of the same length as *strin*g but the order of the characters is reversed.

Examples:

```
REVERSE('HelloWorld')               'dlroWolleH'
REVERSE(' Hello World')             'dlroW olleH '
REVERSE('1.2345')                   '5432.1'
```

4.81 RIGHT( )

```
RIGHT(string,length[,padchar])
```

Returns the *length* rightmost characters in *strin*g. If *length* (must be a non-negative whole number) is greater than the length of *string*, the result is padded on the left with the necessary number of *padchar*s. *Padchar* defaults to <space>.

Examples:

```
RIGHT('Hello World',7)              'o World'
RIGHT('Hello World',4)              'orld'
RIGHT('Hello World',13)             '  Hello World'
RIGHT('Hello World',13,'*')         '**Hello World'
```

4.82 SCRIPTCANCEL( )

```
SCRIPTCANCEL( connection, name, pid )
connection is defined as:
HOSTNAME=ip-addr/name,PORT=port[,AGENT=ip-addr/name]
        [,USER=userid,PWD=password][,SSL={Y|N}[,ssldef]]
```

The function cancels the script indicated by *name* and *pid* on the corresponding Agent. When using AGENT=, the request is forwarded to the named Agent. When using SSL=Y, an encrypted connection is established. The required certificate information is to be provided with *ssldef* (refer to SOCKETCALL - SSL_CONNECT function).

The return code is either 0 if it has been successfully canceled or the error code of the communication. The value of *pid* can be determined with a previous call of functions SCRIPTINSTANCE() or SCRIPTLIST().

Return code values:

0	Call successful
<>0	Error during call

Example:

```
rc = SCRIPTCANCEL("HOSTNAME=192.168.0.231,PORT=4131", "TEST.TSF", pid)
```

4.83 SCRIPTINFO()

```
SCRIPTINFO(connection, name, pid, stem)
connection is defined as:
HOSTNAME=ip-addr/name,PORT=port[,AGENT=ip-addr/name]
        [,USER=userid,PWD=password][,SSL={Y|N}[,ssldef]]
```

Returns statistical information about the script identified by *name* and *pid* on the specified Agent. When using AGENT=, the request is forwarded to the named Agent. When using SSL=Y, an encrypted connection is established. The required certificate information is to be provided with *ssldef* (refer to SOCKETCALL - SSL_CONNECT function).

The return value is zero if the function has been completed successfully or contains an error code. The content of argument *pid* can be determined using a previous call of function SCRIPTINSTANCE() or SCRIPTLIST().

0	Call successful
<> 0	Error during call

After a successful completion of the function, the following information is provided in STEM variables:

STEM.ScriptName	Script name
STEM.ScriptID	Script PID
STEM.ScriptStatus	Internal script state
STEM.LC_InCount	Number of input control records
STEM.LB_InCount	Number of input data blocks
STEM.LR_InCount	Number of input data records
STEM.LC_OutCount	Number of output control records
STEM.LB_OutCount	Number of output data blocks
STEM.LR_OutCount	Number of output data records
STEM.LUW_Count	Number of processed LUWs
STEM.LUW_MaxOpenCount	Number of parallel open LUWs
STEM.LUW_UncomittedCount	Number of currently open LUWs
STEM.BUBC_Inserted	BatchCompare: Number of detected INSERTs
STEM.BUBC_Deleted	BatchCompare: Number of detected DELETEs
STEM.BUBC_Updated	BatchCompare: Number of detected UPDATEs
STEM.BUBC_Unchanged	BatchCompare: Number of unchanged records
STEM.STAT_MemoryAllocated	Max. allocated storage
STEM.RemoteScriptName	Name of remote script
STEM.RempteScriptID	PID of remote script
STEM.RemoteSysid	Sysid where the remote script is executed
STEM.LastProcessTimestamp	Timestamp of last processed data records

Example:

```
connection = "HOSTNAME=192.168.0.231,PORT=4131"
```

```
name = "TEST.TSF"
rc = SCRIPTINSTANCE(connection, name)
IF rc > 0 THEN DO
  pid = rc
  say 'PID of script =' pid
  rc = SCRIPTINFO(connection, name, pid, "List")
  IF rc <> 0 THEN DO
    say 'Error during SCRIPTINFO' rc
  END
  ELSE DO
    say "LastProcessTimestamp:" List.LastProcessTimestamp
  END
END
```

4.84 SCRIPTINSTANCE()

```
SCRIPTINSTANCE(connection, name)
connection is defined as:
HOSTNAME=ip-addr/name,PORT=port[,AGENT=ip-addr/name]
        [,USER=userid,PWD=password][,SSL={Y|N}[,ssldef]]
```

This function checks whether a script with the specified *name* is currently running on the defined Agent. When using AGENT=, the request is forwarded to the named Agent. When using SSL=Y, an encrypted connection is established. The required certificate information is to be provided with *ssldef* (refer to SOCKETCALL - SSL_CONNECT function).

The following return codes are possible:

-1	A communication error has occurred (error message in REXX variable RESULT).
0	The script is not active.
>0	The script is active, the return code is the process ID (*pid*).

Example:

```
rc = SCRIPTINSTANCE("HOSTNAME=192.168.0.231,PORT=4131", "TEST.TSF")
IF rc > 0 THEN say 'PID of script =' rc
```

4.85 SCRIPTLIST()

```
SCRIPTLIST(connection, stem)
connection is defined as:
HOSTNAME=ip-addr/name,PORT=port[,AGENT=ip-addr/name]
        [,USER=userid,PWD=password][,SSL={Y|N}[,ssldef]]
```

Returns a list of all scripts currently executed on the Agent. When using AGENT=, the request is forwarded to the named Agent. When using SSL=Y, an encrypted connection is established. The required certificate information is to be provided with *ssldef* (refer to SOCKETCALL - SSL_CONNECT function).

Possible return values are:

0	Call successful
<>0	Error during call

After a successful execution of the function, the number of entries is returned in STEM variable STEM.0. The individual values for every entry are part of STEM variables STEM.n.ScriptName and STEM.n.ScriptID.

Example:

```
rc = SCRIPTLIST("HOSTNAME=192.168.0.231,PORT=4131", "List")
IF rc <> 0 THEN DO
      say 'Error during function SCRIPTLIST' rc
END
ELSE DO
      SAY "No. of scripts:" List.0
      DO i = 1 TO List.0
         SAY "Name:" List.i.ScriptName ", PID:" List.i.ScriptID
      END
END
```

4.86 SCRIPTSTART()

```
SCRIPTSTART(connection, name [, parameter])
connection is defined as:
HOSTNAME=ip-addr/name,PORT=port[,AGENT=ip-addr/name]
        [,USER=userid,PWD=password][,SSL={Y|N}[,ssldef]]
```

The function starts the script defined with *name* on the specified Agent. When using AGENT=, the request is forwarded to the named Agent. When using SSL=Y, an encrypted connection is established. The required certificate information is to be provided with *ssldef* (refer to SOCKETCALL - SSL_CONNECT function).

The return code is ZERO if the script has been started successfully. If the script start fails, the return code contains the error code. Parameters of stem PM_I. can be passed to the script.

Example:

```
parm="PM_I.INPUT_SOURCE_NAME=ISNRANGE=00100-00200"
rc = SCRIPTSTART("HOSTNAME=192.168.0.231,PORT=4131", "TEST.TSF", parm)
IF rc = 0 THEN say 'Script successfully started'
```

4.87 SCRIPTSTOP()

```
SCRIPTSTOP(connection, name, pid)
connection is defined as:
HOSTNAME=ip-addr/name,PORT=port[,AGENT=ip-addr/name]
        [,USER=userid,PWD=password][,SSL={Y|N}[,ssldef]]
```

The function stops the specified script with *name* and *pid* on the corresponding Agent. When using AGENT=, the request is forwarded to the named Agent. When using SSL=Y, an encrypted connection is established. The required certificate information is to be provided with *ssldef* (refer to SOCKETCALL - SSL_CONNECT function).

The return code is ZERO if the stop of the script was successful. If it was unsuccessful, the return code contains the error code. The *pid* number can be obtained by functions SCRIPTINSTANCE() or SCRIPTLIST().

Example:

```
rc = SCRIPTSTOP("HOSTNAME=192.168.0.231,PORT=4131", "TEST.TSF", pid)
```

4.88 SIGN( )

```
SIGN(number)
```

Returns -1, 0, or 1 depending on whether *number* is negative, zero, or positive.

Number must be a valid REXX number and is normalized according to the current settings of NUMERIC before comparison.

Examples:

```
SIGN(0.000)                    0
SIGN(-0.0)                     0
SIGN(32)                       1
SIGN(-0.000004)                -1
SIGN(-5)                       -1
```

4.89 SLEEP( )

```
SLEEP(seconds) (CMS)
```

Pauses for the supplied number of seconds.

Examples:

```
SLEEP(10)                                 /* sleeps for 10 seconds */
```

4.90 SOCKETCALL( )

```
SOCKETCALL( function [, parameter1] [, parameter2] )
function is defined as:
{ INITIALIZE | TERMINATE | CONNECT | SEND | RECEIVE | CLOSE | BIND |
  LISTEN | ACCEPT | GETHOSTBYNAME | GETSOCKETNAME | IOCTL |
  SSL_INITIALIZE | SSL_CONNECT | SSL_WRITE | SSL_READ | SSL_LISTEN |
  SSL_ACCEPT | SSL_SHUTDOWN | SSL_GET_ERROR | ERR_ERROR_STRING}
```

This function allows a TCP/IP communication between a script and a remote listener (generally a tcVISION Agent). Depending on the selected function, up to two parameters are required. The number of parameters, their meaning, and the corresponding return values are described with the function. All parameters and all return values are passed as strings.

The following functions are possible:

INITIALIZE
Initialization of the TCP/IP API.
This function can be called without parameters. On z/OS, the name of the TCP/IP address space can be supplied. The return value of the TCP/IP provider is passed back.

TERMINATE
Termination of the TCP/IP API.
This function can be called without parameters. The return value of the TCP/IP provider is passed back.

CONNECT
A connection to a remote listener is established.
This function requires the name and the port of the remote listener in parameter 1. The keywords 'HOSTNAME=' and 'PORT=' can be used.

The following is returned depending on the result of the function:
'return_code socket_id', if return_code = 0
'return_code socket_error_code', if return_code <> 0

SEND
Data is sent.
This function requires the socket_id in parameter 1 and the data that will be sent in parameter 2.

The following is returned:
'return_code socket_error_code'
If return_code > 0, return_code is the length of the data sent and socket_error_code is 0.
If return_code < 0, socket_error_code contains the return code of the TCP/IP provider.

RECEIVE
Data is received.
This function requires the socket_id in parameter 1 and the length of the data to be received in parameter 2.

The following is returned:
'return_code socket_error_code'
If return_code >= 0, return_code is the length of the received data and socket_error_code is 0.
The REXX variable 'BUFFER' contains the data.
If return_code = -1, socket_error_code contains the return code of the TCP/IP provider.

CLOSE
A socket is closed.
This function requires the socket_id in parameter 1.

The following is returned:
'return_code socket_error_code'

BIND
A local address is associated with a socket.
This function requires the name and the port in parameter 1. The keywords 'HOSTNAME=' and 'PORT=' are used. If the keyword 'HOSTNAME=' is not specified, '127.0.0.1' is used as a default value.

The following is returned depending on the result of the function:
'return_code socket_id' if return_code = 0
'return_code socket_error_code' if return_code <> 0

LISTEN
Preparation for incoming connection requests.
This function requires the socket_id in parameter 1.

The following is returned:
'return_code socket_error_code'

ACCEPT
Waiting for a client connection.
This function requires the socket_id in parameter 1 and a timeout value (in seconds) of the local listeners in parameter 2. The keyword 'TIMEOUT=' can be used.

The following is returned depending on the result of the function:

'return_code socket_id address:port' if return_code = 0
'return_code socket_error_code' if return_code <> 0

GETHOSTBYNAME
Determines the IP address for a URL.
The function requires the URL in parameter 1.

The following is returned depending on the result of the function:
'return_code ipaddress' if return_code = 0
'return_code' if return_code <> 0

GETSOCKETNAME
The local name of a socket is determined after ACCEPT.
The function requires the socket_id in parameter 1.

The following is returned depending on the result of the function:
'return_code address:port' if return_code = 0
'return_code' if return_code <> 0

IOCTL
The number of bytes that can be received by a socket is determined.
The function requires the socket_id in parameter 1 and key word 'FIONREAD' in parameter 2.

The following is returned:
'return_code number_of_read_bytes_available'

Examples for unsecured connections:
```
SOCKETCALL('INITIALIZE')           /* Initialize API                      */
SOCKETCALL('INITIALIZE', 'space')  /* Initialize API in z/OS
                                      with special address space         */

SOCKETCALL('TERMINATE')            /* Terminate API                       */

SOCKETCALL('CONNECT', 'HOSTNAME=192.168.0.123, PORT=4120')
                                   /* Connect to listener                 */

SOCKETCALL('SEND', '1234', 'Data to be send.')
                                   /* Send of data                        */

SOCKETCALL('RECEIVE', '1234', '100')
                                   /* Receive 100 bytes of data           */

SOCKETCALL('CLOSE', '1234')        /* Close socket                        */

SOCKETCALL('BIND', 'HOSTNAME=192.168.0.123, PORT=5555')
                                   /* Bind a socket to port 5555          */

SOCKETCALL('LISTEN', '3456')       /* Prepare for incoming inquiries      */

SOCKETCALL('ACCEPT', '3456', 'TIMEOUT=10')
                                   /* Wait for a client connection
                                      with a timeout of 10 seconds        */

SOCKETCALL('GETHOSTBYNAME', 'www.bos-digitec.de')
                                   /* Determine the IP-address of a URL  */
```

```
SOCKETCALL('GETSOCKETNAME', '5678')
                                    /* Determine the IP-address of
                                       socket_id (received by ACCEPT)      */

SOCKETCALL('IOCTL', '1234', 'FIONREAD')
                                    /* Determine the number of receivable
                                       bytes                               */
```

SSL_INITIALIZE
Initialization of the TCP/IP API and the SSL environment for secured connections.
This function can be called without parameters. On z/OS, the name of the TCP/IP address space can be supplied.
The function call was successful if the return code is 0. All other return codes indicate an error.

SSL_CONNECT
A connection to a remote listener is established.

This function requires the name and the port of the remote listener in parameter 1. The keywords 'HOSTNAME=' and 'PORT=' can be used.

The required certificate information for the secured connection are given in the first parameter using the following keywords:

'SSLFILE='	Name of a file in the local file system.

This file contains the certificate information using the following keywords. As an alternative, the following keywords can be given directly in parameter 1 of the function.

In z/OS environments:

'GSKKEYRING='	Name of the key database holding the imported certificates.
'GSKLABEL='	Name of the certificate in the key database to be used (optional)
'KEYPWD='	Password for the key database.
'USEFIPS='	FIPS mode usage (Y/N).

In all other environments:

'SSLCERT='	Name of the certificate file to be used for the secured connection.
'SSLCACERT='	Name of the CA certificate file for verification of the client certificate (optional).
'SSLKEY='	Name of the key file for the client certificate.
'SSLKEYPWD='	Password for the key file.

The function call was successful if the return code is 0. All other return codes indicate an error.

SSL_WRITE

Data is sent.
This function requires the data that will be sent in parameter 1.

The function call was not successful if the return code is negative. Otherwise the return code is the number of bytes sent.

SSL_READ

Data is received.
This function requires the length of the data to be received in parameter 2.

The function call was not successful if the return code is negative. Otherwise the return code is the number of bytes received. The data is available in the variable BUFFER.

SSL_LISTEN

Preparation for incoming connection requests.

This function requires the name and the port in parameter 1. The keywords 'HOSTNAME=' and 'PORT=' are used. The required certificate information for the secured connection is also given in the first parameter (see SSL_CONNECT).

The function call was successful if the return code is 0. All other return codes indicate an error.

SSL_ACCEPT

Waiting for a client connection.
This function can take a timeout value (keyword 'TIMEOUT=', value in seconds) for the local listener in parameter 1.

The function call was successful if the return code is 0. All other return codes indicate an error.

SSL_PENDING

The number of bytes that can be received without blocking is determined.

The function call was not successful if the return code is negative. Otherwise the return code is the number of receivable bytes.

SSL_SHUTDOWN

The connection is closed and all used resources of the SSL environment are freed.

The function call was not successful if the return code is negative. Otherwise the return code is the number of bytes sent.

SSL_GET_ERROR

Returns the SSL return code in case of a previous SSL problem.

ERR_ERROR_STRING

Return the SSL error message in case of a previous SSL problem.

The function call was successful if the return code is 0. The message is available in the variable ERRORTEXT.

Examples for secured connections:
```
SOCKETCALL('SSL_INITIALIZE')        /* Initialize API                    */
SOCKETCALL('SSL_INITIALIZE', 'stackname') /* Initialize API in z/OS
                                       with special address space        */

SOCKETCALL('SSL_CONNECT', 'HOSTNAME=192.168.0.10, PORT=4444,
      SSLFILE=C:\tcVISION\SSL\client.certdef')
                                    /* Connect with the Listener        */

SOCKETCALL('SSL_WRITE', 'Data to be send.')
                                    /* Send data                      */

SOCKETCALL('SSL_READ', 100)         /* Receive 100 bytes of data         */

SOCKETCALL('SSL_SHUTDOWN')          /* Close connection                  */
SOCKETCALL('SSL_LISTEN', 'HOSTNAME=192.168.0.10, PORT=4444,
      SSLFILE=TCVISION.MACLIB(CERTSRVR)')
                                    /* Prepare for incoming inquiries    */

SOCKETCALL('SSL_ACCEPT', 'TIMEOUT=10')
                                    /* Wait for a client connection
                                       with a timeout of 10 seconds */

SOCKETCALL('SSL_PENDING')           /* Determine the number of receivable
                                       bytes                            */
```

4.91 SOURCELINE( )

```
SOURCELINE([lineno])
```

If *lineno* (must be a positive whole number) is specified, this function will return a string containing a copy of the script source code on that line. If *lineno* is greater than the number of lines in the script source code, an error is reported.
If *lineno* is unspecified, the number of lines in the script source code is returned.

Examples:

```
SOURCELINE()                        '123' /* Maybe */
SOURCELINE(1)                       '/* This script calculates... */'
SOURCELINE(23)                      'DO counter = 1 TO limit /* Example */'
```

4.92 SPACE( )

```
SPACE(string[,[length][,padchar]])
```

With only one parameter *string* is returned, stripped of any trailing or leading blanks, and any consecutive blanks inside *string* are translated to a single <space> character (or *padchar* if specified).
Length must be a non-negative whole number. If specified, consecutive blanks within *string* are replaced by exactly *length* instances of <space> (or *padchar* if specified).
padchar will only be used in the output string. In the input string, blanks are still the "magic" characters. So if any *padchar*s exist in *strin*g, they will remain untouched and will not affect the spacing.

Examples:

```
SPACE(' Hello World ')                   'Hello World'
SPACE(' Hello World ',2)                 'Hello  World'
SPACE(' Hello World ',,'*')              'Hello*World'
SPACE(' Hello World ',3, '-')            'Hello---World'
SPACE(' Hello World ',,'o')              'HellooWorld'
```

4.93 STORAGEINFO( )

```
STORAGEINFO( )
```

On mainframe systems, the following storage information is returned:

```
Free(n1) Used(n2) HWM(n3) MCL(n4)
```

whereas
n1 = CURRENT FREE MEMORY SPACE
n2 = CURRENT USED MEMORY SPACE
n3 = HIGH WATER MARK, MAX. EVER USED SO FAR
n4 = MAXIMUM CONTIGUOUS FREE SPACE

4.94 STREAM( )

```
STREAM(streamid[,option[,command]])
```

The function provides a general mechanism for doing operations on streams, or more simply: with files.
The *streamid* identifies a stream (file). The content and format of this string depends on the platform.
The *option* selects one of several operations STREAM() should perform. The possible operations are:

[D] (Description)
Returns a description of the state of *streami*d.

[S] (Status)
Returns a state describing the state of *streami*d. The state can be one of the following: ERROR, NOTREADY, READY, and UNKNOWN.
[C] (Command)
If this option is selected, a third parameter must be present – *comman*d, which is the command to be performed on the stream. Commands consist of one or more space-separated words. The following values can be used as valid *commands* for *streamid*:

[READ]
Open for read access. The file pointer is positioned at the start of the file and only read operations are allowed.
[WRITE]
Open for write access. Positions the current write position at the end of the file. An error is returned if it is not possible to get appropriate access.
[APPEND]
Open for append access. Positions the current write position at the end of the file.
[UPDATE]
Open for update access. Positions the current write position at the end of the file.
[CREATE]
Open for write access. Positions the current write position at the start of the file.
[CLOSE]
Close the stream (file).
[FLUSH]
Flush any pending write to the stream.
[STATUS]

Returns tcSCRIPT status information about the stream in readable form.
[FSTAT]
Returns status information from the operating system about the stream. This consists of at least eight words:
Device Number
Under Windows this represents the disk number, with 0 being Drive A.
Inode Number
Under Windows this is zero.
Permissions
User/Group/Other permissions mask. Consists of three octal numbers with 4 representing read, 2 representing write, and 1 representing execute. Therefore a value of 750 is read/write/execute for user, read/execute for group, and no permissions for other.
Number Links
Under Windows this will always be 1.
User Name
The owner of the stream. Under Windows this will always be “USER”.
Group Name
The group owner of the stream. Under Windows this will always be “GROUP”.
Size
Size of stream in bytes
Stream Type
One or more of the following:
RegularFile
A normal file
Directory
A directory
BlockSpecial
A block special file
FIFO
Usually a pipe
SymbolicLink
A symbolic link
Socket
A socket
SpecialName
A named special file

If the stream is a symbolic link, the returned details are details about the link and not about the file to which the link points.
[RESET]
Resets the stream after an error. Only resettable streams can be reset.
[READABLE]
Returns 1 if the stream is readable by the user, otherwise 0.
[WRITABLE]
Returns 1 if the stream is writable by the user, otherwise 0.
[EXECUTABLE]
Returns 1 if the stream is executable by the user, otherwise 0.
[QUERY]
Returns information about the named stream. If the named stream does not exist, the empty string is returned. This command is further broken down into the following sub-commands:

DATETIME
Returns the date and time of the last modification of the stream in US date format: MM-DD-YY HH:MM:SS.

EXISTS
Returns the fully qualified file name of the specified stream.
HANDLE
Returns the internal file handle of the stream. This will only return a valid value if the stream was opened explicitly or implicitly by tcSCRIPT.
SEEK READ CHAR
Returns the current read position of the open stream expressed in characters.
SEEK READ LINE
Returns the current read position of the open stream expressed in lines.
SEEK WRITE CHAR
Returns the current write position of the open stream expressed in characters.
SEEK WRITE LINE
Returns the current write position of the open stream expressed in lines.
SEEK SYS
Returns the current read position of the open stream as the operating reports it. This is expressed in characters.
SIZE
Returns the size of the persistent stream expressed in characters.
STREAMTYPE
Returns the type of the stream. TRANSIENT, PERSISTENT, or UNKNOWN is returned.
TIMESTAMP
Returns the date and time of the last modification of the stream. The format of the string returned is YYYY-MM-DD HH:MM:SS.

DELETE
Deletes the file in the local file system.

POSITION can be used in place of **SEEK** in the above options.

[OPEN]
Opens the stream in the specified optional mode. If no optional mode is specified, the default is **OPEN BOTH**.
READ
The file pointer will be positioned at the start of the file. Only read operations are allowed.
WRITE
Open for write access. Positions the current write pointer at the end of the file. On platforms where it is not possible to open a file for write without also allowing reads, the read pointer will be positioned at the start of the file.

BOTH
Open for read and write access. Positions the current read pointer at the start of the file and the current write pointer at the end of the file.
WRITE APPEND
Open for write access. Positions the write pointer at the end of the file. On platforms where it is not possible to open a file for write without also allowing reads the read, the pointer will be positioned at the start of the file.
WRITE REPLACE
Open for write access. Positions the current write position at the start of the file. On platforms where it is not possible to open a file for write without also allowing reads, the read pointer will be positioned at the start of the file. This operation will clear the contents of the file.
BOTH APPEND
Open for read and write access. Positions the current read position at the start of the file and the current write position at the end of the file.

BOTH REPLACE

Open for read and write access. Positions both the current read and write pointers at the start of the file.

[SEEK *position* READ|WRITE [CHAR|LINE]]

Positions the file's read or write pointer in the file to the specified *position*. **SEEK** is a synonym for **POSITION**.
A position can be of the following forms. [*relative*]*offset*. *relative* can be one of:

=	The file pointer is moved to the specified *offset* relative to the start of the file. This is the default.
<	The file pointer is moved to the specified *offset* relative to the end of the file.
-	The file pointer is moved backwards relative to the current position.
+	The file pointer is moved forwards relative to the current position

offset is a positive whole number.

READ The read file pointer is positioned.
WRITE The write file pointer is positioned.
CHAR The *offset* specified in *position* above is in terms of characters.
LINE The *offset* specified in *position* above is in terms of lines.

Examples:

Assume a file; '/home/jeff/myfile' last changed March 17th 2014 at 14:37:32, with 100 lines, each line 10 characters long, and the following commands executed in sequence.

```
STREAM('myfile','C','QUERY EXISTS')       '/home/jeff/myfile'
STREAM('myfile','C','QUERY SIZE')         1000
STREAM('myfile','C','QUERY TIMESTAMP')    '2014-03-17 14:37:32'
STREAM('myfile','C','QUERY DATETIME')     '03-17-14 14:37:32'
STREAM('myfile','D')                      ''
STREAM('myfile','S')                      'UNKNOWN'
STREAM('myfile','C','QUERY SEEK READ')    'UNKNOWN'
STREAM('myfile','C','OPEN READ')          'READY:'
STREAM('myfile','D')                      ''
STREAM('myfile','S')                      'READY'
STREAM('myfile','C','QUERY SEEK READ')    1
STREAM('myfile','C','CLOSE')              'UNKNOWN'
STREAM('myfile','C','STATUS')             'UNKNOWN'
STREAM('myfile','C','FSTAT')              '10 25012 600 1 jeff
                                           jeff 1000 RegularFile'
STREAM('myfile','C','READABLE')           1
STREAM('myfile','C','WRITABLE')           1
STREAM('myfile','C','EXECUTABLE')         0
```

4.95 STRIP( )

```
STRIP(string[,[option][,char]])
```

Returns *string* after possibly stripping it of any number of leading and/or trailing characters. The default action is to strip off both leading and trailing blanks. If *char* (which must be a string containing exactly one character) is specified, that character will be stripped off instead of blanks. Blanks within a word (or *char*s, if defined, that are neither leading nor trailing) are untouched.

If *option* is specified, it will define what to strip. The possible values for *option* are:

[L] (Leading)
Only strip off leading blanks, or *char*s if specified.
[T] (Trailing)
Only strip off trailing blanks, or *char*s if specified.
[B] (Both)
Combine the effect of L and T: Strip off both leading and trailing blanks, or *char*s if specified. This is the default action.

Examples:

```
STRIP(' Hello World ')            'Hello World'
STRIP(' Hello World ','L')        'Hello World '
STRIP(' Hello World ','t')        ' Hello World'
STRIP(' Hello World ','Both')     'Hello World'
STRIP('0.1234500',,'0')           '.12345'
STRIP('0.1234500 ',,'0')          '.1234500 '
```

4.96 SUBSTR( )

```
SUBSTR(string,start[,[length][,padchar]])
```

Returns the substring of *string* that starts at *start*, and has the length *length*. *Length* defaults to the rest of the string. *Start* must be a positive whole number, while *length* can be any non-negative whole number.
start can be larger than the length of *string*. If *length* is specified and the sum of *length* and *start* minus 1 is greater than the length of *string*, the result will be padded with *padchar*s to the specified length. The default value for *padchar* is the <space> character.

Examples:

```
SUBSTR('Hello World',3)           'llo World'
SUBSTR('Hello World',3,3)         'llo'
SUBSTR('Hello World',6,7)         ' World '
SUBSTR('Hello World',6,7,'*')     ' World*'
SUBSTR('Hello World',12,3,'*')    '***'
```

4.97 SUBWORD( )

SUBWORD(*string*,*start*[,*length*])

Returns the part of *string* that starts at a blank-delimited word *start* (must be a positive whole number). If *length* (must be a non-negative whole number) is specified, that number of words is returned. The default value for *length* is the rest of the string.

It is not an error to specify *length* to refer to more words than *string* contains, or for *start* and *length* together to specify more words than *string* holds. The result string will be stripped of any leading and trailing blanks but blanks within a word will be preserved as is.

Examples:

```
SUBWORD('A danger foreseen is a danger avoided',4)   'is a danger avoided'
SUBWORD('A danger foreseen is a danger avoided',4,2) 'is a'
SUBWORD('A danger foreseen is a danger avoided',4,5) 'is a danger avoided'
SUBWORD('A danger foreseen is a danger avoided',1,3) 'A danger foreseen'
```

4.98 SYMBOL( )

```
SYMBOL(name)
```

Checks whether the string *name* is a valid symbol (a positive number or a possible variable name) and returns a three-letter string indicating the result of that check. If *name* is a symbol and a currently set variable, VAR is returned. If *name* is a valid symbol name but has not been given a value (or is a constant symbol which cannot be used as a variable name), LIT is returned to signify that it is a literal. If *name* is not a valid symbol name, the string BAD is returned.

Examples:

```
Drop A.3;
J=3
SYMBOL('J')                     'VAR'
SYMBOL(J)                       'LIT'
SYMBOL('a.j')                   'LIT'
A.3 = 5
SYMBOL('a.j')                   'VAR'
SYMBOL(2)                       'LIT'
SYMBOL('*')                     'BAD'
SYMBOL('Hallo World')           'BAD'
SYMBOL('.Hello->World')         'BAD'
```

4.99 TIME( )

```
TIME([option_out [,time [option_in]]])
```

Returns a string containing information about the time. To get the time in a particular format, *option_out* can be specified. The default *option_out* is Normal. The meaning of the possible options is:

[C] (Civil)
Returns the time in civil format. The return value might be "hh:mmXX". XX is either am or pm. The hh part will be stripped of any leading zeros and is in the range 1-12.
[E] (Elapsed)
Returns the time elapsed in seconds since the internal stopwatch was started. The result will not have any leading zeros or blanks. The output will be a floating-point number with six digits after the decimal point.

[H] (Hours)
Returns the number of complete hours passed since last midnight in the form "hh". The output has no leading zeros and is in the range 0-23.
[L] (Long)
Returns the exact time down to the microsecond. This is called the long format. The output might be "hh:mm:ss.mmmmmm". Be aware that most computers do not have a clock of that accuracy, so the actual granularity you can expect will be about a few milliseconds. The hh, mm, and ss parts will be identical to what is returned by the options H, M, and S respectively except that each part will have leading zeros as indicated by the format.
[M] (Minutes)
Returns the number of complete minutes since midnight in a format without leading zeros. Range: 0-1440
[N] (Normal)
The output format is "hh:mm:ss"and will contain the hours, minutes, and seconds respectively. Each part will be padded with leading zeros to make it double-digit.
[R] (Reset)
Returns the value of the internal stopwatch just like the E option using the same format. It also resets the stopwatch.
[S] (Seconds)
Returns the number of complete seconds since midnight in a format without leading spaces. Range: 0-86400
[T] *(*time_t)
Returns the current date/time in UNIX *time_t* format. *time_t* is the number of seconds since January 1st 1970.

The time is never rounded, only truncated.

Examples:

Assuming that the time is exactly 16:46:27.753863 on March 17th 2014, the following is true:

```
TIME('C')                           '4:46pm'
TIME('E')                           '0.01200'   /* as an example */
TIME('H')                           '16'
TIME('L')                           '16:46:27.753863'
TIME('M')                           '1006'
TIME('N')                           '16:46:27'
TIME('R')                           '0.430221'   /* as an example */
TIME('S')                           '60387'
```

If the *time* option is specified, the time is converted. The optional *option_in* specifies the format in which *time* is supplied. The possible values for o*ption_in* are: **CHLMNS.** The default value for *option_in* is N.

Examples:

TIME('C','12:34:56')	'12:34pm'
TIME('N','12:34pm','C')	'12:34:00'

4.100 TRANSLATE( )

```
TRANSLATE(string[,[tableout][,[tablein][,padchar]]])
```

Performs a translation on the characters in *string*. If neither *tablein* nor *tableout* is specified, it translates *string* from lowercase to uppercase. Note that this operation may depend on the language setting. Two translation tables might be specified as the strings *tablein* and *tableout*. If one or both of the tables are specified, each character in *string* that exists in *tablein* is translated to the character in *tableout* that occupies the same position as the character did in *tablein*. The *tablein* defaults to the whole character set (all 256) in numeric sequence, while *tableout* defaults to an empty set. Characters that are not in *tablein* are left unchanged.
If *tableout* is larger than *tablein*, the extra entries are ignored. If it is smaller than *tablein,* it is padded with *padchar* to the correct length. *Padchar* defaults to <space>.
If a character occurs more than once in *tablein*, only the first occurrence matters.

Examples:

```
TRANSLATE('HelloWorld')                                        'HELLOWORLD'
TRANSLATE('HelloWorld','AEHLODWaehloDw','aehlodwAEHLODW')      'hELLOwOrLD'
TRANSLATE('HelloWorld','aehlodw')                              '          '
TRANSLATE('HelloWorld','aehlodw',,'#')                         '##########'
```

4.101 TRIM( )

```
TRIM(string)
```

Removes trailing blanks from the string argument.

Example:

```
TRIM(' Hello World ')                          ' Hello World'
```

4.102 TRUNC( )

```
TRUNC(number[,length])
```

Returns *number* truncated to the number of decimals specified by *length*. *Length* defaults to 0, that is, returns a whole number without decimal part. The decimal point will only be present if it is a non-empty decimal part, i.e. *length* is non-zero. The number will always be returned in a simple form and never in an exponential form, no matter what the current settings of *NUMERIC* might be. If *length* specifies more decimals than *number* has, extra zeros are appended. If *length* specifies fewer decimals than *number* has, the number is truncated. Note that *number* is never rounded, except for the rounding that might take place during normalization.

Examples:

```
TRUNC(1.23)          '1'
TRUNC(1.99)          '1'
TRUNC(1.23,3)        '1.230'
TRUNC(1.234,2)       '1.23'
```

4.103 TSTP( )

```
TSTP()
```

Returns the current date and time as ISO timestamp.

Example:

```
TSTP()                                  '2010-03-01.17.45.56.556286'
```

4.104 TSTPDIFF( )

```
TSTPDIFF(timestamp1,timestamp2)
```

Returns the difference between two ISO timestamps.

Examples:

```
TSTPDIFF('2010-03-01.16.45.56.556286', '2010-03-01.17.45.56.556286')
                                        '001.00.00.000000'

TSTPDIFF('2010-03-01.16.45.56.556286', '2010-03-02.16.45.56.556286')
                                        '024.00.00.000000'
```

4.105 TSTPUTC( )

```
TSTPUTC()
```

Returns the current UTC date and UTC time as ISO timestamp.

Example:

```
TSTPUTC()                               '2010-03-01.16.45.56.556286'
```

4.106 UNLINK( )

```
UNLINK( filename )
```

Deletes the file referred to with *filename*. If the file could be deleted, 0 is returned. Otherwise a string with error number and error text is returned.

Examples:

```
UNLINK('/tmp/myfile')                   0
UNLINK('/tmp/myfile')                   '2, No such file or directory'
```

4.107 UPPER( )

```
UPPER(string)
```

Translates the strip to uppercase. The action of this function is equivalent to that of TRANSLATE(string), but it is slightly faster for short strings.

Example:

```
UPPER('Hello World')                'HELLO WORLD'
```

4.108 UTF8A2E( )

```
A2E(expression)
```

Converts the result of *expression* from ASCII to EBCDIC. If the *expression* contains characters that are coded based on UTF8, they will be treated as single byte characters.

Example:

```
IF UTF8A2E(substr(FLDA,1,1) = FIELD_EBCDIC
```

4.109 UTF8CHAR( )

```
UTF8CHAR('string')
```

This function only applies to mainframe systems.
The function eliminates the translation from UTF8 to EBCDIC for character string *string*. This function is ignored when executed on a Workstation Agent. The result of the function is also a *string*.

Examples:

```
RESULT = UTF8CHAR("SELECT F1, F2 FROM A.TABLE WHERE F1 = '漢字'")
RESULT = UTF8CHAR("SELECT F1, F2 FROM A.TABLE WHERE F2 LIKE '%Ü%'")
```

4.110 UTF8LEFT( )

```
UTF8LEFT(string,length[,padchar])
```

The function treats characters that are coded based on UTF-8 like single-byte characters.
Returns the *length* leftmost characters in *string*. If *length* (must be a non-negative whole number) is greater than the length of *string*, the result is padded on the right with <space> (or *padchar* if that is specified) so it has the correct length.

Examples:

```
UTF8LEFT('Hello World',5)               'Hallo'
UTF8LEFT('Hello World',3)               'Hal'
UTF8LEFT('Hello World',13)              'Hello World  '
UTF8LEFT('Hello World',13,'*')          'Hello World**'
```

4.111 UTF8LENGTH( )

```
UTF8LENGTH(string)
```

The function treats characters that are coded based on UTF-8 like single-byte characters.

Returns the number of characters in *string*.

Examples:

```
UTF8LENGTH('')                           0
UTF8LENGTH('Hello')                      5
UTF8LENGTH('Hello World')                11
UTF8LENGTH(' Hello World  ')             14
```

4.112 UTF8RIGHT( )

```
UTF8RIGHT(string,length[,padchar])
```

The function treats characters that are coded based on UTF-8 like single-byte characters.

Returns the *length* rightmost characters in *string*. If *length* (must be a non-negative whole number) is greater than the length of *string*, the result is padded on the left with the necessary number of *padchar*s to make it as long as *length* specifies. *Padchar* defaults to <space>.

Examples:

```
UTF8RIGHT('Hello World',5)               ' World'
UTF8RIGHT('Hello World',3)               'rld'
UTF8RIGHT('Hello World',13)              '  Hello World'
UTF8RIGHT('Hello World',13,'*')          '**Hello World'
```

4.113 UTF8SUBSTR( )

```
UTF8SUBSTR(string,start[,[length][,padchar]])
```

The function treats characters that are coded based on UTF-8 like single-byte characters.
Returns the substring of *string* that starts at *start* and has the length *length*. *Length* defaults to the rest of the string. *Start* must be a positive whole number, while *length* can be any non-negative whole number.
It is not an error for *start* to be larger than the length of *string*. If *length* is specified and the sum of *length* and *start* minus 1 is greater than the length of *string*, the result will be padded with *padchar*s to the specified length. The default value for *padchar* is the <space> character.

Examples:

```
UTF8SUBSTR('Hello World',4)              'lo World'
```

```
UTF8SUBSTR('Hello World',5,3)           'o W'
UTF8SUBSTR('Hello World',6,7)           ' World '
UTF8SUBSTR('Hello World',6,7,'*')       ' World*'
UTF8SUBSTR('Hello World',12,3,'*')      '***'
```

4.114 VALUE( )

```
VALUE(symbol[,[value],[pool]])
```

This function expects string *symbol* as first parameter, which names an existing variable. The result returned from the function is the value of that variable. If *symbol* does not name an existing variable, the default value is returned and the NOVALUE condition is not raised. If *symbol* is not a valid symbol name and this function is used to access a normal tcSCRIPT variable, an error occurs.
If the optional second parameter is specified, the variable will be set to that value after the old value has been extracted.
The optional parameter *pool* might be specified to select a particular pool of variables to search for *symbo*l. The default is to search in the variables at the current procedural level. The available *pools* depend on the implementation but typically variables can be set in application programs or in the operating system.

Examples:

```
var= 'Hello'
VALUE('VAR')                            'Hello'
VALUE('VAR','World')                    'Hello'
VALUE('VAR')                            'World'
VALUE('VAR','HelloWorld','SYSTEM')      ''
VALUE('VAR',,'SYSTEM')                  'HelloWorld'
Drop A3
A33=7
K=3
VALUE('a'k)                             'A3'
VALUE('a'k||k)                          '7'
```

Please note the so-called "double-expansion" effect. If necessary, the first parameter must be specified with leading quotes.

Examples:

```
K=3
var='K'
VALUE('var')                            'K'
VALUE(var)                              '3'
```

4.115VERIFY( )

```
VERIFY(string,ref[,[option][,start]])
```

With only the first two parameters, it will return the position of the first character in *string* that is not also a character in the string *ref*. If all characters in *string* are also in *ref*, it will return 0. If *option* is specified, it can be one of the following:

[N] (Nomatch)
The result will be the position of the first character in *string* that does exist in *ref*, or zero if all exist in *ref*. This is the default option.
[M] (Match)
Reverses the search and returns the position of the first character in *string* that exists in *ref*. If none exists in *ref*, zero is returned.

If *start* (must be a positive whole number) is specified, the search will start at that position in *strin*g. The default value for *start* is 1.

Examples:

```
VERIFY('hello','ehlo')                  0
VERIFY('hello','ehlo','M')              1
VERIFY('hello','eho','N')               3
VERIFY('hello','eho','N',3)             3
VERIFY('hello','hlo','N',3)             0
```

4.116WORD( )

```
WORD(string,wordno)
```

Returns the blank-delimited word number *wordno* from the string *strin*g. If *wordno* (must be a positive whole number) refers to a non-existing word, a null string is returned. The result will be stripped of any blanks.

Examples:

```
WORD('A danger foreseen is a danger avoided',3)      'foreseen'
WORD('A danger foreseen is a danger avoided',4)      'is'
WORD('A danger foreseen is a danger avoided',9)      ''
```

4.117WORDINDEX( )

```
WORDINDEX(string,wordno)
```

Returns the character position of the first character of the blank-delimited word number *wordno* in *strin*g, which is interpreted as a string of blank-delimited words. If *number* (must be a positive whole number) refers to a word that does not exist in *strin*g, *0* is returned.

Examples:

```
WORDINDEX('A danger foreseen is a danger avoided',3)       10
WORDINDEX('A danger foreseen is a danger avoided',4)       19
WORDINDEX('A danger foreseen is a danger avoided',8)       0
```

4.118 WORDLENGTH( )

```
WORDLENGTH(string,wordno)
```

Returns the number of characters in blank-delimited word number *number* in *strin*g. If *number* (must be a positive whole number) refers to a non-existent word, 0 is returned. Trailing or leading blanks do not count when calculating the length.

Examples:

```
WORDLENGTH('A danger foreseen is a danger avoided',3)      8
WORDLENGTH('A danger foreseen is a danger avoided',4)      2
WORDLENGTH('A danger foreseen is a danger avoided',0)      0
```

4.119WORDPOS( )

```
WORDPOS(phrase,string[,start])
```

Returns the word number in *string* at which phrase begins – provided that *phrase* is a subphrase of *strin*g. If not, 0 is returned to indicate that the phrase was not found. A phrase differs from a substring in one significant way: A phrase is a set of words separated by any number of blanks.
For instance, "is a" is a subphrase of "This is a phrase". Notice the different amount of white space between "is" and "a". If *start* is specified, it sets the word in *string* at which the search starts. The default value for *start* is 1.

Examples:

```
WORDPOS('foreseen is','A danger foreseen is a danger avoided')         3
WORDPOS('avoided','A danger foreseen is a danger avoided')             7
WORDPOS('is a','A danger foreseen is a danger avoided')                4
WORDPOS('danger   foreseen','A danger foreseen is a danger avoided',1) 2
```

4.120 WORDS( )

```
WORDS(string)
```

Returns the number of blank-delimited words in the *strin*g.

Examples:

```
WORDS('A danger foreseen is a danger avoided') 7
WORDS('Hello world')                           2
WORDS('')                                      0
```

4.121 WTO( )

```
WTO(string)
```

Writes the contents of *string* to the operating system console.

Examples:

```
WTO('A danger foreseen is a danger avoided')
WTO('Hello world')
```

4.122 X2B( )

```
X2B(hexstring)
```

Translate *hexstring* to a binary string. Each hexadecimal digit in *hexstring* is translated to four binary digits in the result. There are blanks in the result.

Examples:

```
X2B('')                     ''
X2B('48616C6C6F 57656C74')  '010010000110000101101100011011000110111101010111011001010110110001110100'
X2B('57 65 6C 74')          '01010111011001010110110001110100'
```

4.123 X2C( )

```
X2C(hexstring)
```

Returns the (packed) string representation of *hexstrin*g. The *hexstring* will be converted byte wise and blanks may optionally be inserted into the *hexstring* between pairs or hexadecimal digits to divide the number into groups and improve readability. All groups must have an even number of hexadecimal digits, except for the first group. If the first group has an odd number of hexadecimal digits, it is padded with an extra leading zero before conversion.

Examples:

```
X2C('')                         ''
X2C('48656C6C6F 576F726C64')    'HelloWorld'
X2C('57 6F 72 6C 64')           'World'
```

4.124 X2D( )

```
X2D(hexstring[,length])
```

Returns a whole number that is the decimal representation of *hexstrin*g. If *length* is specified, *hexstring* is interpreted as a complement of two hexadecimal number consisting of the *number*

rightmost hexadecimal numerals in *hexstring*. If *hexstring* is shorter than *number*, it is padded on the left with <NUL> characters (that is:x'00').
If *length* is not specified, *hexstring* will always be interpreted as an unsigned number. Otherwise it is interpreted as a signed number and the leftmost bit in *hexstring* decides the sign.

Examples:

```
X2D('01 23')                                  291
X2D('0234')                                   564
X2D('ffff')                                   65535
X2D('ffff',5)                                 65535
X2D('ffff',4)                                 -1
X2D('ff80',3)                                 -128
X2D('56234',3)                                564
```

4.125 XRANGE( )

```
XRANGE([start][,end])
```

Returns a string that consists of all the characters from *start* to and including *end*. The default value for character *start* is X'00', while the default value for character *end* is x'ff'. Without any parameters, the whole character set in "alphabetic" order is returned. Note that the representation of the output from XRANGE() depends on the character set used by the computer.
If the value of *start* is larger than the value of *end*, the output will wrap around from x'ff' to x '00'.
If *start* or *end* is not a string containing exactly one character, an error is reported.

Examples:

```
XRANGE('a','e')                    'abcde'
XRANGE('F0'x)                      'F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF'x
XRANGE(,'0F'x)                     '000102030405060708090A0B0C0D0E0F'x
XRANGE('F8'x,'07'x)                'F8F9FAFBFCFDFEFF0001020304050607'x
```

4.126 XWTO( )

```
XWTO( connection, string )
connection is defined as:
HOSTNAME=ip-addr/name,PORT=port[,AGENT=ip-addr/name])
        [,USER=userid,PWD=password][,SSL={Y|N}[,ssldef]]
```

Sends a message *string* to a remote Agent. When using AGENT=, the request is forwarded to the named Agent. When using SSL=Y, an encrypted connection is established. The required certificate information is to be provided with *ssldef* (refer to SOCKETCALL - SSL_CONNECT function).

Examples:

```
XWTO('HOSTNAME=192.168.0.124,PORT=4120','Everything is OK')
```

Sends message directly to another Agent

```
XWTO('HOSTNAME=localhost,PORT=4120,AGENT=TVSM-z/OS16','EVERYTHING IS OK')
```

Sends message to another Agent via the local Agent
Both Agents must be part of the same Agent network

5 Copyright

License information on third-party components used can be found in the manual ‘Copyright’.