



Caché Multi Value Basic Reference

5.2.3MV
21 February 2007

Caché MultiValue Basic Reference
Caché 5.2.3MV 21 February 2007
Copyright © 2007 InterSystems Corporation.
All rights reserved.

This book was assembled and formatted in Adobe Page Description Format (PDF) using tools and information from the following sources: Sun Microsystems, RenderX, Inc., Adobe Systems, and the World Wide Web Consortium at www.w3c.org. The primary document development tools were special-purpose XML-processing applications built by InterSystems using Caché and Java.



Caché WEBLINK, Distributed Cache Protocol, M/SQL, N/NET and M/PACT are registered trademarks of InterSystems Corporation.



InterSystems Jalapeño Technology, Enterprise Cache Protocol, ECP, and InterSystems Zen are trademarks of InterSystems Corporation.

All other brand or product names used herein are trademarks or registered trademarks of their respective companies or organizations.

This document contains trade secret and confidential information which is the property of InterSystems Corporation, One Memorial Drive, Cambridge, MA 02142, or its affiliates, and is furnished for the sole purpose of the operation and maintenance of the products of InterSystems Corporation. No part of this publication is to be used for any other purpose, and this publication is not to be reproduced, copied, disclosed, transmitted, stored in a retrieval system or translated into any human or computer language, in any form, by any means, in whole or in part, without the express prior written consent of InterSystems Corporation.

The copying, use and disposition of this document and the software programs described herein is prohibited except to the limited extent set forth in the standard software license agreement(s) of InterSystems Corporation covering such programs and related documentation. InterSystems Corporation makes no representations and warranties concerning such software programs other than those set forth in such standard software license agreement(s). In addition, the liability of InterSystems Corporation for any losses or damages relating to or arising out of the use of such software programs is limited in the manner set forth in such standard software license agreement(s).

THE FOREGOING IS A GENERAL SUMMARY OF THE RESTRICTIONS AND LIMITATIONS IMPOSED BY INTERSYSTEMS CORPORATION ON THE USE OF, AND LIABILITY ARISING FROM, ITS COMPUTER SOFTWARE. FOR COMPLETE INFORMATION REFERENCE SHOULD BE MADE TO THE STANDARD SOFTWARE LICENSE AGREEMENT(S) OF INTERSYSTEMS CORPORATION, COPIES OF WHICH WILL BE MADE AVAILABLE UPON REQUEST.

InterSystems Corporation disclaims responsibility for errors which may appear in this document, and it reserves the right, in its sole discretion and without notice, to make substitutions and modifications in the products and practices described in this document.

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Customer Support

Tel: +1 617 621-0700
Fax: +1 617 374-9391
Email: support@InterSystems.com

Table of Contents

Symbols	1
Symbols Used in Caché MVBasic	1
Caché MultiValue Basic Commands	5
ABORT, ABORTE, ABORTM	5
BEGIN TRANSACTION	6
CALL	7
CASE	10
CHAIN	11
CLEARDATA	12
CLEARFILE	13
CLOSE	14
CLOSESEQ	15
COMMIT	15
COM (COMMON)	17
CONTINUE	18
CONVERT	19
\$COPYRIGHT	21
CREATE	21
CRT	23
DATA	24
DEBUG	25
DEL	26
DELETE, DELETEU	27
DIM (DIMENSION)	28
DISPLAY	30
ECHO	31
END	32
END TRANSACTION	32
ERRMSG	33
EXECUTE	34
EXIT	37
FIND	38
FINDSTR	40

FOOTING	42
FOR...NEXT	44
FUNCTION	46
GOSUB	48
GOTO	50
HEADING	51
IF...THEN...ELSE	54
INPUT	56
INS	58
LET	59
LOCATE	61
LOCK	63
LOOP...REPEAT	64
NAP	65
ON	66
ON ERROR GOTO	67
OPEN	68
OPENPATH	70
OPENSEQ	71
PERFORM	73
PRECISION	74
PRINT	75
PRINTER	76
PROG (PROGRAM)	77
PROMPT	77
RANDOMIZE	78
READ, READU, READV, READVU	79
READBLK	81
READLIST	83
READNEXT	84
READSEQ	86
RELEASE	88
REM	88
REMOVE	89
RETURN	91
REVREMOVE	92
ROLLBACK	93

RQM	94
SEEK	95
SELECT	96
SELECTE	98
SELECTV	99
SLEEP	100
STATUS	101
STOP	102
SUBROUTINE	103
SWAP	104
TRANSACTION ABORT	105
TRANSACTION COMMIT	106
TRANSACTION START	107
UNLOCK	108
WEOFSEQ	109
WRITE, WRITEU, WRITEV, WRITEVU	110
WRITEBLK	112
WRITESEQ	114
Caché MultiValue Basic Functions	117
ABS	117
ABSS	118
ACOS	119
ADDS	120
ALPHA	121
ANDS	122
ASCII	123
ASIN	124
ASSIGNED	125
ATAN	126
BITAND	127
BITNOT	128
BITOR	130
BITRESET	132
BITSET	133
BITTEST	135
BITXOR	136

CATS	138
CHANGE	139
CHAR	140
CHARS	142
CHECKSUM	143
COL1	144
COL2	145
CONVERT	146
COS	148
COSH	149
COUNT	150
COUNTS	151
DATE	152
DCOUNT	153
DELETE	154
DIV	155
DIVS	156
DOWNCASE	157
DQUOTE	158
DTX	159
EBCDIC	160
EQS	161
EREPLACE	162
EXP	163
EXTRACT	165
FADD	166
FDIV	168
FIELD	169
FIELDS	171
FIELDSTORE	173
FILEINFO	175
FIX	177
FMT	178
FMTS	179
FMUL	181
FSUB	182
GES	183

GETREM	184
GROUP	186
GTS	188
ICONV	189
ICONVS	191
INDEX	192
INDEXS	193
INDICES	194
INMAT	195
INSERT	196
ITYPE	197
KEYIN	198
LEFT	199
LEN	200
LENS	201
LES	202
LN	203
LOWER	204
LTS	205
MAXIMUM	206
MINIMUM	207
MOD	208
MODS	209
MULS	210
NEG	211
NES	212
NOT	213
NOTS	214
NUM	215
NUMS	216
OCONV	217
OCONVS	220
ORS	222
PWR	223
QUOTE	224
RAISE	225
REM	226

REMOVE	227
REPLACE	229
REUSE	230
RIGHT	232
RND	233
SADD	234
SCMP	235
SDIV	236
SEQ	238
SEQS	239
SIN	240
SINH	241
SMUL	242
SPACE	244
SPACES	245
SPLICE	246
SPOOLER	247
SQRT	249
SQUOTE	250
SSUB	251
STATUS	252
STR	253
STRS	254
SUBR	255
SUBS	256
SUBSTRINGS	257
SUM	258
SUMMATION	259
TAN	260
TANH	261
TIME	262
TIMEDATE	263
TRIM	264
TRIMB	266
TRIMBS	267
TRIMF	268
TRIMFS	269

UNASSIGNED	270
UNICHAR	271
UNICHARS	272
UNISEQ	273
UNISEQS	275
UPCASE	276
XTD	277
Caché MultiValue Basic General Concepts	279
Comments	279
Dynamic Arrays	280
Labels	282
MATCH Pattern Matching	283
MultiValue Files	285
Operators	286
Strings	289
Variables	290

Symbols

Symbols Used in Caché MVBasic

A table of characters used in Caché MVBasic as operators, etc.

Table of Symbols

The following are the literal symbols used in Caché MVBasic. (This list does not include symbols indicating format conventions, which are not part of the language.) There is a separate table for symbols used in Caché ObjectScript.

The name of each symbol is followed by its ASCII decimal code value.

Symbol	Name and Usage
[space] or [tab]	<i>White space (Tab (9) or Space (32))</i> : One or more whitespace characters between keywords, identifiers, and variables.
!	<i>Exclamation Mark (33)</i> : Single-line comment indicator. Logical OR operator .
"	<i>Double Quote (34)</i> : Used to enclose string literals . You can use "" to specify an empty string.
#<	<i>Pound, Less than</i> : Less than or equal to operator .
#>	<i>Pound, Greater than</i> : Greater than or equal to operator .
\$	<i>Dollar sign (36)</i> : Permitted character in variable names.
\$*	<i>Dollar sign, Asterisk</i> : A single-line comment indicator.
%	<i>Percent sign (37)</i> : Permitted character in variable names.
&	<i>Ampersand (38)</i> : Logical AND operator .
'	<i>Single Quote (39)</i> : Used to enclose string literals . You can use " to specify an empty string.

Symbol	Name and Usage
()	<p><i>Parentheses (40,41):</i> Used to enclose a procedure or function parameter list.</p> <p>Used to nest expressions; nesting overrides the default order of operator precedence.</p> <p>Used to specify array subscripts.</p> <p>In CALL statement, used to specify argument passed by value.</p>
*	<p><i>Asterisk (42):</i> Multiplication operator.</p> <p>Single-line comment indicator.</p>
**	<p><i>Double Asterisk:</i> Exponentiation operator.</p>
* =	<p><i>Asterisk, Equal:</i> Multiplication assignment operator.</p>
+	<p><i>Plus sign (43):</i> Addition operator.</p>
+ =	<p><i>Plus, Equal:</i> Increment (addition assignment) operator.</p>
,	<p><i>Comma (44):</i> Used to separate parameters in a function parameter list.</p> <p>Used to separate subscripts in an array.</p> <p>With DIM statements, used to separate multiple assignments.</p> <p>With PRINT or CRT statements, inserts a tab between arguments.</p>
-	<p><i>Minus sign (45):</i> Unary arithmetic negative operator.</p> <p>Subtraction operator.</p>
- =	<p><i>Minus, Equal:</i> Decrement (subtraction assignment) operator.</p>
.	<p><i>Period (46):</i> Decimal point character.</p> <p>Permitted character in variable names; cannot be first character.</p>
...	<p><i>Three Periods:</i> MATCH operator pattern match code.</p>
/	<p><i>Slash (47):</i> Division operator</p> <p>In COMMON statement, used to enclose a storage area name. For example: <code>/sharedvars/</code>.</p>

Symbol	Name and Usage
:	<i>Colon (58)</i> : Label suffix . For example, LabelOne: String concatenation operator. With Case function, used to associate case:value pairs.
;	<i>Semicolon (59)</i> : MVBasic shell Basic language command prefix. For example: ;print date() MVBasic statement end indicator before in-line single-line comment . In INSERT and REPLACE functions, an argument separator.
<	<i>Less than (60)</i> : Less than operator .
<=	<i>Less than, Equal</i> : Less than or equal to operator .
<>	<i>Less than, Greater than</i> : Inequality logical operator . For example 3<>4. Used to enclose integers specifying the Field, Value, and Subvalue level when extracting, deleting, or replacing a dynamic array element value. For example: <1,2,2>
=	<i>Equal sign (61)</i> : Equality operator . Assignment operator.
=<	<i>Equal, Less than</i> : Less than or equal to operator .
=>	<i>Equal, Greater than</i> : Greater than or equal to operator .
>	<i>Greater than (62)</i> : Greater than operator .
>=	<i>Greater than, Equal</i> : Greater than or equal to operator .
@	<i>At sign (64)</i> : Prefix for system variable names. A function used with PRINT , CRT , or INPUT to position the cursor on the screen, or control display modes. For example, PRINT @(15):"Over here!" or PRINT @(-5):"Blinking text"
[]	<i>Square Brackets (91 & 93)</i> : Substring extract operator ; brackets enclose integers specifying the substring to extract.
\	<i>Backslash (92)</i> : Used to enclose string literals . Cannot be used in MATCH strings. You can use \\ to specify an empty string. In HEADING or FOOTING , inserts the current time and date.

Symbol	Name and Usage
]	<i>Right Square Bracket (93)</i> : In HEADING or FOOTING , starts a new line.
^	<i>Caret (94)</i> : In HEADING or FOOTING , inserts a page number.
_	<i>Underscore (95)</i> : Line continuation indicator.

Caché MultiValue Basic Commands

ABORT, ABORTE, ABORTM

Terminates program execution and returns to MVBasic shell.

```
ABORTE [errmsgID,param1[,param2[,...]]]
ABORT  [errmsgID,param1[,param2[,...]]]

ABORTM [string]
ABORT  [string]
```

Arguments

<i>errmsgID</i>	The ID of a record in the error message file ERRMSG.
<i>param</i>	A parameter, or comma-separated list of parameters, to the error message.
<i>string</i>	A string to be displayed.

Description

The **ABORT** statements are used to terminate program execution and return to the MVBasic shell programming prompt. If an argument is specified, they use this argument to display an error message before terminating program execution.

- **ABORTE** with an specified argument uses the ERRMSG file to obtain the error message to display.
- **ABORTM** with an specified argument uses the literal *string* as the error message to display.
- **ABORT** may be functionally identical to either **ABORTE** or **ABORTM**, depending on the emulation setting.

ABORT and STOP

The **ABORT** command terminates all program execution and returns to the programming prompt. The **STOP** terminates the executing routine and returns control to the calling routine.

See Also

- **STOP** statement
- Caché ObjectScript: QUIT command

BEGIN TRANSACTION

Begins a transaction.

```
BEGIN TRANSACTION
```

Description

The **BEGIN TRANSACTION** statement initiates a transaction. All subsequent statements are part of this transaction until the transaction is closed, either by a **COMMIT** statement or a **ROLLBACK** statement. After the transaction is closed, program execution continues at the **END TRANSACTION** statement.

Note: Caché MVBasic supports two sets of transaction statements:

- UniVerse-style **BEGIN TRANSACTION**, **COMMIT**, **ROLLBACK**, and **END TRANSACTION**.
- UniData-style **TRANSACTION START**, **TRANSACTION COMMIT**, and **TRANSACTION ABORT**.

These two sets of transaction statements should not be combined.

CAUTION: There is a fundamental difference in the way Caché transactions operate in comparison to most other transaction systems in the MV world. In Caché, items written to a file are immediately available both to the writing process and any other process accessing the file. If the transaction is aborted either programmatically or because of some failure, then the the written item will be rolled back to the state prior to the start of the transaction.

Most other transaction systems in the MV world will make an item written to a file available to the process that wrote the item (in other words, if it reads the item back from the file after the write, it will be given the version that it wrote to the file), but any other process READING the item will see the version of the item as it was before a write. This is generally referred to as the isolation level. This difference may have connotations for systems that wish to scan files without taking locks.

See Also

- [END TRANSACTION](#) statement
- [COMMIT](#) statement
- [ROLLBACK](#) statement

CALL

Transfers control to an external subroutine.

```
CALL name[(argumentlist)]
```

Arguments

<i>name</i>	Name of the external subroutine to call.
<i>argumentlist</i>	<i>Optional</i> — Comma-delimited list of arguments to pass to the external subroutine. The number of arguments specified must match the number of argument defined for the subroutine. Specify the MAT keyword before an array argument.

Description

The **CALL** statement can be used to call an external subroutine and to optionally pass arguments to that subroutine. The external subroutine must have been compiled and cataloged. You can use the **RETURN** statement within the external subroutine to return control to the next statement following the **CALL** statement.

Note: The **RETURN** statement will first return from internal **GOSUB** subroutines and then return from the external **SUBROUTINE** when the **GOSUB** stack is exhausted.

You can use *name* to specify the external subroutine either directly or indirectly:

- The *name* argument can specify the exact name under which the subroutine was cataloged.
- The *name* argument can specify the name of a variable that contains the name of the subroutine. A variable of this type is prefaced with the @ symbol. A variable name can be a local variable, or an element of an array.

The argument list can contain any combination of regular variables and array variables. An array variable must be dimensioned in the calling program using the **DIM** statement. Caché dimensionless arrays can also be passed to the subroutine as arguments, providing they are **DIMENSIONED** using **DIM var()**.

In *argumentlist*, an array variable name must be preceded by the **MAT** keyword. The following is an argument list that specifies a literal, a regular variable, and an array variable:

```
CALL MySub(123,myvar,MAT myarray)
```

By default, all arguments are passed by reference. If the subroutine changes the value of an argument passed by reference, this value is also changed in the calling program. You can specify that an argument is to be passed by value by enclosing the argument name in parentheses (which changes the variable in to an expression; expressions are always passed by value). If the subroutine changes the value of an argument passed by value, the value of this argument in the calling program remains unchanged.

You can also use the **COMMON** statement to make specified variables available to all external subroutines. You should avoid calling **SUBROUTINEs** using a variable that is declared in **COMMON** as a subroutine argument as you will have two references to the same variable in the subroutine – the original **COMMON** reference, and the subroutine parameter.

Note: An array may be dimensioned differently in the subroutine than it is in the calling program, but that the number of dimensioned elements should remain the same. Hence a variable **A** declared as **DIM A(10)** may be declared as **A(5,2)** in the subroutine.

CALL, GOSUB, and SUBR

The **CALL** statement is used to call an external subroutine. The **GOSUB** statement is used to call an internal subroutine. The **SUBR** function is used to call an external subroutine that returns a value.

Examples

The following example uses **CALL** to pass an argument by reference:

```
Main
  x="Burma"
  PRINT x           ! Returns "Burma"
  CALL MapSub(x)
  PRINT x           ! Returns "Myanmar"

MapSub(country)
  PRINT country     ! Returns "Burma"
  country="Myanmar"
  PRINT country     ! Returns "Myanmar"
  RETURN
```

The following example uses **CALL** to pass an argument by value by using parentheses around the argument:

```
Main
  x="Burma"
  PRINT x           ! Returns "Burma"
  CALL MapSub((x))
  PRINT x           ! Returns "Burma"

MapSub(country)
  PRINT country     ! Returns "Burma"
  country="Myanmar"
  PRINT country     ! Returns "Myanmar"
  RETURN
```

See Also

- [COMMON](#) statement
- [RETURN](#) statement
- [SUBROUTINE](#) statement
- [END](#) statement
- [DIM](#) statement
- [GOSUB](#) statement
- [SUBR](#) function

CASE

Selects one of several statements based on the value of expressions.

```
BEGIN CASE      CASE expression1      statement      CASE expression2
      statement . . . END CASE
```

Arguments

<i>expression</i>	A value, variable, or expression that evaluates to a boolean value.
<i>statement</i>	One or more MVBasic statements to execute if the corresponding <i>expression</i> = true (any numeric value other than 0).

Description

The **CASE** statement tests each case in the order specified, and executes the *statement(s)* associated with the first *expression* that evaluates to true (a numeric value other than 0). An unlimited number of **CASE** statements can be specified within the **BEGIN CASE ... END CASE** clause. At most, only one **CASE** statement is taken — the first case that evaluates to a true value. Matching stops when the first *expression* that evaluates to true is encountered.

If no **CASE** *expression* evaluates to true, execution continues with the first statement after the **END CASE** statement.

You can specify a default case by specifying an *expression* that always evaluates to true (1). Typically, the literal integer value 1 is used as the *expression* in the last **CASE** clause. The statements associated with this clause will be executed if all the other **CASE** clauses evaluate to false (0).

Arguments

expression

CASE evaluates *expression* to a boolean value. If true, the case is taken and its statements executed. If false, the case is skipped over, and the next **CASE** *expression* is evaluated.

See Also

- [IF...THEN...ELSE](#) statement

CHAIN

Executes a MultiValue command from a program, exiting the program.

CHAIN command

Arguments

<i>command</i>	A MultiValue command specified as a quoted string.
----------------	--

Description

The **CHAIN** command executes the specified Caché MultiValue command, but does not return execution to the MVBasic program. Commonly, **CHAIN** is used with the MultiValue **RUN** command to “chain” execution from one program to another.

EXECUTE, PERFORM, and CHAIN

The **EXECUTE** command executes one or more MultiValue commands from within MVBasic, then returns execution to the next MVBasic statement in the invoking program. **EXECUTE** can pass values to the MultiValue command(s) and return values from the MultiValue command(s).

The **PERFORM** command executes one or more MultiValue commands from within MVBasic, then returns execution to the next MVBasic statement in the invoking program. **PERFORM** cannot pass or return values.

The **CHAIN** command executes a single MultiValue command from within MVBasic. It does not return execution to the invoking program. **CHAIN** cannot pass values.

Examples

The following example issues the MultiValue **RUN** command, to initiate execution of the `bignumprog` MVBasic program:

```
IF x>100
  THEN
    CHAIN "RUN bignumprog"
  END
ELSE
  PRINT "continuing execution"
END
```

See Also

- [EXECUTE](#) statement

- [PERFORM](#) statement
- Caché ObjectScript: XECUTE command

CLEARDATA

Clears all data stored by the **DATA** statement.

```
CLEARDATA
```

Arguments

None.

Description

The **CLEARDATA** statement flushes (clears) all remaining data stored in the input stack by the **DATA** statement. Following **CLEARDATA**, the **INPUT** statement issues a user prompt, rather than automatically receiving data stored by the **DATA** statement.

Examples

The following example illustrates the use of the **CLEARDATA** statement:

```
DATA "New York", "Chicago", "", "Annapolis"  
FOR x=1 TO 4  
  INPUT cityname  
  IF cityname=""  
    THEN CLEARDATA  
    PROMPT "Missing name: "  
    INPUT cityname  
  ELSE  
    PRINT cityname  
NEXT
```

See Also

- [DATA](#) statement
- [INPUT](#) statement

CLEARFILE

Deletes all records from a MultiValue file.

```
CLEARFILE filevar [ON ERROR statements] [LOCKED statements]
```

Arguments

<i>filevar</i>	A file variable name used to refer to a MultiValue file. This <i>filevar</i> value is supplied by the OPEN statement.
ON ERROR <i>statements</i>	<i>Optional</i> — One or more MVBasic statements executed if the file could not be accessed for record deletion.
LOCKED <i>statements</i>	<i>Optional</i> — One or more MVBasic statements executed if CLEARFILE could not delete all records due to lock contention.

Description

The **CLEARFILE** statement is used to delete all data from a MultiValue file. It does not delete the file itself. **CLEARFILE** takes the file identifier *filevar*, defined by the **OPEN** statement.

CAUTION: **CLEARFILE** can delete large quantities of data. This data may be accessed by multiple processes.

To delete individual data records, use the **DELETE** statement.

You can optionally specify an ON ERROR clause. If the data deletion fails, the ON ERROR clause is executed.

See Also

- [OPEN](#) statement
- [DELETE](#) statement
- [STATUS](#) statement
- [STATUS](#) function

CLOSE

Closes a MultiValue file.

```
CLOSE filevar [ON ERROR statements]
```

Arguments

<i>filevar</i>	A file variable name used to refer to a MultiValue file. This <i>filevar</i> is supplied by the OPEN statement.
ON ERROR <i>statements</i>	<i>Optional</i> — One or more MVBasic statements executed if the file to be closed could not be located.

Description

The **CLOSE** statement is used to close a MultiValue file. It takes the file identifier *filevar*, defined by the **OPEN** statement.

If multiple **OPEN** statements have been issued for the same MultiValue file:

- If the process has issued multiple **OPEN** statements specifying different *filevar* variables, you must issue a **CLOSE** for each *filevar*.
- If the process has issued multiple **OPEN** statements specifying the same *filevar*, a single **CLOSE** for this *filevar* closes the MultiValue file.
- If multiple processes have issued an **OPEN** statement for the same MultiValue file, you must issue a **CLOSE** for the *filevar* in each process, even if the processes specified the same *filevar* variable.

You can optionally specify an ON ERROR clause. If file close fails, the ON ERROR clause is executed. This may occur if *filevar* does not refer to an existing file, or if the *filevar* file has already been closed.

See Also

- [OPEN](#) statement
- [STATUS](#) statement
- [STATUS](#) function

CLOSESEQ

Closes a file opened for sequential access.

```
CLOSESEQ filevar [ON ERROR statements]
```

Arguments

<i>filevar</i>	A file variable name used to refer to the file in Caché MVBasic. This <i>filevar</i> is obtained from OPENSEQ .
----------------	--

Description

The **CLOSESEQ** statement is used to close a file that has been opened for sequential access using **OPENSEQ**. A file opened for sequential access is exclusively held by the process that opened it. Issuing a **CLOSESEQ** allows that file to be accessed by other processes.

You can use the **STATUS** function to determine the status of the close operation, as follows: 0=close successful; -1=close failed either because file variable not defined or file has already been closed.

You can optionally specify an **ON ERROR** clause. If file close fails, the **ON ERROR** clause is executed.

See Also

- [OPENSEQ](#) statement
- [STATUS](#) statement

COMMIT

Commits all changes made during the current transaction.

```
COMMIT [WORK] [THEN statements][ELSE statements]
```

Description

The **COMMIT** statement ends the current transaction initiated by a **BEGIN TRANSACTION** statement. All file changes issued during the transaction are committed, and cannot be subsequently reverted.

The **WORK** keyword is optional and provides no functionality. It is provided solely for compatibility with other MultiValue vendor products.

You can optionally specify a **THEN** clause, an **ELSE** clause, or both a **THEN** and an **ELSE** clause. If the transaction commit is successful, the **THEN** clause is executed. If the transaction commit fails, the **ELSE** clause is executed.

To revert the changes made during the current transaction, issue a **ROLLBACK** statement, rather than a **COMMIT** statement.

After the transaction is closed, program execution continues at the **END TRANSACTION** statement.

Note: Caché MVBasic supports two sets of transaction statements:

- UniVerse-style **BEGIN TRANSACTION**, **COMMIT**, **ROLLBACK**, and **END TRANSACTION**.
- UniData-style **TRANSACTION START**, **TRANSACTION COMMIT**, and **TRANSACTION ABORT**.

These two sets of transaction statements should not be combined.

Please refer to the documentation for [BEGIN TRANSACTION](#) for notes on important differences regarding the isolation level of transactions within Caché vs the that generally found in MV systems.

See Also

- [BEGIN TRANSACTION](#) statement
- [END TRANSACTION](#) statement
- [ROLLBACK](#) statement

COM (COMMON)

Lists variables available to external subroutines.

```
COM [/store/] var [,var2][. . .]
COMMON [/store/] var [,var2][. . .]
```

Arguments

<i>store</i>	<i>Optional</i> — A named storage area for the listed variables. If specified, this name is enclosed with slashes (/).
<i>var</i>	A variable or a comma-separated list of multiple variables.

Description

The **COMMON** statement allows you to specify list of common local variables that are available to external subroutines. You can specify one variable or a comma-separated list of variables. These variables do not have to be defined to be listed as common.

You specify a **COMMON** statement in both the calling program and each called subroutine that uses the variables. The corresponding variables in an external subroutine do not have to have the same names; they correspond by being in the same sequence. Thus the first variable in the main program's **COMMON** statement corresponds with the first variable in the external subroutine's **COMMON** statement, the second with the second, and so forth.

Note: Arrays dimensioned in COMMON areas in one program do not need to be dimensioned in the same way in the definition of the same COMMON area in another program. However, the number of elements defined should be the same in both cases. It is best practice to defined COMMON areas via a single INCLUDE file in order to avoid using different definitions in different programs.

See Also

- [CALL](#) statement
- [SUBROUTINE](#) statement

CONTINUE

Jumps to FOR or LOOP statements and re-executes test and loop.

CONTINUE

Arguments

The **CONTINUE** statement does not have any arguments.

Description

The **CONTINUE** statement is used within the code block of a **FOR...NEXT** or **LOOP...REPEAT** statement. **CONTINUE** causes execution to immediately jump back to the **FOR** or **LOOP** keyword, starting a new iteration of the loop. The **FOR** or **LOOP** statement evaluates its test condition, and, based on that evaluation, may re-execute the code block loop.

Examples

The following example illustrates the use of the **CONTINUE** statement:

```
FOR i=1 TO 10
  PRINT i
  IF i=5 THEN
    CONTINUE
  ELSE
    PRINT "not five"
  END IF
NEXT
```

See Also

- [FOR...NEXT](#) statement
- [LOOP...REPEAT](#) statement
- [EXIT](#) statement
- [GOTO](#) statement

CONVERT

Replaces single characters in a string.

```
CONVERT charsout TO charsin IN string
```

Arguments

<i>charsout</i>	One or more characters to be replaced. Any expression that resolves to a valid string or numeric.
<i>charsin</i>	The character or characters to be inserted in place of the corresponding characters in <i>charsout</i> . Any expression that resolves to a valid string or numeric.
<i>string</i>	The string in which character substitutions are made. Any expression that resolves to a valid string. <i>string</i> may be a dynamic array .

Description

The **CONVERT** statement edits the value of *string* by replacing all instances of single characters in *charsout* with single characters from *charsin*. **CONVERT** performs a character-for-character substitution. Matching of characters is case-sensitive.

CONVERT can be used as follows:

- To remove all instances of a character from a string, specify the character to be removed in *charsout* and a null string in *charsin*. For example, to remove the # character from *mystring*: `CONVERT "#" TO "" IN mystring`
- To replace all instances of a character in a string with another character, specify the character to be replaced in *charsout* and the replacement character in *charsin*. For example, to replace all instances of the # character with the * character in *mystring*: `CONVERT "#" TO "*" IN mystring`
- To replace all instances of a list of single characters with corresponding other single characters, specify those characters to be replaced in *charsout* and the corresponding replacement characters in *charsin*. For example, to replace all instances in *mystring* of the each lowercase letter a, b, c, and d with the corresponding uppercase letter: `CONVERT "abcd" TO "ABCD" IN mystring`
- To both replace some single characters and remove others, specify those characters to be replaced or removed in *charsout*. First specify those to be replaced, then those to be removed. Specify the corresponding replacement characters in *charsin*, and nothing for

the characters to be removed. For example, to replace all instances of + with &, and to remove all instances of # in *mystring*: `CONVERT "+#" TO "&" IN mystring`

The value of *charsout* and *charsin* can be a string or a numeric. If numeric, the value is converted to canonical form (plus sign, leading and trailing zeros removed) before performing the **CONVERT** operation.

If *charsout* contains more characters than *charsin*, the unpaired characters are deleted from *string*. If *charsin* contains more characters than *charsout*, the unpaired characters are ignored and have no effect.

Note: **CONVERT** performs single character one-for-one substitution for all instances in a string. The **CHANGE** function performs substring replacement, and can specify how many instances to replace and where to begin replacement.

The **CONVERT** statement and the **CONVERT** function perform the same operation, with the following difference: the **CONVERT** statement changes the supplied string; the **CONVERT** function returns a new string with the specified changes and leaves the supplied string unchanged.

Examples

The following example illustrates use of the **CONVERT** statement in converting a string to a dynamic array by replacing the # character with a Value Mark level delimiter character:

```
cities="New York#Chicago#Boston#Los Angeles"  
CONVERT "#" TO CHAR(253) IN cities  
PRINT cities
```

See Also

- [CONVERT](#) function
- [CHANGE](#) function
- [SWAP](#) statement
- [Strings](#)

\$COPYRIGHT

Inserts the specified text into generated object code.

```
$COPYRIGHT text
```

Arguments

<i>text</i>	A string enclosed in double or single quotes.
-------------	---

Description

The **\$COPYRIGHT** statement inserts the specified copyright text into the generated object code. This text is a non-executable comment. If there is more than one **\$COPYRIGHT** statement in a routine, only the text from the final one is inserted into the generated object code.

See Also

- [Comments](#)

CREATE

Creates a file for sequential access.

```
CREATE filevar [THEN statements][ELSE statements]
```

Arguments

<i>filevar</i>	A file variable name used to refer to the file in Caché MVBASIC. This <i>filevar</i> is obtained from OPENSEQ .
----------------	--

Description

The **CREATE** statement is used to create a file for sequential access. To create a file, you must first issue an **OPENSEQ** statement, giving the fully-qualified pathname for the file you wish to create. Because the file does not yet exist, the **OPENSEQ** appears to fail, taking its ELSE clause and setting the value returned by the **STATUS** function to -1. However, the **OPENSEQ** sets its *filevar* to an identifier for the specified file pathname. You then supply this *filevar* to **CREATE** to create a new file.

You can optionally specify a **THEN** clause, an **ELSE** clause, or both a **THEN** and an **ELSE** clause. If the file creation is successful, the **THEN** clause is executed. If file creation fails, the **ELSE** clause is executed.

The **CREATE** statement is:

- Optional if the first operation you perform on the new file is to issue a **WRITESEQ**. If you issue an **OPENSEQ** and then issue a **WRITESEQ**, this first write operation automatically creates the file.
- Mandatory if the first operation you perform on the new file is to issue a **WRITEBLK**. The **CREATE** creates the file, and then you may issue a **WRITEBLK** to write to the file.

You can use the **STATUS** function to determine the status of the file creation operation. A successful file creation returns a status of 0. A failed file creation returns a status of -1, for any of the following reasons:

- The directory specified in **OPENSEQ** does not exist. **CREATE** can create a file, but not the directory to contain the file. You can create the directory after issuing an **OPENSEQ** and then use the *filevar* returned by **OPENSEQ** to create the file.
- The file already exists.
- The specified *filevar* is invalid.

After creating a file, you can use the **STATUS** statement to obtain file status information. The file is open for read and write operations. You can use **CLOSESEQ** to release an open file, making it available to other processes.

Examples

The following example creates a new sequential file on a Windows system:

```
OPENSEQ "C:/myfiles/test1" TO mytest
  IF STATUS()=0
  THEN PRINT "File already exists"
  END
  ELSE PRINT STATUS();    ! returns -1
  CREATE mytest
  IF STATUS()=0
  THEN PRINT "File created"
  ELSE PRINT "File create failed"
  END
```

See Also

- [OPENSEQ](#) statement

- [CLOSESEQ](#) statement
- [WRITEBLK](#) statement
- [WRITESEQ](#) statement
- [STATUS](#) statement
- [STATUS](#) function

CRT

Displays on the screen.

```
CRT [printlist]
```

Arguments

<i>printlist</i>	<i>Optional</i> — Any MVBASIC expression that resolves to a quoted string. You can also specify a series of quoted strings, separated by either commas (,) or colons (:). A comma inserts a pre-defined tab spacing between the two strings. A colon concatenates the two strings. If <i>printlist</i> is omitted, a blank line is returned.
------------------	--

Description

CRT displays the items specified in *printlist* to the terminal. **CRT** will never send its output to an open **PRINTER** channel, which allows **CRT** to be executed without using **PRINTER OFF** and **PRINTER ON**. If no *printlist* is specified, **CRT** returns a blank line.

If you use a comma to separate strings in the *printlist*, a tab is inserted between the two items. By default, tabs are set at ten column intervals. To add spaces between items, use the **SPACE** function.

If you use a colon to separate strings in the *printlist*, the strings are concatenated. By default, a **CRT** statement ends by issuing a linefeed and carriage return. However, if you end the **CRT** argument with a colon, **CRT** does not issue the linefeed and carriage return. This enables you to concatenate the output of the next statement to the **CRT** output.

You can use an **@** function to specify the column position at which to print. For example, `CRT @(15):"Over here!"` prints the literal string starting at column 16.

The **PRINT** and **CRT** commands are identical.

Examples

The following examples illustrate the use of the CRT statement:

```
CRT "hello", "world": "!"
```

returns:

```
hello world!
```

See Also

- [PRINT](#) statement
- [LPTR](#) statement
- [SPACE](#) function

DATA

Provides user input data.

```
DATA exp [,exp2][. . .]
```

Arguments

<i>exp</i>	An expression to use as user input data. It can be a literal or a defined variable. You can specify a comma-separated list of multiple expressions.
------------	---

Description

The **DATA** statement defines one or more input values on an input stack for future use. A **DATA** value is taken from the input stack by the next **INPUT** statement, rather than pausing program execution for user input.

You can specify a comma-separated list of **DATA** values; these are used successively by multiple invocations of the **INPUT** statement.

A **DATA** value of the empty string (`DATA ""`) is treated as an actual data value: If the optional length parameter of a subsequent **INPUT** statement is set to -1, **INPUT** sets *variable* to 1 (indicating that there is input available). If the **INPUT** statement has a **THEN** clause, **INPUT** executes the statements associated with **THEN** clause as if the user had entered data from the keyboard.

You can use **CLEARDATA** to flush all remaining data stored by a **DATA** statement.

You cannot use **DATA** to supply a character to the **KEYIN** function.

See Also

- [INPUT](#) statement
- [CLEARDATA](#) statement

DEBUG

Interrupts program execution to enter debug mode.

```
DEBUG
```

Arguments

None.

Description

The **DEBUG** statement interrupts program execution by issuing a break to another stack level. From this point you can issue debug commands, including returning to the execution of the interrupted program.

DEL

Deletes an element from a dynamic array.

```
DEL dynarray <f[,v[,s]]>
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
<i>f</i>	An integer specifying the Field (attribute) level of the dynamic array on which to perform the deletion. Fields/Attributes are counted from 1.
<i>v</i>	<i>Optional</i> — An integer specifying the Value level of the dynamic array on which to perform the deletion. Values are counted from 1 within a Field.
<i>s</i>	<i>Optional</i> — An integer specifying the Subvalue level of the dynamic array on which to perform the deletion. Subvalues are counted from 1 within a Value.

Description

The **DEL** statement deletes one element from a dynamic array. It deletes both the data and the dynamic array delimiter. Which element to delete is specified by the *f*, *v*, and *s* integers. The enclosing angle brackets are mandatory. For example, if *f*=2 and *v*=3, this means delete the third value from the second field. If *f*=2 and *v* is not specified, this means to delete the entire second field.

The **DEL** statement and the **DELETE** function perform the same operation, with the following difference: **DEL** changes the supplied dynamic array; **DELETE** creates a new dynamic array with the specified change and leaves the supplied dynamic array unchanged.

Examples

The following example uses the **DEL** statement to delete the second value from the first field of a dynamic array:

```
cities="New York":@VM:"London":@VM:
"Chicago":@VM:"Boston":@VM:"Los Angeles"
PRINT cities
! Returns: "New YorkvLondonvChicagovBostonvLos Angeles"
DEL cities <1,2>
PRINT cities
! Returns: "New YorkvChicagovBostonvLos Angeles"
```

See Also

- [COUNTS](#) function
- [DELETE](#) function
- [EXTRACT](#) function
- [Dynamic Arrays](#)

DELETE, DELETEU

Deletes a record from a MultiValue file.

```
DELETE filevar,recID [LOCKED statements] [THEN statements] [ELSE
statements]
```

```
DELETEU filevar,recID [LOCKED statements] [THEN statements] [ELSE
statements]
```

Arguments

<i>filevar</i>	A local variable used as the file identifier of an open MultiValue file. This variable is set by the OPEN statement.
<i>recID</i>	The record ID of the record to be deleted.
<i>LOCKED statements</i>	<i>Optional</i> — One or more MVBasic statements executed if DELETE or DELETEU could not delete the record due to lock contention.
<i>THEN statements</i>	<i>Optional</i> — One or more MVBasic statements executed if DELETE has successfully deleted the record.
<i>ELSE statements</i>	<i>Optional</i> — One or more MVBasic statements executed if DELETE fails.

Description

The **DELETE** statement deletes a record from a MultiValue file. The **DELETEU** statement performs the same operation, but does not release an existing update lock if one was established.

You must use the **OPEN** statement to open a file before issuing either of these **DELETE** statements.

The optional **THEN** clause and **ELSE** clause specify one or more MVBasic statements to execute depending on the status of the delete operation. **DELETE** executes the **THEN** clause if the delete was successful. **DELETE** executes the **ELSE** clause if the delete was not successful.

DELETE completes successfully if the *recID* refers to a non-existent record.

Examples

The following example illustrates the use of the **DELETE** statement:

```
OPEN "Myfile.Test" TO myfile
DELETE myfile,myrec ON ERROR PRINT "no delete"
```

See Also

- [OPEN](#) statement
- [READ](#) statement
- [WRITE](#) statement
- [CLEARFILE](#) statement

DIM (DIMENSION)

Dimensions an array of variables.

```
DIM arrayname(rows[,columns])
DIMENSION arrayname(rows[,columns])
```

Arguments

The **DIM** statement syntax has these parts:

<i>arrayname</i>	Name of an array. Follows standard variable naming conventions.
<i>rows</i>	<i>Optional</i> — An integer specifying the number of array elements to dimension for a one-dimensional array, or the number of rows to dimension for a two-dimensional array.
<i>columns</i>	<i>Optional</i> — For two-dimensional (matrix) arrays, an integer specifying the number of columns per row. Can only be used in conjunction with the optional rows argument.

Description

The **DIM** statement dimension a one-dimensional or two dimensional array. This specifies the maximum number of elements that can be defined for that array.

All uninitialized variables are treated as zero-length strings ("").

Caché MVBasic allows arbitrary dimensioned arrays as well as arrays that can be subscripted using strings rather than numeric indices. Such variables must be declared with an argumentless DIM statement. This declares the variables as arrays, but the number of dimensions and number of elements in each dimension may be expanded at will, at runtime. Variables whose names start with a % are known as public arrays and their values are preserved across SUBROUTINE calls in a similar manner to COMMON arrays. Variables whose name begins with ^ are known as globals and their values are stored on disk automatically. Variables with normal naming conventions are known as local arrays and their value is lost when the program terminates as with any other variable.

Note: The **DIMENSION** and **DIM** keywords are synonyms.

Examples

The following examples illustrate the use of the **DIM** statement:

```
! Dimensions a one-dimensional array with 10 elements.
DIM MyVector(10)

! Dimensions a two-dimensional matrix array
! with 10 rows and 10 columns.
DIM MyMatrix(10,5)

! Dimension a local array of arbitrary size and subscript type.
DIM MyLocal()
  MyLocal(88) = "88"
  MyLocal(88,"The") = "The 88"
  MyLocal("Hello") = "World!"
```

Notes

Caché MVBasic does not require the dimension of arrays to be specified, and therefore does not implement the **ReDim** Statement.

See Also

- [Variables](#)

DISPLAY

Displays on the screen.

```
DISPLAY [printlist]
```

Arguments

<i>printlist</i>	<i>Optional</i> — Any MVBasic expression that resolves to a quoted string. You can also specify a series of quoted strings, separated by either commas (,) or colons (:). A comma inserts a pre-defined tab spacing between the two strings. A colon concatenates the two strings. If <i>printlist</i> is omitted, a blank line is returned.
------------------	--

Description

DISPLAY is identical in function to the **CRT** statement. Please refer to the documentation for **CRT** for further information.

See Also

- [CRT](#) statement
- [PRINT](#) statement
- [ECHO](#) statement
- [LPTR](#) statement
- [SPACE](#) function

ECHO

Suppresses input display on the screen.

```
ECHO {OFF | ON}
ECHO {expression}
```

Arguments

<i>expression</i>	A MVBASIC expression that resolves to a boolean value, either 0 (off) or 1 (on). You can also specify these values using the keywords OFF and ON. The default is 1.
-------------------	---

Description

The **ECHO** statement suppresses or allows the display of input characters on the terminal screen. If set to OFF, or 0, echoing of user input on the terminal screen is suppressed. If set to ON, or 1, user input is echoed on the terminal screen. One common use for **ECHO** is when entering a password, using the **INPUT** statement. **ECHO OFF** suppresses display of the input password; the password is written to the **INPUT** variable.

Examples

The following example illustrates the use of the **ECHO** statement:

```
PRINT "Type your user name"
INPUT uname
ECHO OFF
PRINT "Type your password"
INPUT pword
ECHO ON
```

See Also

- [CRT](#) statement
- [PRINT](#) statement
- [INPUT](#) statement

END

Terminates program execution.

```
END
```

Arguments

None.

Description

The **END** statement is used to terminate program execution.

The **END** keyword is also used as part of an **IF...THEN** statement, where it terminates execution of the block of code for the current clause of the **IF...THEN** statement.

See Also

- [GOTO](#) statement
- [IF...THEN](#) statement
- [RETURN](#) statement

END TRANSACTION

Specifies where to continue execution after a transaction.

```
END TRANSACTION
```

Description

The **END TRANSACTION** statement specifies where to continue program execution following the conclusion of a transaction. It should be placed in the code after the current transaction is closed, either by a **COMMIT** statement or a **ROLLBACK** statement.

If an **END TRANSACTION** is encountered before either a **COMMIT** or a **ROLLBACK**, the current transaction is rolled back.

Note: Caché MVBasic supports two sets of transaction statements:

- UniVerse-style **BEGIN TRANSACTION**, **COMMIT**, **ROLLBACK**, and **END TRANSACTION**.
- UniData-style **TRANSACTION START**, **TRANSACTION COMMIT**, and **TRANSACTION ABORT**.

These two sets of transaction statements should not be combined.

See Also

- [BEGIN TRANSACTION](#) statement
- [COMMIT](#) statement
- [ROLLBACK](#) statement

ERRMSG

Displays the specified error message.

```
ERRMSG num[ ,val ]
```

Arguments

<i>num</i>	An integer corresponding to a Caché MultiValue error message.
<i>val</i>	A string or numeric to be supplied as a value in the error message. You can specify a comma-separated list of <i>val</i> arguments.

Description

ERRMSG displays the error message text corresponding to the *num* error number. The error number (in square brackets) is also returned.

If you specify a *num* value that does not correspond to an error code, **ERRMSG** displays the string “Errmsg” with the error number in square brackets.

If you specify a *val* argument, **ERRMSG** displays the *num* error message text with *val* inserted in the message. If the *num* error message does not take an inserted value, the *val* argument is ignored. If the *num* value does not correspond to an error code, **ERRMSG** returns the “Errmsg” string with *val* appended and followed by a caret (^) separator character.

Examples

The following examples return an error message that does not take a supplied value:

```
ERRMSG 94
ERRMSG 94,24
ERRMSG 94,"test1","test2"
```

all of these return: [94] End of file.

The following examples return an error message that takes one supplied value:

```
ERRMSG 40
ERRMSG 40,24
ERRMSG 40,"test1","test2"
```

these return:

```
[40] Program " has not been compiled.
[40] Program '24' has not been compiled.
[40] Program 'test1' has not been compiled.
```

The following examples specify a *num* value that does not correspond to an error code:

```
ERRMSG 50
ERRMSG 50,24
ERRMSG 50,"test1","test2"
```

these return:

```
Errmsg[50]
Errmsg[50]24^
Errmsg[50]test1^test2^
```

EXECUTE

Executes a MultiValue command from within a program, passing and returning values.

Use any of the following three syntactical forms:

```
EXECUTE command [CAPTURING dynarray] [PASSLIST [dynarray]] [RTNLIST
var] [{SETTING | RETURNING} dynarray]

EXECUTE command
    [ , IN < expression]
    [ , OUT > var]
    [ , SELECT[ (list) ] < dynarray]
    [ , SELECT[ (list) ] > var]
    [ , PASSLIST[ (dynarray) ]]
    [ , STATUS > var]

EXECUTE command
    [ ,//IN. < expression]
    [ ,//OUT. > var]
    [ ,//SELECT.[ (list) ] < dynarray]
    [ ,//SELECT.[ (list) ] > var]
    [ ,//PASSLIST.[ (dynarray) ]]
    [ ,//STATUS. > var]
```

Arguments

<i>command</i>	One or more MultiValue commands, each command specified as a quoted string. To specify multiple commands, separate the commands with a Field Mark.
<i>var</i>	A variable used to hold a value.
<i>dynarray</i>	A dynamic array .

Description

The **EXECUTE** command executes the specified Caché MultiValue command(s), then returns execution to the next MVBasic statement following the **EXECUTE**.

The first syntactical form supports the following optional clauses:

- The **CAPTURING** clause diverts all terminal output from the MultiValue command to the supplied *var* variable. This output is stored as a dynamic array, with lines separated by Field Marks. **CAPTURING NULL** discards all terminal output. (Output from non-MultiValue commands or shell commands cannot be captured.)
- The **PASSLIST** clause supplies the specified *dynarray* to the executed command as the current default external select list.
- The **RTNLIST** clause receives the default select list (if any) produced by the executed command.

- The **RETURNING** clause receives the **ERRMSG** error message string with which the command terminated. The format is a dynamic array containing the **ERRMSG** number followed by the parameters.

The second and third syntactical forms support the following optional clauses:

- The **IN** clause specifies the input value for *command*.
- The **OUT** clause assigns the output from *command* to *var*.
- The **PASSLIST** clause supplies the specified *dynarray* to the executed command as the current default external select list.
- The **STATUS** clause *var* contains the return code of the last executed command.

EXECUTE, PERFORM, and CHAIN

The **EXECUTE** command executes one or more MultiValue commands from within MVBasic, then returns execution to the next MVBasic statement. **EXECUTE** can pass values to the MultiValue command(s) and return values from the MultiValue command(s).

The **PERFORM** command executes one or more MultiValue commands from within MVBasic, then returns execution to the next MVBasic statement. **PERFORM** cannot pass or return values.

The **CHAIN** command executes a single MultiValue command from within MVBasic. It does not return execution to MVBasic. **CHAIN** cannot pass values.

Examples

The following example issues the MultiValue **LISTME** command, captures its output in the dynamic array variable *currusers* and then returns execution to the MVBasic program:

```
PRINT TIME()  
EXECUTE "LISTME" CAPTURING currusers  
PRINT TIME()  
PRINT currusers
```

The following example shows how to use **EXECUTE** to execute multiple MultiValue commands:

```
PRINT TIME()  
EXECUTE "SLEEP 2":@FM:"SLEEP 3"  
PRINT TIME()
```

See Also

- **CHAIN** statement

- [PERFORM](#) statement
- Caché ObjectScript: XECUTE command

EXIT

Exits a LOOP...REPEAT or FOR...NEXT statement.

EXIT

Arguments

The **EXIT** statement takes no arguments.

Description

The **EXIT** statement can only be used within a **LOOP...REPEAT** or **FOR...NEXT** control structure to provide an alternate way to exit the loop. **EXIT** transfers control to the statement immediately following the end of the loop structure (the **NEXT** or **REPEAT** keyword).

Any number of **EXIT** statements may be placed anywhere in the block of code statements. **EXIT** is commonly used with the evaluation of some condition (such as an **IF...THEN** statement).

When used within nested loop statements, **EXIT** only exits the loop in which it occurs; **EXIT** transfers control to the loop that is nested one level above the exited loop.

The **GOTO** statement can also be used to exit from a loop control structure. The **CONTINUE** statement exits from the current iteration of a loop; the **EXIT** statement exits from the loop.

See Also

- [FOR...NEXT](#) statement
- [LOOP...REPEAT](#) statement
- [GOTO](#) statement
- [CONTINUE](#) statement

FIND

Finds an element of a dynamic array by exact value.

```
FIND data IN dynarray SETTING f[,v[,s]] [THEN statements][ELSE
statements]
```

Arguments

<i>data</i>	The data value of an element. This value must be the complete value of the element.
<i>dynarray</i>	Any valid dynamic array .
<i>f</i>	A variable that receives an integer denoting the Field level of the dynamic array where the element <i>data</i> was found. Fields are counted from 1.
<i>v</i>	<i>Optional</i> — A variable that receives an integer denoting the Value level of the dynamic array where the element <i>data</i> was found. Values are counted from 1 within a Field.
<i>s</i>	<i>Optional</i> — A variable that receives an integer denoting the Subvalue level of the dynamic array where the element <i>data</i> was found. Subvalues are counted from 1 within a Value.

Description

The **FIND** statement locates the *data* value in a dynamic array and returns its location by setting the *f*, *v*, and *s* variables to integers. For example, if *data* is located in the third Value of the second Field, **FIND** sets *f*=2 and *v*=3.

The *data* value must be an exact match with the full value of an element in *dynarray*. It cannot be a substring of an element value. Matching is case-sensitive. If *data* does not match an element value, *f*, *v*, and *s* are unchanged and retain their previous values.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If *data* is located in *dynarray*, the THEN clause is executed. If *data* is not located in *dynarray*, the ELSE clause is executed.

The **FIND** statement returns the *f*, *v*, and *s* position of a dynamic array element by specifying the element's exact value. The **FINDSTR** statement returns the *f*, *v*, and *s* position of a dynamic array element by specifying a substring found in that element. The **EXTRACT** function returns the value of a dynamic array element by specifying its *f*, *v*, and *s* position.

You can use the `<>` operator or the **REPLACE** function to replace an element value in a dynamic array based on position. For further details, see the [Dynamic Arrays](#) page of this manual.

Examples

The following example uses the **FIND** statement to find the second value from the first field of a dynamic array:

```
cities="New York":@VM:"London":@VM:
"Chicago":@VM:"Boston":@VM:"Los Angeles"
FIND "London" IN cities SETTING v,f,s
PRINT v,f,s
```

See Also

- [FINDSTR](#) statement
- [LOCATE](#) statement
- [EXTRACT](#) function
- [REPLACE](#) function
- [Dynamic Arrays](#)
- [Strings](#)
- [Variables](#)

FINDSTR

Finds an element of a dynamic array by substring value.

```
FINDSTR substring IN dynarray[,occurrence] SETTING fm[,vm[,sm]] [THEN
statements][ELSE statements]
```

Arguments

<i>substring</i>	A string to match against each element in <i>dynarray</i> .
<i>dynarray</i>	The target dynamic array in which <i>substring</i> is located.
<i>occurrence</i>	<i>Optional</i> — An integer that specifies which occurrence of substring to return <i>dynarray</i> . The default is 1.
<i>fm</i> <i>vm</i> <i>sm</i>	Variables that receive an integer specifying the Field Mark (<i>fm</i>) Value Mark (<i>vm</i>) and Subvalue Mark (<i>sm</i>) where <i>substring</i> is located in <i>dynarray</i> . For further information on these level delimiters, see the Dynamic Arrays page of this manual.
<i>statements</i>	<i>Optional</i> — One or more Caché MVBasic commands following the THEN or ELSE keyword.

Description

The **FINDSTR** statement searches a dynamic array for the specified substring. If it locates the substring, it sets integer count variables specifying which element of the dynamic array contains the substring. By default, it locates the first occurrence of *substring* in the dynamic array, reading left to right. You can set the optional *occurrence* argument for subsequent occurrences of *substring* in the dynamic array.

If **FINDSTR** finds *substring*, it sets *fm*, *vm*, and *sm* to an integer count. If dynamic array delimiters for a lower level do not exist, **FINDSTR** sets this level's variable (*vm* and/or *sm*) to 1. If *substring* is not located, *fm*, *vm*, and *sm* are not modified, and continue to hold their previous values.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If *substring* is located in *dynarray*, the THEN clause is executed. If *substring* is not located in *dynarray*, the ELSE clause is executed.

The **FINDSTR** statement returns the *f*, *v*, and *s* position of a dynamic array element by specifying a substring found in that element. The **FIND** statement returns the *f*, *v*, and *s*

position of a dynamic array element by specifying the element's exact value. The **EXTRACT** function returns the value of a dynamic array element by specifying its *f*, *v*, and *s* position.

Examples

The following example shows how to use the **FINDSTR** statement:

```
statecity="Kansas":@VM:"Kansas City":@VM:"Topeka"
:@FM:"Missouri":@VM:"St Louis":@VM:"Kansas City"
FOR x=1 TO 5
FINDSTR "Kansas" IN statecity,x SETTING f,v,s
PRINT f,v,s
NEXT
```

This example returns the following values for *f*, *v*, and *s*:

```
1 1 1 ! 1st occurrence of substring "Kansas"
1 2 1 ! 2nd occurrence of substring "Kansas"
2 3 1 ! 3rd occurrence of substring "Kansas"
2 3 1 ! no further occurrences, variables unchanged
2 3 1
```

See Also

- [FIND](#) statement
- [EXTRACT](#) function
- [REPLACE](#) function
- [Dynamic Arrays](#)
- [Strings](#)
- [Variables](#)

FOOTING

Prints a footer at the bottom of each output page.

```
FOOTING [ON channel] footer
```

Arguments

<i>channel</i>	<i>Optional</i> — An integer that specifies a logical print channel. The default is 0.
<i>footer</i>	The footer to print on output pages, specified as a quoted string. This footer can consist of any combination of literal text and code characters. Code character letters are enclosed in single quote characters, and are case-insensitive.

Description

The **FOOTING** statement prints a footer at the bottom of each page of printed output text. The *footer* can consist of a literal text and code characters that either specify text (for example, include the current date), or control the printing of footer text (for example, center the footer). A *footer* is always enclosed in double quotation marks. To include letter code characters, enclose them in single quotation marks. To include a literal single quotation mark, double it. For example: "Mary' 's Report".

The optional *channel* specifies the logical print channel for this output. The range of available values is -1 through 255 (inclusive). If *channel*=-1, output is displayed on the terminal screen. If *channel* is not specified, the default logical print channel is 0.

The following are the available code characters that supply footer text:

'D'	Include current date formatted as dd mmm yyyy. For example, 11 Sep 2006.
'T' \ 'P'	Include current time and date formatted as hh:mm:ss dd mmm yyyy. Time is in 12-hour format with "am" or "pm" appended. For example, 7:45:22pm 11 Sep 2006 .
^	Include current page number, right-aligned. The default alignment is 4 digits. You can specify a larger or smaller alignment by appending an integer to 'P'. For example, 'P2'.
'S'	Include current page number, left-aligned.
'R'	Include record ID, left-justified.

The 'S' and 'P' code characters specify whether an increasing number of digits (1, 10, 100, etc.) should expand the page number to the left or to the right. These code characters can be included at any point within the text of a footer. The page number appears at that point, either left-aligned ('S') or right-aligned ('P'). By default, both 'S' and 'P' are left-justified. To right-justify a page number, use the 'G' code, as follows: 'GS' or 'GP'.

The following are the available code characters that format footer text:

'C'	Center the footer. You can adjust centering alignment by appending an integer to 'C'. For example, 'C15'. You can also center a footer using the 'G' code character.
'G'	Insert spaces to evenly distribute the footer across the full available width. You can specify multiple 'G' codes within a footer.
'L']	Line break. Text after line break defaults to left-justified.
'N'	Suppress automatic paging.
'Q'	Treat \,], and ^ as literals, not code characters for rest of footer.

By default, a footer is left-justified. To right-justify a footer, specify a 'G' before the footer text: "'G'Annual Report". To center a footer, specify a 'G' before and after the text: "'G'Annual Report'G' ". To spread out the parts of a footer, specify a 'G' between literals in the footer: "'G'Annual'G'Report'G' ".

By default, the backslash (\), right square bracket (]), and caret (^) are code characters. To include these characters as literals in a footer, use the 'Q' code character. Any instances of these three characters following the 'Q' code in the footer are treated as literals, not code characters.

The **FOOTING** statement places text at the bottom of each page. The **HEADING** statement places text at the top of each page.

Examples

The following example centers the current date at the bottom of each page. Note that the footer must be enclosed in double quotation marks, even when there is no literal footer text:

```
FOOTING "'CD' "
```

The following example centers two lines of footer, with the page number right-justified on the first footer line:

```
FOOTING "'G'Big Widgets Corporation'GS''LC'First Quarter Report"
```

The following example left-justifies two lines of footer, with the page number at the end of the first footer line and the time and date at the end of the second footer line. Note that the punctuation code characters are not enclosed in single quotes:

```
FOOTING "Big Widgets Corporation^]First Quarter Report \"
```

See Also

- [HEADING](#) statement
- [PRINT](#) statement

FOR...NEXT

Repeats a group of statements a specified number of times.

```
FOR var = start TO end
  [STEP increment]
  [WHILE expression]
  [UNTIL expression]
  statements
NEXT
```

Arguments

The **FOR...NEXT** statement syntax has these parts:

<i>var</i>	A numeric variable used as a loop counter. The variable can't be an array element or an element of a user-defined type.
<i>start</i>	Initial value of counter.
<i>end</i>	Final value of counter.
STEP <i>increment</i>	<i>Optional</i> — The STEP clause sets the amount the counter is changed each time through the loop. A positive or negative number. If a STEP clause is not specified, <i>increment</i> defaults to 1. If <i>increment</i> is 0, FOR...NEXT loops infinitely.
WHILE <i>expression</i> UNTIL <i>expression</i>	<i>Optional</i> — The WHILE and UNTIL clauses specify a test condition for exiting the FOR loop. You can omit or specify either clause, or specify both clauses in any order.
<i>statements</i>	One or more statements between FOR and NEXT that are executed the specified number of times.

Description

The **FOR...NEXT** statement begins with a FOR keyword with *var=start TO end* to establish a loop counter. This is followed by one or more optional clauses: STEP, WHILE, and UNTIL. The loop itself consists on one or more executable *statements*. The FOR loop is ended by the mandatory NEXT keyword.

The counter functions as follows:

- If $start < end$, the loop executes the specified number of times.
- If $start = end$, the loop executes once.
- If $start > end$, the loop does not execute.

Most commonly, *start* and *end* are positive integers. They can, however, be positive or negative integers or decimal numbers.

The optional STEP clause sets an increment (or decrement) for the counter. By default, the counter increments by 1. The *increment* argument can be either positive (increment) or negative (decrement). Most commonly *increment* is an integer, but it can be a decimal number. An *increment* of 0 causes an infinite loop.

Once the loop starts and all statements in the loop have executed, *increment* is added to the counter. At this point, either the statements in the loop execute again (based on the same test that caused the loop to execute initially), or the loop is exited and execution continues with the statement following the NEXT keyword.

You can nest **FOR...NEXT** loops by placing one **FOR...NEXT** loop within another. Give each loop a unique variable name as its counter. The following construction is correct:

```
FOR i = 1 TO 10
  FOR j = 1 TO 10
    FOR k = 1 TO 10
      ! Some statements
    NEXT
  NEXT
NEXT
```

You can use a **CONTINUE** statement to interrupt a loop and return to the counter.

Notes

Changing the value of counter while inside a loop can make it more difficult to read and debug your code.

See Also

- [CONTINUE](#) statement
- [EXIT](#) statement
- [LOOP...REPEAT](#) statement
- [IF...THEN](#) statement

FUNCTION

Declares the name, arguments, and code that form the body of a Function procedure.

```
[Public | Private] Function name [(arglist)] [ As classname ]
    [statements]
    [name = expression]
    [Exit Function]
    [statements]
    [name = expression]
End Function
```

Arguments

The **FUNCTION** statement syntax has these parts:

Public	Indicates that the Function procedure is accessible to all other procedures in all scripts.
Private	Indicates that the Function procedure is accessible only to other procedures in the script where it is declared.
<i>name</i>	Name of the Function ; follows standard variable naming conventions.
<i>arglist</i>	List of variables representing arguments that are passed to the Function procedure when it is called, separated by commas.
<i>classname</i>	Name of the class of the return value.
<i>statements</i>	Any group of statements to be executed within the body of the Function procedure.
<i>expression</i>	Return value of the Function .

The arglist argument has the following syntax and parts:

```
[ByVal | ByRef] varname[( )]
```

ByVal	Indicates that the argument is passed by value.
ByRef	Indicates that the argument is passed by reference.
<i>varname</i>	Name of the variable representing the argument; follows standard variable naming conventions.

Description

Function procedures are visible to all other procedures in your script. The value of local variables in a **Function** is not preserved between calls to the procedure.

All executable code must be contained in procedures. You can't define a **Function** procedure inside another **Function** or **Sub** procedure.

The **Exit Function** statement causes an immediate exit from a **Function** procedure. Program execution continues with the statement following the statement that called the **Function** procedure. Any number of **Exit Function** statements can appear anywhere in a **Function** procedure.

Like a **Sub** procedure, a **Function** procedure is a separate procedure that can take arguments, perform a series of statements, and change the values of its arguments. However, unlike a **Sub** procedure, you can use a **Function** procedure on the right side of an expression in the same way you use any intrinsic function, such as **Sqr**, **Cos**, or **Chr**, when you want to use the value returned by the function.

You call a **Function** procedure using the function name, followed by the argument list in parentheses, in an expression. See the **Call** statement for specific information on how to call **Function** procedures.

There are two ways to return a value from a **function**: you can specify the value on a **Return** statement, or you can assign the value to the function name. Any number of such assignments can appear anywhere within the procedure. If no value is assigned to name, the procedure returns a default value: a zero-length string (""). A function that returns an object reference returns a zero-length string ("") if no object reference is assigned to name within the **Function**.

Variables used in **Function** procedures fall into two categories: those that are explicitly declared within the procedure and those that are not. Variables that are explicitly declared in a procedure (using **Dim** or the equivalent) are always local to the procedure. Variables that are used but not explicitly declared in a procedure are also local unless they are explicitly declared at some higher level outside the procedure.

Examples

The following example shows both ways to assign a return value. First by specifying “True” to the **Return** statement, then by assigning “False” to the function named `IsGreaterThan`. False is assigned to the function name to indicate that an invalid value was found.

```
Function IsGreaterThan(lower, upper)
IF lower < upper THEN Return True
IsGreaterThan = False
End Function
```

Notes

Function procedures can be recursive; that is, they can call themselves to perform a given task. However, recursion can lead to stack overflow.

See Also

- [CALL](#) statement
- [DIM](#) statement
- [RETURN](#) statement
- [SUBROUTINE](#) statement

GOSUB

Transfers program execution to a label, with return option.

```
GOSUB label
```

Arguments

<i>label</i>	Any valid label . The <i>label</i> name can be optionally followed by a colon (:)
--------------	---

Description

The **GOSUB** statement is used to transfer execution to the line of code identified by *label*. This label identifies an internal subroutine that is executed until a **RETURN** statement is encountered. Execution then reverts to the line immediately following the **GOSUB** statement. (Execution of an internal subroutine can also terminate with an **END** statement, which does not return control to **GOSUB**.)

The *label* argument value corresponds to line of code identified by a [label](#) identifier. Non-numeric labels end with a colon character; this colon is option when specifying the *label* argument.

The **GOSUB** statement is similar to **GOTO**, except that **GOSUB** permits a **RETURN**. The **ON** statement provides a way to select one of several **GOSUB** labels, based on an integer value.

Examples

The following example illustrates the use of the **GOSUB** statement:

```
IF TIME()=0 THEN
  GOSUB Midnight:
  PRINT "Delayed",TIME()
ELSE
  PRINT TIME()
END IF
Midnight:
PRINT "It's midnight, time is reset to 0"
SLEEP 1
RETURN
```

See Also

- [GOTO](#) statement
- [RETURN](#) statement
- [END](#) statement
- [ON](#) statement
- [Labels](#)

GOTO

Transfers program execution to a label.

```
GOTO label
GO label
GO TO label
```

Arguments

<i>label</i>	Any valid label . The <i>label</i> name can be optionally followed by a colon (:)
--------------	---

Description

The **GOTO** statement is used to transfer execution to the line of code identified by *label*. The *label* argument value corresponds to line of code identified by a [label](#) identifier. Non-numeric labels end with a colon character; this colon is option when specifying the *label* argument.

GO and **GO TO** are alternate forms of the **GOTO** statement. The **GOSUB** statement is similar to **GOTO**, except that it permits a **RETURN**. The **ON** statement provides a way to select one of several **GOTO** labels, based on an integer value.

Commonly, **GOTO** is used within a code block of an **IF...THEN** statement.

You can use the **EXIT** statement to cause execution to jump out of a **FOR...NEXT** or **LOOP...REPEAT** loop. You can use the **CONTINUE** statement to cause execution to jump back to the **FOR** or **LOOP** statement to perform the next loop iteration.

Examples

The following example illustrates the use of the **GOTO** statement:

```
IF TIME()=0 THEN
  GOTO Midnight:
ELSE
  PRINT Time()
END IF
Midnight:
  PRINT "It's midnight, time is reset to 0"
```

See Also

- [GOSUB](#) statement

- [ON](#) statement
- [IF...THEN](#) statement
- [EXIT](#) statement
- [FOR...NEXT](#) statement
- [LOOP...REPEAT](#) statement
- [CONTINUE](#) statement
- [Labels](#)

HEADING

Prints a header at the top of each output page.

```
HEADING [ON channel] header
```

Arguments

<i>channel</i>	<i>Optional</i> — An integer that specifies a logical print channel. The default is 0.
<i>header</i>	The header to print on output pages, specified as a quoted string. This header can consist of any combination of literal text and code characters. Code character letters are enclosed in single quote characters, and are case-insensitive.

Description

The **HEADING** statement prints a header at the top of each page of printed output text. The *header* can consist of a literal text and code characters that either specify text (for example, include the current date), or control the printing of header text (for example, center the header). A *header* is always enclosed in double quotation marks. To include letter code characters, enclose them in single quotation marks. To include a literal single quotation mark, double it. For example: "Mary' 's Report".

The optional *channel* specifies the logical print channel for this output. The range of available values is -1 through 255 (inclusive). If *channel*=-1, output is displayed on the terminal screen. If *channel* is not specified, the default logical print channel is 0.

The following are the available code characters that supply header text:

'D'	Include current date formatted as dd mmm yyyy. For example, 11 Sep 2006.
'T' \ \	Include current time and date formatted as hh:mm:ss dd mmm yyyy. Time is in 12-hour format with "am" or "pm" appended. For example, 7:45:22pm 11 Sep 2006 .
'P' ^	Include current page number, right-aligned. The default alignment is 4 digits. You can specify a larger or smaller alignment by appending an integer to 'P'. For example, 'P2'.
'S'	Include current page number, left-aligned.
'R'	Include record ID, left-justified.

The 'S' and 'P' code characters specify whether an increasing number of digits (1, 10, 100, etc.) should expand the page number to the left or to the right. These code characters can be included at any point within the text of a header. The page number appears at that point, either left-aligned ('S') or right-aligned ('P'). By default, both 'S' and 'P' are left-justified. To right-justify a page number, use the 'G' code, as follows: 'GS' or 'GP'.

The following are the available code characters that format header text:

'C'	Center the header. You can adjust centering alignment by appending an integer to 'C'. For example, 'C15'. You can also center a header using the 'G' code character.
'G'	Insert spaces to evenly distribute the header across the full available width. You can specify multiple 'G' codes within a header.
'L']	Line break. Text after line break defaults to left-justified.
'N'	Suppress automatic paging.
'Q'	Treat \,], and ^ as literals, not code characters for rest of header.

By default, a header is left-justified. To right-justify a header, specify a 'G' before the header text: "'G'Annual Report". To center a header, specify a 'G' before and after the text: "'G'Annual Report'G' ". To spread out the parts of a header, specify a 'G' between literals in the header: "'G'Annual'G'Report'G' ".

By default, the backslash (\), right square bracket (]), and caret (^) are code characters. To include these characters as literals in a header, use the 'Q' code character. Any instances of

these three characters following the 'Q' code in the header are treated as literals, not code characters.

The **HEADING** statement places text at the top of each page. The **FOOTING** statement places text at the bottom of each page.

Examples

The following example centers the current date at the top of each page. Note that the header must be enclosed in double quotation marks, even when there is no literal header text:

```
HEADING "'CD' "
```

The following example centers two lines of header, with the page number right-justified on the first header line:

```
HEADING "'G'Big Widgets Corporation'GS''LC'First Quarter Report"
```

The following example left-justifies two lines of header, with the page number at the end of the first header line and the time and date at the end of the second header line. Note that the punctuation code characters are not enclosed in single quotes:

```
HEADING "Big Widgets Corporation^]First Quarter Report \"
```

See Also

- [FOOTING](#) statement
- [PRINT](#) statement

IF...THEN...ELSE

Conditionally executes a group of statements, depending on the value of an expression.

```
IF condition THEN statements
IF condition ELSE elsestatements
IF condition THEN statements ELSE elsestatements

IF condition
[ THEN
  statements
END]
[ ELSE
  elsestatements
END]
```

Arguments

<i>condition</i>	An expression that evaluates to True or False . For further details on boolean logical operators, refer to the Operators page of this manual.
<i>statements</i>	One or more statements executed if <i>condition</i> is True .
<i>elsestatements</i>	One or more statements executed if no previous <i>condition</i> expression is True.

Description

The **IF** statement performs a boolean test on *condition*, and then executes either the **THEN** clause (*condition*=1 (true)) or the **ELSE** clause (*condition*=0 (false)). You can omit or include either the **THEN** clause or the **ELSE** clause. Further **IF** statements can be nested within **THEN** or **ELSE** clauses.

IF can be coded as a single-line statement, or as a code block statement using the **END** keyword. You can use any of the single-line forms for short, simple tests. However, the block form provides more structure and flexibility than the single-line form and is usually easier to read, maintain, and debug.

When executing a block **IF**, *condition* is tested. If *condition* is **True**, the statements following **THEN** are executed. If *condition* is **False**, the statements following **ELSE** are executed. After executing the statements following **THEN** or **ELSE**, execution continues with the statement following **END**.

What follows the **THEN** keyword is examined to determine whether or not a statement is a block **IF**. If anything other than a comment appears after **THEN** on the same line, the statement is treated as a single-line **IF** statement.

For a block **IF** statement, the **IF** keyword must be the first statement on a line. The block **IF** must end with an **END** statement.

See Also

- [CASE](#) statement
- [Operators](#)

INPUT

Receives user input.

```
INPUT [@(col[,row])] variable [,length] [THEN statements ELSE
statements]
```

Arguments

<i>@(col,row)</i>	<i>Optional</i> — A clause that specifies the location (column and row) to put the input prompt on the screen. If you specify this clause, INPUT displays the previous value of <i>variable</i> at the prompt. A <i>col</i> value of 0 or 1 displays the prompt at column 1. If <i>row</i> is omitted, it defaults to <i>row=1</i> , the top of the Caché terminal window; <i>row=23</i> is the bottom of the Caché terminal window.
<i>variable</i>	A variable used to receive the user input. This variable does not need to be previously defined.
<i>length</i>	<i>Optional</i> — An integer specifying the length of the input data. If omitted, or <i>length=0</i> , the input data length is determined by pressing the return key. If <i>length</i> is -1, <i>variable</i> is assigned a boolean value indicating whether or not data was input. The <i>length</i> integer can be followed by the underscore (<code>_</code>) character, and/or the colon (<code>:</code>) character. These suffix characters are described below.
<i>statements</i>	<i>Optional</i> — One or more executable MVBASIC statements. The execution of these statements depends upon the presence or absence of user input data: If any data is input, INPUT branches to the THEN clause. If no data is input (the Enter key is pressed), INPUT branches to the ELSE clause.

Description

The **INPUT** statement is used in interactive programs to receive input from the user. **INPUT** pauses program execution while awaiting user input. By default, it displays a question mark (?) prompt to receive user input. (This prompt is modifiable using the **PROMPT** statement.) The user types this input which appears character-by-character at the input prompt. Program execution continues when:

- The user presses the return key, if *length* is not specified.
- The user presses the return key, if *length=0*, or the input data is less than the number of characters specified in *length*.

- The input data is equal to the number of characters specified in *length*.
- The user presses the return key, if *length* has the underscore (_) character suffix. In this case, at most *length* number of characters are input, regardless of the number of characters typed.

You can also use the **KEYIN** function to receive a single character of user input.

By default, **INPUT** concludes data input with a line return. You can suppress this line return by following the *length* argument with a colon character (:).

If *length*=0, user input continues until the return key is pressed. If *length*=-1, user input is not stored; **INPUT** checks the input buffer for data, and places a boolean value in *variable*: 1 if data was input, 0 if no data was input.

If you specify the optional *@(col,row)* clause, the question mark (?) prompt appears at the specified column and row location. This prompt displays the previous value of *variable*. To accept this previous value, press the return key. To delete and replace this value, type the new value. To replace this value with a null value, press the space bar or tab key, then press the return key. This *@(col,row)* clause suppresses the line return following data input.

INPUT and DATA

If you use the **DATA** statement to pre-define a user input value, the **INPUT** statement takes its value from the **DATA** statement rather than from user input. The **INPUT** statement does not pause program execution or require user interaction. The **DATA** statement value does not conclude with a return character, and the **INPUT** statement does not issue a line return. If the *length* argument is specified, only that number of characters is input from the **DATA** statement value. The *length* argument suffix characters have no effect on **DATA** statement input.

INPUT treats a **DATA** value of the empty string (**DATA** " ") as an actual data value: If *length*=-1, **INPUT** sets *variable*=1. If the **INPUT** has a **THEN** clause, **INPUT** branches execution to the **THEN** clause statements.

If a **DATA** statement contains a comma-separated list of arguments, these arguments are supplied in order to multiple invocations of the **INPUT** statement.

Values supplied by a **DATA** can be flushed using the **CLEARDATA** statement. Following a **CLEARDATA**, the next **INPUT** prompts the user for input data.

Examples

The following example shows how to use the **INPUT** statement, This program pauses for user input:

```
PRINT "Input the person's last name"
INPUT x,16_
IF x=""
    PRINT "No name input"
ELSE
    PRINT "Last name (16 chars) ":x
```

The following example shows how to use **INPUT** with **DATA**. This program does not pause for user input:

```
DATA "Smith" INPUT x
IF x=""
    PRINT "DATA input failed"
ELSE
    PRINT "Last name (16 chars) ":x
```

See Also

- [DATA](#) statement
- [PROMPT](#) statement
- [CLEARDATA](#) statement
- [KEYIN](#) function

INS

Inserts data in a dynamic array.

```
INS expression BEFORE dynarray <f[,v[,s]]>
```

Arguments

<i>expression</i>	The data to be inserted.
<i>dynarray</i>	The name of a valid dynamic array. If the dynamic array does not exist, INS creates it.
<i>f</i>	An integer specifying the Field level of the dynamic array in which to insert the data. Fields are counted from 1.
<i>v</i>	<i>Optional</i> — An integer specifying the Value level of the dynamic array in which to insert the data. Values are counted from 1 within a Field.
<i>s</i>	<i>Optional</i> — An integer specifying the Subvalue level of the dynamic array in which to insert the data. Subvalues are counted from 1 within a Value.

Description

The **INS** statement inserts a data value at the specified dynamic array location. Which element to insert is specified by the *f*, *v*, and *s* integers. For example, if *f*=2 and *v*=3, this means insert the new data value as the third value in the second field. The **INS** statement does not overwrite; if there already was a third value, the insert increments its location to the fourth value. **INS** adds multiple delimiter characters, when needed, to place the data value at the specified location.

Examples

The following example uses the **INS** statement to insert the second value in the first field of a dynamic array:

```
cities="New York":@VM:"London":@VM:
"Chicago":@VM:"Boston":@VM:"Los Angeles"
INS "Providence" BEFORE cities <1,2>
PRINT cities
! Returns: "New YorkvProvidencevLondonvChicagovBostonvLos Angeles"
```

See Also

- [INSERT](#) function
- [COUNTS](#) function
- [DELETE](#) function
- [EXTRACT](#) function
- [Dynamic Arrays](#)

LET

Assigns a value to a variable.

```
LET var=expression
```

Arguments

<i>var</i>	Any valid variable name.
<i>expression</i>	Any MVBASIC expression that resolves to a value.

Description

The **LET** statement assigns the value of *expression* to the variable *var*. You can perform the same assignment operation by just specifying *var=expression* without the LET keyword. For further details on assignment operations, refer to the [Variables](#) page of this manual.

Examples

The following examples use **LET** to assign values to the variable x:

```
LET x=12
LET x="Fred"
LET x="Con": "catenate"
LET x=" "
LET x=4+4*3;      ! Returns 16
LET x=(4+4)*3;   ! Returns 24
```

See Also

- [Variables](#)

LOCATE

Finds an element in a specified part of a dynamic array by exact value.

```
LOCATE data IN dynarray <f[,v[,s]]> SETTING variable [THEN
statements][ELSE statements]
```

Arguments

<i>data</i>	The element value to search for in <i>dynarray</i> . This value must be the complete value of the element. An expression that evaluates to a string or a numeric value. Values are case-sensitive.
<i>dynarray</i>	A valid dynamic array .
<i>f</i>	An integer that denotes the Field level of the dynamic array to search for the element <i>data</i> . Fields are counted from 1. The surrounding angle brackets are required.
<i>v</i>	<i>Optional</i> — An integer that denotes the Value level of the dynamic array to search for the element <i>data</i> . Values are counted from 1.
<i>s</i>	<i>Optional</i> — An integer that denotes the Subvalue level of the dynamic array to search for the element <i>data</i> . Subvalues are counted from 1.
<i>variable</i>	A local variable that LOCATE sets to an integer specifying either where <i>data</i> is located or where <i>data</i> can be added.

Description

The **LOCATE** statement is used to search for an element value in a dynamic array and return the search results by setting *variable*. You can set the *f*, *v*, and *s* variables to integers to specify which data item(s) of the dynamic array to search. For example, setting *f*=2 searches the second dynamic array Field for the *data* value.

The *data* value must be an exact match with the full value of an element in *dynarray*. It cannot be a substring of an element value. Matching is case-sensitive. If *data* does not match an element value, *variable* is set to an integer 1 larger than the current last element. This specifies how many elements were searched and where the missing value can be appended to the existing values.

You can optionally specify a **THEN** clause, an **ELSE** clause, or both a **THEN** and an **ELSE** clause. If *data* is located in *dynarray*, the **THEN** clause is executed. If *data* is not located in *dynarray*, the **ELSE** clause is executed.

LOCATE and FIND

The **LOCATE** statement and the **FIND** statement both search for an exact element value in a dynamic array and return its location. Both support optional syntax **THEN** for successful search and **ELSE** for unsuccessful search. They differ in the following ways:

- **FIND** is used to search an entire dynamic array; there is no way to limit its scope to a portion of the dynamic array. **LOCATE** can use the *f*, *v*, and *s* variables to limit the scope of the search.
- When a search is successful, **FIND** returns an absolute location within the dynamic array; **LOCATE** returns a count relative to the specified starting location.
- When a search is unsuccessful, **FIND** provides no location information; **LOCATE** provides information on where the missing value could be appended to the existing values.

To locate an element in a dynamic array by a substring value, use the **FINDSTR** statement. To return the value of an element by specifying its dynamic array location, use the **EXTRACT** function.

Examples

The following example uses the **LOCATE** statement to find the second value from the first field of a dynamic array:

```
cities="New York":@VM:"London":@VM:
"Chicago":@VM:"Boston":@VM:"Los Angeles"
LOCATE "London" IN cities<1> SETTING a
    THEN PRINT "found",a;    ! returns 1 (found in FM 1)
    ELSE PRINT "not found",a
LOCATE "London" IN cities<2> SETTING a
    THEN PRINT "found",a
    ELSE PRINT "not found",a;    ! returns 2 (append FM 2)
LOCATE "London" IN cities<1,3> SETTING a
    THEN PRINT "found",a;    ! returns 1 (found in FM 1, VM 3)
    ELSE PRINT "not found",a
```

See Also

- [FIND](#) statement
- [FINDSTR](#) statement
- [EXTRACT](#) function
- [Dynamic Arrays](#)

- [Strings](#)
- [Variables](#)

LOCK

Obtains a logical process lock.

```
LOCK expression [THEN statements][ELSE statements]
```

Arguments

<i>expression</i>	A number or string, or an expression that evaluates to a number or string specifying a lock. Commonly, an integer from 0 through 64.
-------------------	--

Description

The **LOCK** statement establishes a process lock, preventing other processes from obtaining a lock on the same resource.

Each time a lock is obtained on an *expression* a lock count is incremented for this *expression*. **UNLOCK** decrements this count. Only when the lock count falls to zero will the logical lock be released. For this reason, you should balance each successful invocation of **LOCK** with a corresponding invocation of **UNLOCK**.

Commonly, *expression* evaluates to an integer in the range 0 through 64. However, in Caché any number or string may be specified as a logical lock.

You can optionally specify a **THEN** clause, an **ELSE** clause, or both a **THEN** and an **ELSE** clause. If a lock on *expression* is obtained, the **THEN** clause is executed. If a lock on *expression* could not be obtained, the **ELSE** clause is executed.

Unlike **READU** locks, process locks set in a program are not released automatically when the program terminates. The lock belongs to the process, and persists for the life of the process, unless unlocked explicitly using the **UNLOCK** statement.

Example

The following example uses the **LOCK** statement to obtain a logical lock on an *expression*.

```
a=17
LOCK a THEN PRINT "Got the Lock"
  ELSE PRINT "Couldn't get the lock"
  .
  .
  .
UNLOCK a
```

See Also

- [UNLOCK](#) statement

LOOP...REPEAT

Repeats a block of statements while a condition is true or until a condition becomes true.

```
LOOP [{WHILE | UNTIL} condition]
[DO] statements
REPEAT

LOOP statements
[{WHILE | UNTIL} condition]
[DO] REPEAT
```

Arguments

<i>condition</i>	Numeric or string expression that evaluates to True or False .
<i>statements</i>	One or more statements that are repeated while or until condition is True .

Description

The **LOOP...REPEAT** statement is a flow-of-control statement that repeats a block of program statements zero or more times. The loop is performed either **UNTIL** *condition* becomes true, or **WHILE** *condition* remains true. The two syntax forms are equivalent.

The **REPEAT** keyword is mandatory, signalling the end point of the loop. The **DO** keyword is optional.

You can use the **CONTINUE** statement to cause execution to jump to the next iteration of the loop.

Examples

The following examples illustrate use of the **LOOP...REPEAT** statement. All four examples are exactly equivalent; each executes the loop 10 times:

```
x=0
LOOP UNTIL x=10
  PRINT RND(100)
  ! Generate a random number between 1 and 100
  x=x+1
REPEAT
```

```
x=0
LOOP WHILE x<10
  PRINT RND(100)
  ! Generate a random number between 1 and 100
  x=x+1
REPEAT
```

```
x=0
LOOP
  PRINT RND(100)
  ! Generate a random number between 1 and 100
  x=x+1
  UNTIL x=10
REPEAT
```

```
x=0
LOOP
  PRINT RND(100)
  ! Generate a random number between 1 and 100
  x=x+1
  WHILE x<10
REPEAT
```

See Also

- [CONTINUE](#) statement
- [EXIT](#) statement
- [FOR...NEXT](#) statement

NAP

Suspends processing for a specified number of milliseconds.

```
NAP [milliseconds]
```

Arguments

<i>milliseconds</i>	<i>Optional</i> — An integer count of milliseconds. If omitted, execution is suspended for 1 millisecond.
---------------------	---

Description

The **NAP** statement specifies the number of milliseconds to suspend program execution. There are one thousand milliseconds in a second. If you specify **NAP** with no argument, it suspends program execution for one millisecond.

The **SLEEP** and **RQM** statements can be used to suspend program execution for a specified number of seconds.

See Also

- [SLEEP](#) statement
- [RQM](#) statement

ON

Transfers program execution to one of several internal subroutines or labels.

```
ON integer GOSUB label1[,label2][...]  
ON integer GOTO label1[,label2][...]
```

Arguments

<i>integer</i>	A positive non-zero integer that corresponds to the list of <i>labels</i> .
<i>label</i>	Any valid label . The <i>label</i> name can be optionally followed by a colon (:).

Description

The **ON** statement is used to transfer execution to one of the *labels* specified by the **GOSUB** or **GOTO** keyword. Which label to transfer execution to is specified by the *integer* argument: a value of 1 transfers control to the first listed *label*, a value of 2 transfers control to the second listed *label*, and so forth.

For a **GOTO**, this label identifies a line of code in the current program. For a **GOSUB**, this label identifies an internal subroutine that is executed until a **RETURN** statement is encountered. Execution then reverts to the line immediately following the **ON...GOSUB** statement. (Execution of an internal subroutine can also terminate with an **END** statement, which does not return control.)

The *label* argument value corresponds to line of code identified by a [label](#) identifier. Non-numeric labels end with a colon character; this colon is option when specifying the *label* argument.

See Also

- [GOSUB](#) statement
- [GOTO](#) statement
- [RETURN](#) statement
- [Labels](#)

ON ERROR GOTO

Enables an error-handling routine and specifies the location of the routine within a procedure.

```
ON ERROR GOTO [ line | 0 ]
```

Arguments

The **ON ERROR GOTO** statement has the following argument:

<i>line</i>	The line argument is any line label.
-------------	--------------------------------------

Description

Enables the error-handling routine that starts at the line specified in the required line argument. If a runtime error occurs, control branches to the specified line, making the error handler active. The specified line must be in the same procedure as the **ON ERROR GOTO** statement, or a compile-time error will occur.

Use **ON ERROR GOTO 0** to disable error handling if you have previously enabled it.

Error-handling routines rely on the value in the Number property of the **Err** object to determine the cause of the error. The routine should test or save relevant property values in the **Err** object before any other error can occur or before a procedure that might cause an error is called. The property values in the **Err** object reflect only the most recent error. The error message associated with **Err.Number** is contained in **Err.Description**.

Examples

The following example shows the use of the **ON ERROR GOTO** statement:

```
Print ErrorTest(1)
Print ErrorTest(0)

Function ErrorTest(Arg)
    On Error Goto Label
    return 1/Arg
Label:
    Print "Error ", Err.Number, " ", Err.Description, " ", Err.Source
    Err.Clear
    return 0
End Function
```

Notes

An error-handling routine is not a **Sub** procedure or a **Function** procedure. It is a section of code marked by a line label.

OPEN

Opens a MultiValue file.

```
OPEN [DATA, | DICT,] mvfile TO filevar [THEN statements][ELSE
statements]
```

Arguments

DATA DICT	<i>Optional</i> — A keyword specifying whether to access the MultiValue data file or the dictionary file. The default is to access the data file.
<i>mvfile</i>	The name of a MultiValue file defined in the VOC, specified as a quoted string. If there are multiple data files, you can specify <i>mvfile</i> as either "dirname,datafile" or simply "datafile".
<i>filevar</i>	A local variable name assigned to the MultiValue file. For local variable naming conventions, refer to the Variables page of this manual.

Description

The **OPEN** statement is used to open a MultiValue file. This must be an existing file defined as a file in the VOC. You can create a MultiValue file by using the **CREATE.FILE** verb.

The **OPEN** statement assigns a *filevar* variable to the specified MultiValue file. *filevar* is a local variable specific to the current process. You use this *filevar* variable to refer to the MultiValue file in subsequent **READ**, **WRITE**, and other file statements. Issuing a **CLOSE** statement deletes the *filevar* value.

A process can successfully issue multiple concurrent **OPEN** statements against the same MultiValue file. Multiple processes can issue concurrent **OPEN** statements against the same MultiValue file.

You can optionally specify a **THEN** clause, an **ELSE** clause, or both a **THEN** and an **ELSE** clause. If the file open is successful (the specified file exists), the **THEN** clause is executed. If file open fails (the specified file does not exist), the **ELSE** clause is executed. You can use the **STATUS** function to determine the status of the file open operation, as follows: 0=success; -1=file does not exist.

After opening a file, you can use the **STATUS** statement to obtain file status information. You can use the **FILEINFO** function to get information about an open file.

OPEN is used to open a MultiValue file for **READ** and **WRITE** access. Use **OPENSEQ** to open a sequential file.

See Also

- [READ](#) statement
- [WRITE](#) statement
- [CLOSE](#) statement
- [OPENSEQ](#) statement
- [STATUS](#) statement
- [FILEINFO](#) function
- [STATUS](#) function

OPENPATH

Opens a directory.

```
OPENPATH pathname
                [TO filevariable]
                [ON ERROR statements]
                [ELSE statements]
                [THEN statements [ELSE statements]]
```

Arguments

<i>pathname</i>	<i>Optional</i> — A fully-qualified pathname of a directory, specified as a quoted string. For example: "C:\foo\"
<i>filevariable</i>	<i>Optional</i> — A structure that contains the file type and the file name. For details, see below.

Description

OPENPATH opens a directory. Each filename within the directory is represented as a record ID. Subsequent **READ** statements specify these files using the record IDs.

The *filevariable* string has the following structure:

```
$_C(128)_$_MVV(1)_$_C(FileType)_$_C(DictFlag)_$_LIST(FileName1[,FileName2])
```

The *FileType* codes are as follows: 0=Select List; 1=OS File; 2=Directory; 3=Global

The *DictFlag* codes are as follows: 0=data file; 1=dictionary file

FileName1 specifies (depending on the *FileType*) the file name, directory name, or global name.

FileName2 is only specified for *FileTypes* 2 and 3: If *FileType* 2=the VOC id for the OS file name. If *FileType* 3 = the VOC id for the file name.

See Also

- [OPEN](#) statement
- [OPENSEQ](#) statement
- [READ](#) statement
- [WRITE](#) statement

OPENSEQ

Opens a file for sequential access.

```
OPENSEQ pathname TO filevar [THEN statements][ELSE statements]
```

Arguments

<i>pathname</i>	A fully-qualified Windows or UNIX file pathname, specified as a quoted string.
<i>filevar</i>	A file variable name used to refer to the file in Caché MVBasic.

Description

The **OPENSEQ** statement is used to open a file for sequential access. This can be an existing file or a new file. It assigns the file to *filevar*. Issuing **OPENSEQ** gives the process exclusive access to the specified file.

You can optionally specify a **THEN** clause, an **ELSE** clause, or both a **THEN** and an **ELSE** clause. If the file open is successful (the specified file exists), the **THEN** clause is executed. If file open fails (the specified file does not exist), the **ELSE** clause is executed. You can use the **STATUS** function to determine the status of the sequential file open operation, as follows: 0=success; -1=file does not exist.

To create a file, you must first issue an **OPENSEQ** statement, giving the fully-qualified pathname for the file you wish to create. Because the file does not yet exist, the **OPENSEQ** appears to fail, taking its **ELSE** clause and setting the value returned by the **STATUS** function to -1. However, the **OPENSEQ** sets its *filevar* to an identifier for the specified file pathname. You then supply this *filevar* to **CREATE** to create a new file.

The *pathname* must be a fully-qualified pathname. The directories specified in *pathname* must exist for a file create to be successful.

After opening a file, you can use the **STATUS** statement to obtain file status information. You can use **READBLK**, **READSEQ**, **WRITEBLK**, and **WRITESEQ** to perform sequential read and write operations. You can use **CLOSESEQ** to release an open file, making it available to other processes.

FILEINFO and @FILENAME

You can use the **FILEINFO** function to return sequential file information, including whether a specified *filevar* has been defined (*key*=0) and the *pathname* specified in **OPENSEQ** for that *filevar* (*key*=2). The **@FILENAME** variable also contains the *pathname* specified in the most recent **OPENSEQ**.

In both cases, the file does not have to exist; if **OPENSEQ** specifies a non-existent file, both **FILEINFO** and **@FILENAME** return the specified pathname as a directory path. Subsequently creating this file does not change the **FILEINFO** and **@FILENAME** pathname values. If the file does not exist, the **FILEINFO** file type (*key*=3) is 0. Creating the file changes this **FILEINFO** file type to 5.

Examples

The following example opens a sequential file on a Windows system and writes a line to it. If the file does not exist, it creates the file:

```
OPENSEQ "C:/myfiles/test1" TO mytest
  IF STATUS()=0
  THEN
    WRITESEQ "John Doe" TO mytest
    CLOSESEQ mytest
  END
  ELSE
    CREATE mytest
    IF STATUS()=0
    THEN
      WRITESEQ "John Doe" TO mytest
      CLOSESEQ mytest
    END
    ELSE
      PRINT "File create failed"
    END
  END
END
```

See Also

- [CREATE](#) statement
- [CLOSESEQ](#) statement
- [STATUS](#) statement
- [FILEINFO](#) function
- [STATUS](#) function
- [@FILENAME](#) variable

PERFORM

Executes a MultiValue command from a program and returns.

PERFORM command

Arguments

<i>command</i>	One or more MultiValue commands, each command specified as a quoted string. To specify multiple commands, separate the commands with a Field Mark.
----------------	--

Description

The **PERFORM** command executes the specified Caché MultiValue command(s), then resumes execution of the MVBasic program.

EXECUTE, PERFORM, and CHAIN

The **EXECUTE** command executes one or more MultiValue commands from within MVBasic, then returns execution to the next MVBasic statement in the invoking program. **EXECUTE** can pass values to the MultiValue command(s) and return values from the MultiValue command(s).

The **PERFORM** command executes one or more MultiValue commands from within MVBasic, then returns execution to the next MVBasic statement in the invoking program. **PERFORM** cannot pass or return values.

The **CHAIN** command executes a single MultiValue command from within MVBasic. It does not return execution to the invoking program. **CHAIN** cannot pass values.

Examples

The following example shows how to use **PERFORM** to execute multiple MultiValue commands:

```
PRINT TIME()
PERFORM "SLEEP 2":@FM:"SLEEP 3"
PRINT TIME()
```

See Also

- [CHAIN](#) statement
- [EXECUTE](#) statement

- Caché ObjectScript: XECUTE command

PRECISION

Specifies the maximum number of decimal digits when transforming a floating point number.

```
PRECISION int
```

Arguments

<i>int</i>	An integer specifying the maximum number of decimal digits.
------------	---

Description

The **PRECISION** statement specifies the maximum number of decimal digits to display when converting a floating point number. It rounds the decimal portion to the number of decimal digits specified in *int*. The default number of decimal digits is 4.

PRECISION *does not* add trailing zeros to numbers with fewer decimal digits than specified.

If *int* is 0, the null string, or a non-numeric string, **PRECISION** rounds all decimal digits to the nearest integer value.

Examples

The following example illustrates use of the **PRECISION** statement to provide a precision value to the **FIX** function:

```
mynum=123.987654321
PRINT FIX(mynum)
    ! returns 123.9877
    ! (rounds to the default precision of 4)
PRECISION 2
PRINT FIX(mynum)
    ! returns 123.99
    ! (rounds to a precision of 2)
```

See Also

- [FIX](#) function

PRINT

Prints to the terminal or to a specified device.

```
PRINT [ON channel] [printlist]
```

Arguments

ON <i>channel</i>	<i>Optional</i> — The ON clause specifies a print channel as an integer value of -1 through 255. If not specified, the print channel defaults to 0, which is the current terminal session screen.
<i>printlist</i>	<i>Optional</i> — Any MVBASIC expression that resolves to a quoted string. You can also specify a series of quoted strings, separated by either commas (,) or colons (:). A comma inserts a pre-defined tab spacing between the two strings. A colon concatenates the two strings. If <i>printlist</i> is omitted, a blank line is returned.

Description

PRINT displays the items specified in *printlist* to the screen, or to the device specified by the ON *channel* clause. If no *printlist* is specified, **PRINT** returns a blank line.

If you use a comma to separate strings in the *printlist*, a tab is inserted between the two items. By default, tabs are set at ten column intervals. To add spaces between items, use the **SPACE** function.

If you use a colon to separate strings in the *printlist*, the strings are concatenated. By default, a **PRINT** statement ends by issuing a linefeed and carriage return. However, if you end the **PRINT** argument with a colon, **PRINT** does not issue the linefeed and carriage return. This enables you to concatenate the output of the next statement to the **PRINT** output.

You can use an @ function to specify the column position at which to print. For example, `PRINT @(15):"Over here!"` prints the literal string starting at column 16.

The **PRINT** (without the ON clause), **DISPLAY**, and **CRT** commands are identical.

Examples

The following examples illustrate the use of the **PRINT** statement:

```
PRINT "hello", "world!"
```

returns:

```
hello world!
```

```
PRINT "hello":"world!"
```

returns:

```
helloworld!
```

```
PRINT "hello"  
PRINT "world!"
```

returns:

```
hello  
world!
```

See Also

- [CRT](#) statement
- [DISPLAY](#) statement
- [ECHO](#) statement
- [LPTR](#) statement
- [SPACE](#) function
- [SPOOLER](#) function

PRINTER

Specifies whether to direct output to the printer.

```
PRINTER ON  
PRINTER OFF  
PRINTER CLOSE
```

Description

The **PRINTER ON** statement directs output to the printer. After setting this option, **PRINT** statements direct their output to the print buffer. This print buffer is spooled to the printer when you issue a **PRINTER CLOSE** statement, or the program terminates. The **PRINTER OFF** statement directs subsequent output to the screen (the default output device).

The **PRINTER ON** command has no effect on **CRT** statements, which always output to the screen.

See Also

- [PRINT](#) statement
- [SPOOLER](#) function

PROG (PROGRAM)

Specifies the program name.

```
PROG name
PROGRAM name
```

Arguments

<i>name</i>	A name used to identify the current program. Must be a valid identifier.
-------------	--

Description

The **PROGRAM** statement is used to specify a name for the current program. It must appear as the first non-comment line of the program.

See Also

- [Labels](#)
- [Comments](#)

PROMPT

Sets the user input prompt.

```
PROMPT string
```

Arguments

<i>string</i>	A quoted string of one or more characters to use as the user input prompt.
---------------	--

Description

The **PROMPT** statement sets the user input prompt to the character (or characters) specified in *string*. The default prompt is the question mark (?) character.

The user input prompt is used by the **INPUT** statement. However, if a **DATA** statement is specified, **INPUT** does not display the input prompt.

See Also

- [DATA](#) statement
- [INPUT](#) statement

RANDOMIZE

Initializes the random-number generator.

```
RANDOMIZE number
```

Arguments

<i>number</i>	Any valid numeric expression.
---------------	-------------------------------

Description

The **RANDOMIZE** statement uses *number* to initialize the **RND** function's random-number generator, giving it a new seed value. By specifying the same **RANDOMIZE** *number* seed, you can use **RND** to repeatedly generate the same series of random numbers. If you omit *number*, the value returned by the system internal clock is used as the new seed value.

If **RANDOMIZE** is not used, the **RND** function (with no arguments) uses the same number as a seed the first time it is called, and thereafter uses the last generated number as a seed value.

Examples

The following example illustrates use of the **RANDOMIZE** statement:

```
RANDOMIZE 10;      ! Initialize random-number generator.  
MyValue = RND(7)  
    ! Generate random value between 1 and 6.  
PRINT MyValue
```

See Also

- [RND](#) function

READ, READU, READV, READVU

Reads data from a MultiValue file.

```

READ dynarray FROM filevar,recID [THEN statements] [ELSE statements]

READU dynarray FROM filevar,recID [LOCKED statements] [THEN statements]
  [ELSE statements]

READV dynarray FROM filevar,recID,fieldno [THEN statements] [ELSE
statements]

READVU dynarray FROM filevar,recID,fieldno [LOCKED statements] [THEN
statements] [ELSE statements]

```

Arguments

<i>dynarray</i>	A dynamic array used to receive the field values from the file.
<i>filevar</i>	A local variable used as the file identifier of an open MultiValue file. This variable is set by the OPEN statement.
<i>recID</i>	The record ID of a record to be read, specified as an integer.
<i>fieldno</i>	The field number of the field to be read, specified as an integer. Used with READV and READVU . If 0, returns the <i>recID</i> .
<i>LOCKED statements</i>	<i>Optional</i> — One or more MVBasic statements executed if READU or READVU could not perform a read due to lock contention.
<i>THEN statements</i>	<i>Optional</i> — One or more MVBasic statements executed if READ has successfully read field values into <i>dynarray</i> . Reading a null string is considered a successful read.
<i>ELSE statements</i>	<i>Optional</i> — One or more MVBasic statements executed if READ could not read a field value.

Description

These read statements read a value from a MultiValue file into a dynamic array. The **READ** and **READU** statements read the specified record into *dynarray*. The **READV** and **READVU** statements reads the specified field within a record into *dynarray*.

The **READU** and **READVU** statements contain an optional **LOCKED** clause.

You must use the **OPEN** statement to open the MultiValue file before issuing any of these **READ** statements.

The optional **THEN** clause and **ELSE** clause specify one or more MVBASIC statements to execute depending on the status of the read operation. **READ** executes the **THEN** clause if the read was successful. The **THEN** clause is executed even when all remaining field identifiers are the null string. **READ** executes the **ELSE** clause if the read operation fails.

Reading a Record

READ (and **READU**) reads the specified MultiValue file record value into *dynarray*. If *recID* refers to a non-existent record, the read operation fails.

Reading a Field

READV (and **READVU**) reads the specified field value from the specified MultiValue file record into *dynarray*. If *fieldno* is 0, **READV** returns the specified *recID* to *dynarray*. If *fieldno* refers to a non-existent field, **READV** returns the null string to *dynarray*. If *recID* refers to a non-existent record and *fieldno* is not 0, the read operation fails.

Examples

The following example illustrates the use of the **READ** statement:

```
OPEN "TEST.FILE" TO myfile
READ mydyn FROM myfile,1
PRINT "the record value:",mydyn
```

The following example illustrates the use of the **READV** statement:

```
OPEN "TEST.FILE" TO myfile
READV mydyn FROM myfile,1,1
PRINT "the field value:",mydyn
```

See Also

- [OPEN](#) statement
- [WRITE](#) statement
- [CLOSE](#) statement

- [STATUS](#) statement
- [Dynamic Arrays](#)

READBLK

Reads a block of data from a sequential file.

```
READBLK data FROM filevar,blksize [THEN statements][ELSE statements]
```

Arguments

<i>data</i>	Name of a variable used to receive a block of data from a file.
<i>filevar</i>	A file variable name used to refer to the file in Caché MVBasic. This <i>filevar</i> is obtained from OPENSEQ .
<i>blksize</i>	A positive integer specifying the block size, in bytes.

Description

The **READBLK** statement is used to read a block of data of a specified size from a file that has been opened for sequential access using **OPENSEQ**. This block of data is written to the *data* variable. The specified *blksize* can be any size.

When invoked, **READBLK** increments a pointer to the end of the data just read, so that repeated invocations of **READBLK** read sequentially through the file data. The same file pointer is used by **READBLK** and **WRITEBLK**. If the file contains less data than *blksize*, the available data is read.

You can determine the current position of this pointer using the **STATUS** statement. You can reposition this pointer using the **SEEK** statement.

You can optionally specify a **THEN** clause, an **ELSE** clause, or both a **THEN** and an **ELSE** clause. If the file read is successful, the **THEN** clause is executed. If file read fails, or if the end of the file is reached, the **ELSE** clause is executed.

You can use the **STATUS** function to determine the status of the read operation, as follows: 0=sequential read successful; -1=read failed because file not open (or opened by another process); 1=end-of-file encountered; 2=read timed out.

READBLK and READSEQ

The **READBLK** command retrieves data from a sequential file in blocks of a specified length. These blocks may be of any length, and have no necessary relationship to the length of logical data units, such as lines or records, within the file. The **READSEQ** command retrieves a single line of data from a sequential file. A line of data is identified by the presence of end-of-line characters. A line of data may be of any size.

Examples

The following example reads the first 100 bytes of data from an existing sequential file on a Windows system:

```
OPENSEQ "C:\myfiles\test1" TO mytest
IF STATUS()=0
THEN
  READBLK mydata FROM mytest,100
  IF mydata=""
  THEN PRINT "no data"
  END
  ELSE PRINT mydata
  END
WEOFSEQ mytest
CLOSESEQ mytest
END
ELSE
  PRINT "File open failed"
END
```

See Also

- [OPENSEQ](#) statement
- [WRITEBLK](#) statement
- [READSEQ](#) statement
- [SEEK](#) statement
- [STATUS](#) statement
- [STATUS](#) function

READLIST

Reads the remaining field ids from a select list.

```
READLIST dynarray FROM slist [THEN statements] [ELSE statements]
```

Arguments

<i>dynarray</i>	A dynamic array used to receive the field values from the select list.
<i>slist</i>	A select list. This can be a numbered select list specified as an integer from 0 through 10, or a named select list specified as a variable name.
THEN <i>statements</i>	<i>Optional</i> — One or more MVBasic statements executed if READLIST has successfully read field values into <i>dynarray</i> . Reading a null string is considered a successful read.
ELSE <i>statements</i>	<i>Optional</i> — One or more MVBasic statements executed if READLIST could not read a field value.

Description

The **READLIST** statement reads all remaining field identifiers from a select list into a dynamic array. If no reads have been performed on the select list, **READLIST** reads the entire select list into *dynarray*. If a **READNEXT** has been performed on the select list, **READLIST** reads the remaining select list field identifiers into *dynarray*.

You can use **SELECT** or **SELECTV** to create a select list. **SELECT** specifies a numbered select list. **SELECTV** specifies a named select list. **SELECTE** copies the default numbered select list (Select List 0) to a named select list.

The optional THEN clause and ELSE clause specify one or more MVBasic statements to execute depending on the status of the read operation. **READLIST** executes the THEN clause if the select list pointer has not reached the end of the select list. The THEN clause is executed even when all remaining field identifiers are the null string. **READLIST** executes the ELSE clause if the select list pointer has reached the end of the select list, or the select list does not exist.

Examples

The following example illustrates the use of the **READLIST** statement. **SELECT** copies all of the field mark identifiers into Select List 4. A **READNEXT** reads the first field mark

identifier from Select List 4 into the *area* variable. A **READLIST** then reads all the remaining field mark identifiers from Select List 4 into the *dynarea* dynamic array:

```
regions="Northeast":@FM:"Southeast":@FM:"Northwest":@FM:"Southwest"
SELECT regions TO 4 ON ERROR PRINT "Select failed"
READNEXT area FROM 4 THEN PRINT area ELSE PRINT "no fields"
    ! returns "Northeast"
READLIST dynarea FROM 4 THEN PRINT dynarea ELSE PRINT "no fields"
    ! returns "SoutheastfNorthwestfSouthwest"
READLIST dynarea FROM 4 THEN PRINT dynarea ELSE PRINT "no fields"
    ! returns "no fields"
```

See Also

- [SELECT](#) statement
- [SELECTV](#) statement
- [READNEXT](#) statement
- [Dynamic Arrays](#)

READNEXT

Reads the next field id from a select list.

```
READNEXT fieldval FROM slist [THEN statements] [ELSE statements]
```

Arguments

<i>fieldval</i>	A variable used to receive a field value from the select list.
<i>slist</i>	A select list. This can be a numbered select list specified as an integer from 0 through 10, or a named select list specified as a variable name.
THEN <i>statements</i>	<i>Optional</i> — One or more MVBASIC statements executed if READNEXT has successfully read a field value into <i>fieldval</i> . Reading a null string is considered a successful read.
ELSE <i>statements</i>	<i>Optional</i> — One or more MVBASIC statements executed if READNEXT could not read a field value.

Description

The **READNEXT** statement reads successive field identifiers from a select list, one field identifier per invocation. The field identifier is read from the *slist* select list into the *feldval* variable.

You can use **SELECT** or **SELECTV** to create a select list. **SELECT** specifies a numbered select list. **SELECTV** specifies a named select list. **SELECTE** copies the default numbered select list (Select List 0) to a named select list.

The optional THEN clause and ELSE clause specify one or more MVBasic statements to execute depending on the status of the read operation. **READNEXT** executes the THEN clause if the select list pointer has not reached the end of the select list. The THEN clause is executed even when a field identifier is the null string. **READNEXT** executes the ELSE clause if the select list pointer has reached the end of the select list, or the select list does not exist.

Note: **READNEXT** reads a single field identifier from a select list into a variable. **READLIST** reads all remaining field identifiers from a select list into a dynamic array.

Examples

The following example illustrates the use of the **READNEXT** statement. **SELECT** copies all of the field mark identifiers into Select List 4. Each iteration of **READNEXT** reads the next field mark identifier from Select List 4 into the *area* variable:

```
regions="Northeast":@FM:"Southeast":@FM:"Northwest":@FM:"Southwest"
SELECT regions TO 4 ON ERROR PRINT "Select failed"
FOR x=1 TO 5
  READNEXT area FROM 4
  PRINT area
NEXT
```

The following example illustrates the use of **READNEXT** with the THEN and ELSE clauses. **SELECTV** copies all of the field mark identifiers into Select List *mylist*. **READNEXT** reads the next field mark identifier from Select List *mylist* into the *area* variable:

```
regions="Northeast":@FM:"Southeast":@FM:"Northwest":@FM:"Southwest"
SELECT regions TO mylist ON ERROR PRINT "Select failed"
x=1
LOOP WHILE x=1
  READNEXT area FROM mylist THEN PRINT area ELSE x=0
REPEAT
```

See Also

- [SELECT](#) statement
- [SELECTV](#) statement
- [READLIST](#) statement
- [Dynamic Arrays](#)

READSEQ

Reads a line of data from a sequential file.

```
READSEQ data FROM filevar [THEN statements][ELSE statements]
```

Arguments

<i>data</i>	Name of a variable used to receive a line of data from a file.
<i>filevar</i>	A file variable name used to refer to the file in Caché MVBasic. This <i>filevar</i> is obtained from OPENSEQ .

Description

The **READSEQ** statement is used to read a line of data from a file that has been opened for sequential access using **OPENSEQ**. This line of data is written to the *data* variable.

A line of data is defined as a unit of data terminated by a newline character. Newline characters are not returned as part of *data*. When invoked, **READSEQ** increments a pointer to the next sequential unit of data, so that repeated invocations of **READSEQ** read sequentially through the file data. The same file pointer is used by **READSEQ** and **WRITESEQ**.

You can determine the current position of this pointer using the **STATUS** statement. You can reposition this pointer using the **SEEK** statement.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If the file read is successful, the THEN clause is executed. If file read fails, or if the end of the file is reached, the ELSE clause is executed.

You can use the **STATUS** function to determine the status of the read operation, as follows: 0=sequential read successful; -1=read failed because file not open (or opened by another process); 1=end-of-file encountered; 2=read timed out.

READSEQ and READBLK

The **READSEQ** command retrieves a single line of data from a sequential file. A line of data is identified by the presence of end-of-line characters. A line of data may be of any size. The **READBLK** command retrieves data from a sequential file in blocks of a specified length. These blocks may be of any length, and have no necessary relationship to the length of logical data units, such as lines or records, within the file.

Examples

The following example reads the first line of data from an existing sequential file on a Windows system:

```
OPENSEQ "C:\myfiles\test1" TO mytest
IF STATUS()=0
THEN
  READSEQ mydata FROM mytest
  IF mydata=""
  THEN PRINT "no data"
  END
  ELSE PRINT "the first line:",mydata
  END
WEOFSEQ mytest
CLOSESEQ mytest
END
ELSE
  PRINT "File open failed"
END
```

See Also

- [OPENSEQ](#) statement
- [WRITESEQ](#) statement
- [READBLK](#) statement
- [CLOSESEQ](#) statement
- [SEEK](#) statement
- [STATUS](#) statement
- [STATUS](#) function

RELEASE

Releases record locks.

```
RELEASE [filevar [,recID]]
```

Arguments

<i>filevar</i>	<i>Optional</i> — A file variable name used to refer to a MultiValue file. This <i>filevar</i> is supplied by the OPEN statement.
<i>recID</i>	<i>Optional</i> — The record ID for which record locks are to be released.

Description

A **RELEASE** statement with no argument releases all record locks held by the current process. A **RELEASE** statement with the *filevar* argument releases all record locks on the specified MultiValue file held by the current process. A **RELEASE** statement with the *filevar* and *recID* arguments releases the record lock for the specified record on the specified MultiValue file held by the current process.

See Also

- [OPEN](#) statement
- [CLOSE](#) statement
- [STATUS](#) statement
- [Dynamic Arrays](#)

REM

Includes a comment in a program.

```
REM comment
```

Arguments

None.

The *comment* argument is the text of any comment you want to include. After the **REM** keyword, a space is required before *comment*.

Description

You can use the **REM** statement to include comments in the source code of your program. A comment can be on a separate line, or on the same line as an executable statement. If you include a comment on the same line as an executable statement, the statement must be ended with a semicolon (;) before the comment indicator.

The **REM** statement is one of several single-line comment indicators. You can also use the exclamation mark (!), asterisk (*), or dollar sign asterisk (\$*) to indicate a comment. Regardless of which indicator you use, all comments are single-line comments; you must specify a comment indicator for every line of a comment.

Note: Caché MVBASIC contains both a REM (remarks) statement and a REM (remainder) function. These are completely unrelated and should not be confused.

Examples

The following example illustrates the use of the **REM** statement:

```
MyStr1="Hello"; REM Comment after a statement.
MyStr2 = "Goodbye"
  REM This is also a comment.
PRINT MyStr1,Mystr2; REM comment (note semicolon)
  ! This too is a comment.
  * This too is a comment.
  $* This too is a comment.
```

See Also

- [Comments](#)

REMOVE

Extracts sequential elements of a dynamic array.

```
REMOVE var FROM dynarray SETTING n
```

Arguments

<i>var</i>	A variable used to receive the extracted element value.
<i>dynarray</i>	A dynamic array from which successive data values are to be extracted.
<i>n</i>	An integer code for the dynamic array delimiter type.

Description

The **REMOVE** statement efficiently extracts successive data values from a dynamic array. The extracted element value is placed in the *var* variable. **REMOVE** operates on a single dynamic array level; you specify the level delimiter using the *n* argument. **REMOVE** maintains an internal pointer so that repeated calls return successive element values. When the last element value has been extracted, **REMOVE** sets *var* to the empty string.

You can use the **GETREM** function to return the character position in *dynarray* of the **REMOVE** pointer.

Note: The **REMOVE** statement is identical to the **REVREMOVE** statement, except that **REVREMOVE** operates in the reverse direction. The **REMOVE** function, **REMOVE** statement, and **REVREMOVE** statement all share the same pointer. It is incremented by a remove and decremented by a revremove.

The *n* integer code values are as follows:

0	End of file
1	@IM Item Mark CHAR(255)
2	@FM Field Mark CHAR(254)
3	@VM Value Mark CHAR(253)
4	@SM Subvalue Mark CHAR(252)
5	@TM Text Mark CHAR(251)

Examples

The following example successively extracts the first 5 Value Mark elements from a dynamic array:

```
names="Fred":@VM:"Barney":@VM:"Wilma":@VM:"Betty"
FOR x=1 TO 5
  REMOVE val FROM names SETTING 3
  PRINT val
  ! Returns:
  !   Fred
  !   Barney
  !   Wilma
  !   Betty
  !   " "
NEXT
```

See Also

- [REVREMOVE](#) statement
- [EXTRACT](#) function
- [GETREM](#) function
- [REMOVE](#) function

RETURN

Returns from a subroutine to the statement that called it.

```
RETURN [TO label]
```

Arguments

<i>TO label</i>	<i>Optional</i> — Any valid label . The <i>label</i> name can be optionally followed by a colon (:)
-----------------	---

Description

The **RETURN** statement with no argument ceases execution of a subroutine and returns control to the **GOSUB** statement (for an internal subroutine) or the **CALL** statement (for an external subroutine) that invoked the subroutine. Program execution resumes with the line immediately following the **GOSUB** or **CALL**.

You can terminate an external subroutine with a **RETURN** or with an **END** statement.

The **RETURN** statement with a *TO label* clause ceases execution of a subroutine and transfers execution to the internal subroutine identified by the specified label.

See Also

- [CALL](#) statement
- [GOSUB](#) statement
- [END](#) statement
- [FUNCTION](#) statement
- [Labels](#)

REVREMOVE

Extracts sequential elements of a dynamic array in reverse order.

```
REVREMOVE var FROM dynarray SETTING n
```

Arguments

<i>var</i>	A variable used to receive the extracted element value.
<i>dynarray</i>	A dynamic array from which successive data values are to be extracted.
<i>n</i>	An integer code for the dynamic array delimiter type.

Description

REVREMOVE efficiently extracts successive data values from a dynamic array beginning at the end of the string. The extracted element value is placed in the *var* variable. **REVREMOVE** operates on a single dynamic array level; you specify the level delimiter using the *n* argument. **REVREMOVE** maintains an internal pointer so that repeated calls return successively previous element values. When the last element value has been extracted, **REMOVE** sets *var* to the empty string.

You can use the **GETREM** function to return the character position in *dynarray* of the **REVREMOVE** pointer.

Note: The **REVREMOVE** statement is identical to the **REMOVE** statement, except that it operates in the reverse direction. The **REMOVE** function, **REMOVE** statement, and **REVREMOVE** statement all share the same pointer. It is incremented by a remove and decremented by a revremove.

The *n* integer code values are as follows:

0	End of file
1	@IM Item Mark CHAR(255)
2	@FM Field Mark CHAR(254)
3	@VM Value Mark CHAR(253)
4	@SM Subvalue Mark CHAR(252)
5	@TM Text Mark CHAR(251)

Examples

The following example successively extracts the last 5 Value Mark elements from a dynamic array:

```
names="Fred":@VM:"Barney":@VM:"Wilma":@VM:"Betty"
FOR x=1 TO 5
  REVREMOVE val FROM names SETTING 3
  PRINT val
  ! Returns:
  !   Betty
  !   Wilma
  !   Barney
  !   Fred
  !   " "
NEXT
```

See Also

- [REMOVE](#) statement
- [EXTRACT](#) function
- [GETREM](#) function
- [REMOVE](#) function

ROLLBACK

Reverts all changes made during the current transaction.

```
ROLLBACK [WORK] [THEN statements][ELSE statements]
```

Description

The **ROLLBACK** statement reverts all changes made during the current transaction initiated by a **BEGIN TRANSACTION** statement. All file changes issued during the transaction are undone, returning the data to the state prior to the **BEGIN TRANSACTION**.

The **WORK** keyword is optional and provides no functionality. It is provided solely for compatibility with other MultiValue vendor products.

You can optionally specify a **THEN** clause, an **ELSE** clause, or both a **THEN** and an **ELSE** clause. If the transaction rollback is successful, the **THEN** clause is executed. If the transaction rollback fails, the **ELSE** clause is executed.

To commit the changes made during the current transaction, issue a **COMMIT** statement, rather than a **ROLLBACK** statement.

After the transaction is closed, program execution continues at the **END TRANSACTION** statement.

Note: Caché MVBasic supports two sets of transaction statements:

- UniVerse-style **BEGIN TRANSACTION**, **COMMIT**, **ROLLBACK**, and **END TRANSACTION**.
- UniData-style **TRANSACTION START**, **TRANSACTION COMMIT**, and **TRANSACTION ABORT**.

These two sets of transaction statements should not be combined.

See Also

- [BEGIN TRANSACTION](#) statement
- [END TRANSACTION](#) statement
- [COMMIT](#) statement

RQM

Suspends processing for a specified number of seconds.

```
RQM [seconds]
RQM [time]
```

Arguments

<i>seconds</i>	<i>Optional</i> — An integer count of seconds. If omitted, execution is suspended for 1 second.
<i>time</i>	<i>Optional</i> — A wakeup time, specified as hh:mm:ss or hh:mm.

Description

The **RQM** statement has two formats. You can either specify the number of seconds to suspend program execution, or specify the time at which to resume execution. If you specify **RQM** with no argument, it suspends program execution for one second. If *seconds* is a decimal number, it is rounded to the nearest whole second.

RQM is a synonym for **SLEEP**.

You can use **NAP** to suspend program execution for a specified number of milliseconds.

See Also

- [NAP](#) statement
- [SLEEP](#) statement

SEEK

Repositions the file pointer for a sequential file.

```
SEEK filevar [,offset [,relto]] [THEN statements][ELSE statements]
```

Arguments

<i>filevar</i>	A file variable name used to refer to the file in Caché MVBasic. This <i>filevar</i> is obtained from OPENSEQ .
<i>offset</i>	<i>Optional</i> — A positive or negative integer count of bytes used to reposition the file pointer relative to the <i>relto</i> position. By default, <i>offset</i> is 0.
<i>relto</i>	<i>Optional</i> — A flag indicating the pointer position is determined relative to some location. The available values are: 0=relative to the beginning of the file; 1=relative to the current pointer position; 2=relative to the end of the file. The default is 0.

Description

The **SEEK** statement is used to position the sequential file pointer in a file that has been opened for sequential access using **OPENSEQ**.

By default, **SEEK** repositions the file pointer to the beginning of the file. **SEEK** can be used to increment or decrement the file pointer from its current position, or from the beginning or end of the file.

You can determine the current position of the file pointer using the **STATUS** statement.

You can optionally specify a **THEN** clause, an **ELSE** clause, or both a **THEN** and an **ELSE** clause. If the pointer reposition is successful, the **THEN** clause is executed. If pointer reposition fails (usually because the specified position is beyond the limits of the file), the **ELSE** clause is executed and the pointer position remains unchanged.

You can use the **STATUS** function to determine the status of the pointer reposition operation, as follows: 0=success; -1=pointer reposition failed because either position is beyond the limits of the file or the file is not open.

See Also

- [OPENSEQ](#) statement
- [READSEQ](#) statement
- [WRITESEQ](#) statement
- [STATUS](#) statement
- [STATUS](#) function

SELECT

Selects field ids into a numbered select list.

```
SELECT dynarray TO listnum [ON ERROR statements]
SELECT filevar TO listnum [ON ERROR statements]
```

Arguments

<i>dynarray</i>	Any valid dynamic array of Field Values.
<i>filevar</i>	A local variable used as the file identifier of an open MultiValue file. This variable is set by the OPEN statement.
<i>listnum</i>	A numbered select list, specified as an integer from 1 through 10.
<i>statements</i>	<i>Optional</i> — One or more statements executed if an error occurs.

Description

The **SELECT** statement selects the field identifiers from a MultiValue file or a dynamic array and places them in a select list. You can then use **READNEXT** to read this select list, one field identifier at a time. Selecting to a select list overwrites any previous values for that select list.

For **SELECT filevar**, you must specify a MultiValue file opened using the **OPEN** statement. The **SELECT** completes successfully even if *filevar* is not defined, but a subsequent **READNEXT** statement fails.

You must specify a *listnum* from 1 through 10; *listnum* 0 is not valid for Caché MVBasic.

Note: **SELECT** specifies a numbered select list. **SELECTV** specifies a named select list. These two statements are otherwise identical. You can use **SELECTE** to copy Select List 0 to a named select list.

The optional ON ERROR clause specifies one or more MVBasic statements to execute if the **SELECT** operation fails. For example, if you specify an invalid *listnum* the ON ERROR statements are executed.

Examples

The following example illustrates the use **SELECT dynarray**. **SELECT** copies all of the field mark identifiers into Select List 4. Each iteration of **READNEXT** reads the next field mark identifier from Select List 4 into the *area* variable:

```
regions="Northeast":@FM:"Southeast":@FM:"Northwest":@FM:"Southwest"  
SELECT regions TO 4 ON ERROR PRINT "Select failed"  
FOR x=1 TO 5  
    READNEXT area FROM 4  
    PRINT area  
NEXT
```

See Also

- [OPEN](#) statement
- [READNEXT](#) statement
- [SELECTE](#) statement
- [SELECTV](#) statement
- [Dynamic Arrays](#)

SELECTE

Copies select list 0 to a named select list.

```
SELECTE TO varname
```

Arguments

<i>varname</i>	A named select list, specified as a variable name.
----------------	--

Description

The **SELECTE** statement copies Select List 0 to a select list named *varname*. You can then use **READNEXT** to read this select list, one field identifier at a time.

Select List 0 is the default select list created by a **SELECT** statement.

Note: **SELECTE** enables you to copy Select List 0 to a named select list. You can create a named select list directly by using **SELECTV**.

Examples

The following example illustrates the use of the **SELECTE** statement. Here **SELECT** copies all of the field mark identifiers into Select List 0. Then **SELECTE** copies Select List 0 to a select list named *rfields*. Each iteration of **READNEXT** reads the next field mark identifier from Select List *rfields* into the *area* variable:

```
regions="Northeast":@FM:"Southeast":@FM:"Northwest":@FM:"Southwest"  
SELECT regions TO 0 ON ERROR PRINT "Select failed"  
SELECTE TO rfields  
FOR x=1 TO 5  
    READNEXT area FROM rfields  
    PRINT area  
NEXT
```

See Also

- [READNEXT](#) statement
- [SELECT](#) statement
- [SELECTV](#) statement
- [Dynamic Arrays](#)

SELECTV

Selects field ids into a named select list.

```
SELECTV dynarray TO varname [ON ERROR statements]
```

Arguments

<i>dynarray</i>	Any valid dynamic array of Field Values.
<i>varname</i>	A named select list, specified as a variable name.
<i>statements</i>	<i>Optional</i> — One or more statements to execute if an error occurs.

Description

The **SELECTV** statement selects the field identifiers from a dynamic array and places them in a named select list. You can then use **READNEXT** to read this select list, one field identifier at a time.

The TO clause specifies a variable as the name of the select list. Selecting to a select list overwrites any previous values for that select list.

Note: **SELECTV** specifies a named select list. **SELECT** specifies a numbered select list. These two statements are otherwise identical.

The optional ON ERROR clause specifies one or more MVBASIC statements to execute if the **SELECTV** operation fails.

Examples

The following example illustrates the use of the **SELECTV** statement. **SELECTV** copies all of the field mark identifiers into a Select List named *rfields*. Each iteration of **READNEXT** reads the next field mark identifier from Select List *rfields* into the *area* variable:

```
regions="Northeast":@FM:"Southeast":@FM:"Northwest":@FM:"Southwest"
SELECTV regions TO rfields ON ERROR PRINT "Select failed"
FOR x=1 TO 5
  READNEXT area FROM rfields
  PRINT area
NEXT
```

See Also

- [READNEXT](#) statement

- [SELECT](#) statement
- [Dynamic Arrays](#)

SLEEP

Suspends processing for a specified number of seconds.

```
SLEEP [seconds]
```

Arguments

<i>seconds</i>	<i>Optional</i> — An integer count of seconds. If omitted, execution is suspended for 1 second.
----------------	---

Description

The **SLEEP** statement specifies the number of seconds to suspend program execution. If you specify **SLEEP** with no argument, it suspends program execution for one second. If *seconds* is a decimal number, it is rounded to the nearest whole second.

RQM is a synonym for **SLEEP**.

You can use **NAP** to suspend program execution for a specified number of milliseconds.

See Also

- [NAP](#) statement
- [RQM](#) statement

STATUS

Provides file status information.

```
STATUS dynarray FROM filevar [THEN statements][ELSE statements]
```

Arguments

<i>dynarray</i>	A dynamic array used by STATUS to hold file information as Field elements.
<i>filevar</i>	A file variable name specifying the file from which status information is to be returned. This <i>filevar</i> is obtained from OPENSEQ .

Description

The **STATUS** statement is used to return status information about a file. This information is returned as Field Mark delimited elements of a dynamic array. You must open the file, using the **OPENSEQ** statement, to obtain the *filevar* required to invoke **STATUS**.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If file status information is obtained, the THEN clause is executed. If file status information could not be obtained, the ELSE clause is executed.

The first field of *dynarray* contains the current position of the sequential file pointer, counting from 0. This count includes the two-character newline (carriage return + line feed) that appears at the end of each line of data in a sequential file. The same file pointer is used by **WRITESEQ** and **READSEQ**. You can reposition this pointer using the **SEEK** statement.

See Also

- [OPENSEQ](#) statement
- [READSEQ](#) statement
- [SEEK](#) statement
- [WRITESEQ](#) statement

STOP

Terminates program execution and returns to the calling environment.

```
STOP [expressionlist]
STOPE [expressionlist]
STOPM [expression]
```

Arguments

<i>expressionlist</i>	The error message file ERRMSG.
<i>expression</i>	A given error message expression, or a comma-separated list of expressions.

Description

The **STOP** statement is used to terminate program execution and return control to the calling environment. If an argument is specified, **STOP** displays this error message before terminating program execution. **STOP** and **STOPE** are functionally identical; both use the ERRMSG file for error messages. **STOPM** uses the literal message text specified in *expression*.

ABORT and STOP

The **ABORT** command terminates all program execution and returns to the programming prompt. The **STOP** terminates the executing routine and returns control to the calling routine.

See Also

- [ABORT](#) statement
- Caché ObjectScript: QUIT command

SUBROUTINE

Defines an external subroutine.

```
SUBROUTINE [name][(argumentlist)]
```

Arguments

<i>name</i>	<i>Optional</i> — Any valid name to assign to the subroutine.
<i>argumentlist</i>	<i>Optional</i> — An argument or a comma-separated list of arguments. The <i>argumentlist</i> is enclosed with parentheses.

Description

The **SUBROUTINE** statement defines an external subroutine. **SUBROUTINE** must be the first non-comment statement in the external subroutine. There can only be one **SUBROUTINE** statement in an external subroutine. The *name* argument allows you to identify the external subroutine; it is not (strictly speaking) required to define or invoke an external subroutine.

An external subroutine must be compiled and cataloged before it can be invoked. You can invoke an external subroutine with a **CALL** statement. The **CALL** statement invokes a subroutine by its name in the catalog; this is not necessarily the same as *name*.

When using **CALL** to invoke a subroutine, you can pass it arguments. The list of arguments passed by **CALL** must correspond in position and number to the number of arguments defined in **SUBROUTINE** to receive the passed values. The names of the arguments do not have to correspond.

The argument list can contain any combination of regular variables and array variables. In *argumentlist*, an array variable name must be preceded by the **MAT** keyword. The following is an argument list that specifies a regular variable and two array variables:

```
SUBROUTINE MySub(myvar,MAT myarray,MAT refarray)
```

By default, all arguments are passed by reference. If the subroutine changes the value of an argument passed by reference, this value is also changed in the calling program. You can specify in the **CALL** statement that an argument is to be passed by value. If the subroutine changes the value of an argument passed by value, the value of this argument in the calling program remains unchanged.

You can also use the **COMMON** statement to make specified variables available to all external subroutines.

You can terminate an external subroutine with a **RETURN** or with an **END** statement. Following a **RETURN**, program execution resumes with the line immediately following the invoking **CALL** statement.

SUBR, CALL, and GOSUB

The **SUBR** function is used to call an external subroutine that returns a value. The **CALL** statement is used to call an external subroutine that does not return a value. The **GOSUB** statement is used to call an internal subroutine.

See Also

- [COMMON](#) statement
- [RETURN](#) statement
- [CALL](#) statement
- [GOSUB](#) statement
- [SUBR](#) function

SWAP

Replaces a substring in a string.

```
SWAP subout WITH subin IN string
```

Arguments

<i>subout</i>	The substring to be replaced. Any expression that resolves to a valid string or numeric.
<i>subin</i>	The substring to be inserted in place of <i>subout</i> . Any expression that resolves to a valid string or numeric.
<i>string</i>	The string in which substring substitutions are made. Any expression that resolves to a valid variable. <i>string</i> may be a dynamic array .

Description

The **SWAP** statement edits the value of *string* by replacing all instances of *subout* with *subin*. The *subout* and *subin* values may be of different lengths. Matching of strings is case-sensitive.

The value of *subout* and *subin* can be a string or a numeric. If numeric, the value is converted to canonical form (plus sign, leading and trailing zeros removed) before performing the **SWAP**.

To remove all instances of *subout* from *string*, specify the null string ("") as the *subin* value. The null string ("") cannot be used as the *subout* value.

Note: Caché MVBasic supports the UniData **SWAP** statement for substring replacement. UniVerse implements a completely different **SWAP** statement for variable value exchange, which we do not support at this time. Caché MVBasic also supports the UniVerse **CHANGE** function for substring replacement.

Examples

The following example illustrates use of the **SWAP** statement, replacing a substring value in all the elements of a dynamic array:

```
cities="Pittsburg Penn.":@VM:"Philadelphia Penn."
SWAP "Penn." WITH "PA" IN cities
```

See Also

- [CHANGE](#) function

TRANSACTION ABORT

Reverts all changes made during the current transaction.

TRANSACTION ABORT

Description

The **TRANSACTION ABORT** statement reverts all changes made during the current transaction initiated by a **TRANSACTION START** statement. All file changes issued during the transaction are undone, returning the data to the state prior to the **TRANSACTION START**.

To commit the changes made during the current transaction, issue a **TRANSACTION COMMIT** statement, rather than a **TRANSACTION ABORT** statement.

Note: Caché MVBasic supports two sets of transaction statements:

- UniData-style **TRANSACTION START**, **TRANSACTION COMMIT**, and **TRANSACTION ABORT**.
- UniVerse-style **BEGIN TRANSACTION**, **COMMIT**, **ROLLBACK**, and **END TRANSACTION**.

These two sets of transaction statements should not be combined.

See Also

- [TRANSACTION START](#) statement
- [TRANSACTION COMMIT](#) statement

TRANSACTION COMMIT

Commits all changes made during the current transaction.

```
TRANSACTION COMMIT {THEN statements [END] | ELSE statements [END]}
```

Description

The **TRANSACTION COMMIT** statement ends the current transaction initiated by a **TRANSACTION START** statement. All file changes issued during the transaction are committed, and cannot be subsequently reverted.

You must specify either a **THEN** clause, an **ELSE** clause, or both a **THEN** and an **ELSE** clause. If the transaction commit is successful, the **THEN** clause is executed. If the transaction commit fails, the **ELSE** clause is executed.

To revert the changes made during the current transaction, issue a **TRANSACTION ABORT** statement, rather than a **TRANSACTION COMMIT** statement.

Note: Caché MVBasic supports two sets of transaction statements:

- UniData-style **TRANSACTION START**, **TRANSACTION COMMIT**, and **TRANSACTION ABORT**.
- UniVerse-style **BEGIN TRANSACTION**, **COMMIT**, **ROLLBACK**, and **END TRANSACTION**.

These two sets of transaction statements should not be combined.

See Also

- [TRANSACTION START](#) statement
- [TRANSACTION ABORT](#) statement

TRANSACTION START

Begins a transaction.

```
TRANSACTION START {THEN statements [END] | ELSE statements [END]}
```

Description

The **TRANSACTION START** statement initiates a transaction. All subsequent statements are part of this transaction until the transaction is closed, either by a **TRANSACTION COMMIT** statement or a **TRANSACTION ABORT** statement.

You must specify either a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If the transaction start is successful, the THEN clause is executed. If the transaction start fails, the ELSE clause is executed.

Note: Caché MVBasic supports two sets of transaction statements:

- UniData-style **TRANSACTION START**, **TRANSACTION COMMIT**, and **TRANSACTION ABORT**.
- UniVerse-style **BEGIN TRANSACTION**, **COMMIT**, **ROLLBACK**, and **END TRANSACTION**.

These two sets of transaction statements should not be combined.

See Also

- [TRANSACTION COMMIT](#) statement
- [TRANSACTION ABORT](#) statement

UNLOCK

Releases a process lock.

```
UNLOCK expression
```

Arguments

<i>expression</i>	A number or string, or an expression that evaluates to a number or string specifying an existing lock to be unlocked.
-------------------	---

Description

The **UNLOCK** statement releases a process lock on *expression* that was obtained by a **LOCK** statement. Each time a lock is obtained on an *expression* a lock count is incremented. **UNLOCK** decrements this count. Only when the lock count falls to zero will the logical lock be released. For this reason, you should balance each successful call to **LOCK** with a corresponding call to **UNLOCK**.

Unlike **READU** locks, process locks set in a program are not released automatically when the program terminates. The lock belongs to the process, and persists for the life of the process, unless unlocked explicitly.

Commonly, *expression* evaluates to an integer in the range 0 through 64. However, in Caché any number or string may be specified as a logical lock.

Examples

The following example uses the **LOCK** statement to obtain a logical lock on an expression, and then uses the **UNLOCK** function to release the logical lock. Note that because the lock on *a* was taken twice, it must be unlocked twice.

```

a=45
LOCK a THEN PRINT "Got the lock"
  ELSE PRINT "Couldn't get the lock"
LOCK a THEN PRINT "Got the lock again"
  ELSE PRINT "Couldn't get the lock"
.
.
.
UNLOCK a
UNLOCK a

```

See Also

- [LOCK](#) statement

WEOFSEQ

Writes an end-of-file to a sequential file.

```
WEOFSEQ filevar [ON ERROR statements]
```

Arguments

<i>filevar</i>	A file variable name used to refer to the file in Caché MVBasic. This <i>filevar</i> is obtained from OPENSEQ .
----------------	--

Description

The **WEOFSEQ** statement is used to write an end-of-file indicator to a file that has been opened for sequential access using **OPENSEQ**. Placing an end-of-file indicator renders all data past that point inaccessible to **READSEQ** statements. Placing an end-of-file indicator has no effect on **WRITESEQ** statements, or on the pointer position count provided by the **STATUS** statement.

You can optionally specify an ON ERROR clause, which is taken if the end-of-file write fails. You can also use the **STATUS** function to determine the status of the write operation, as follows: 0=success; -1=operation failed because file not open (or opened by another process).

See Also

- [OPENSEQ](#) statement
- [READSEQ](#) statement
- [WRITESEQ](#) statement

- [STATUS](#) statement
- [STATUS](#) function

WRITE, WRITEU, WRITEV, WRITEVU

Writes data to a record in a MultiValue file.

```
WRITE data TO filevar,recID [THEN statements][ELSE statements]

WRITEU data TO filevar,recID [LOCKED statements] [THEN statements][ELSE
statements]

WRITEV data TO filevar,recID,fieldno [THEN statements][ELSE statements]

WRITEVU data TO filevar,recID,fieldno [LOCKED statements] [THEN
statements][ELSE statements]
```

Arguments

<i>data</i>	Data to write to the MultiValue file. Can be an expression or variable that resolves to a dynamic array or some other literal value.
<i>filevar</i>	A local variable used as the file identifier of an open MultiValue file. This variable is set by the OPEN statement.
<i>recID</i>	The record ID of the record to be written, specified as either a number or an alphanumeric string of up to 31 characters. Letters in a <i>recID</i> are case-sensitive. Do not specify as a quoted string.
<i>fieldno</i>	The field number of the field to write. Used with WRITEV and WRITEVU .
<i>LOCKED statements</i>	<i>Optional</i> — One or more MVBasic statements executed if WRITEU or WRITEVU could not perform a write due to lock contention.
<i>THEN statements</i>	<i>Optional</i> — One or more MVBasic statements executed if the write operation completed successfully.
<i>ELSE statements</i>	<i>Optional</i> — One or more MVBasic statements executed if the write operation did not complete successfully.

Description

The **WRITE** statements are used to write data to a record in a MultiValue file. You supply this data using the *data* variable.

- **WRITE** writes a record
- **WRITEU** writes a record, retaining an update record lock
- **WRITEV** writes a field within a record
- **WRITEU** writes a field within a record, retaining an update record lock

You can optionally specify a **THEN** clause, an **ELSE** clause, or both a **THEN** and an **ELSE** clause. If the file write is successful, the **THEN** clause is executed. If file write fails, the **ELSE** clause is executed.

You can use the **STATUS** function to determine the status of the write operation, as follows: 0=write successful; -1=write failed because file not open (or opened by another process).

Record Naming Conventions

- A *recID* can be a number or an alphanumeric string.
- If a number, it is converted to canonical form: multiple plus and minus signs are resolved, and the plus sign, and leading and trailing zeros are removed. If the number is enclosed in single or double quotation marks, conversion to canonical form is not performed. Only a single period can be specified, which is used as the decimal separator character.
- If an alphanumeric string, the first character must be a letter, dollar sign (\$), or percent sign (%). Subsequent characters may be letters, numbers, or percent characters. If the first character is a dollar sign (\$), all subsequent characters must be letters.
- The period (.) character can appear within a *recID*. If the *recID* is alphabetic any number of periods can be specified; these periods are stripped out and are not part of the *recID*. If the *recID* is a mixed alphanumeric, no periods may be specified.
- The *recID* may be enclosed in single or double quotation marks, these become part of the record name, unless the *recID* is an integer in canonical form. Single and double quotes are equivalent. Thus: "4"='4'=4 and "rec1"='rec1' but not equal to rec1. Do not specify a blank space within a *recID*.
- A *recID* is case-sensitive.
- A *recID* is limited to 31 characters. You may specify a *recID* longer than 31 characters, but only the first 31 characters are used. Therefore, a *recID* must be unique within its first 31 characters.

Examples

The following example writes a line of data to an existing sequential file on a Windows system:

```
OPEN "TEST.FILE" TO mytest
  IF STATUS()=0
  THEN
    WRITE "John Doe" TO mytest,1
    CLOSE mytest
  END
ELSE
  PRINT "File open failed"
END
```

See Also

- [OPEN](#) statement
- [READ](#) statement
- [CLOSE](#) statement
- [STATUS](#) statement
- [Dynamic Arrays](#)

WRITEBLK

Writes data to a sequential file.

```
WRITEBLK data ON filevar [THEN statements][ELSE statements]
WRITEBLK data TO filevar [THEN statements][ELSE statements]
```

Arguments

<i>data</i>	Data to write to the sequential file. Can be an expression or variable that resolves to a literal value.
<i>filevar</i>	A file variable name used to refer to the file in Caché MVBASIC. This <i>filevar</i> is obtained from OPENSEQ . The ON and TO keywords are equivalent.

Description

The **WRITEBLK** statement is used to write data to a file that has been opened for sequential access using **OPENSEQ**. You supply this data using the *data* variable. The *data* is written as a variable-length “block” (meaning that the data receives no special processing and no

special characters are appended). The length of the block is determined by the length of the specified data; the *data* can be of any length. It has no necessary relationship to logical data units, such as lines or records.

When invoked, **WRITEBLK** increments a pointer to the end of the data just written, so that repeated invocations of **WRITEBLK** write sequential blocks of data to the file. The same file pointer is used by **WRITEBLK** and **READBLK**.

You can determine the current position of this pointer using the **STATUS** statement. You can reposition this pointer using the **SEEK** statement.

To write an end-of-file, use the **WEOFSEQ** statement.

You can optionally specify a **THEN** clause, an **ELSE** clause, or both a **THEN** and an **ELSE** clause. If the file write is successful, the **THEN** clause is executed. If file write fails, the **ELSE** clause is executed.

You can use the **STATUS** function to determine the status of the write operation, as follows: 0=sequential write successful; -1=write failed because file not open (or opened by another process).

WRITEBLK and WRITESEQ

The **WRITEBLK** command writes a string of data to a sequential file. This string may have no relationship to a record within the file. The **WRITESEQ** command writes a single line of data (a data record) to a sequential file, ending the write by appending two newline characters (carriage return & linefeed) to the data.

Issuing a **WRITESEQ** creates a new file, if the file specified in **OPENSEQ** does not exist. Issuing a **WRITEBLK** does not create a new file. You must issue a **CREATE** statement to create a sequential file before invoking **WRITEBLK**.

Examples

The following example writes a block of data to an existing sequential file on a Windows system:

```
OPENSEQ "C:\myfiles\test1" TO mytest
  IF STATUS()=0
  THEN
    WRITEBLK "John Doe" TO mytest
    WEOFSEQ mytest
    CLOSESEQ mytest
  END
  ELSE
    PRINT "File open failed"
  END
```

The following example creates a new sequential file and writes a block of data to it. The **CREATE** statement is mandatory with **WRITEBLK**:

```
OPENSEQ "C:\myfiles\test1" TO mytest
CREATE mytest
WRITEBLK "John Doe" TO mytest
WEOFSEQ mytest
CLOSESEQ mytest
```

See Also

- [OPENSEQ](#) statement
- [CREATE](#) statement
- [READBLK](#) statement
- [WRITESEQ](#) statement
- [WEOFSEQ](#) statement
- [CLOSESEQ](#) statement
- [SEEK](#) statement
- [STATUS](#) statement
- [STATUS](#) function

WRITESEQ

Writes a line of data to a sequential file.

```
WRITESEQ data ON filevar [ON ERROR statements]
WRITESEQ data TO filevar [ON ERROR statements]
WRITESEQ data ON filevar [THEN statements][ELSE statements]
WRITESEQ data TO filevar [THEN statements][ELSE statements]
```

Arguments

<i>data</i>	Data to write to the sequential file. Can be an expression or variable that resolves to a literal value.
<i>filevar</i>	A file variable name used to refer to the file in Caché MVBasic. This <i>filevar</i> is obtained from OPENSEQ . The ON and TO keywords are equivalent.

Description

The **WRITESEQ** statement is used to write a line of data to a file that has been opened for sequential access using **OPENSEQ**. You supply this data using the *data* variable. **WRITESEQ** appends the two newline characters (carriage return & linefeed) to the data, defining it as a line of data.

By default, **WRITESEQ** begins writing at the beginning of the file, overwriting any existing file data.

WRITESEQ increments a pointer to the end of the data it has just written (plus the two newline characters), so that repeated invocations of **WRITESEQ** write sequential lines of data to the file. The same file pointer is used by **WRITESEQ** and **READSEQ**.

You can determine the current position of this pointer using the **STATUS** statement. You can reposition this pointer using the **SEEK** statement.

To write an end-of-file, use the **WEOFSEQ** statement.

You can optionally specify a **THEN** clause, an **ELSE** clause, or both a **THEN** and an **ELSE** clause. If the file write is successful, the **THEN** clause is executed. If file write fails, the **ELSE** clause is executed.

You can optionally specify an **ON ERROR** clause. If file write fails, the **ON ERROR** clause is executed. This clause is equivalent to the **ELSE** clause.

You can use the **STATUS** function to determine the status of the write operation, as follows: 0=sequential write successful; -1=write failed because file not open (or opened by another process).

If you are creating a new file, issue an **OPENSEQ** and then issue a **WRITESEQ**. Issuing a **CREATE** is optional; the first **WRITESEQ** creates the file.

WRITESEQ and **WRITEBLK**

The **WRITEBLK** command writes a string of data to a sequential file. This string can be of any length, and may have no relationship to a record within the file. The **WRITESEQ** command writes a single line of data (a data record) to a sequential file, ending the write by appending two newline characters (carriage return & linefeed) to the data.

Issuing a **WRITESEQ** creates a new file, if the file specified in **OPENSEQ** does not exist. Issuing a **WRITEBLK** does not create a new file.

Examples

The following example writes a line of data to an existing sequential file on a Windows system:

```
OPENSEQ "C:\myfiles\test1" TO mytest
  IF STATUS()=0
  THEN
    WRITESEQ "John Doe" TO mytest
    WEOFSEQ mytest
    CLOSESEQ mytest
  END
ELSE
  PRINT "File open failed"
END
```

See Also

- [OPENSEQ](#) statement
- [READSEQ](#) statement
- [WRITEBLK](#) statement
- [WEOFSEQ](#) statement
- [CLOSESEQ](#) statement
- [SEEK](#) statement
- [STATUS](#) statement
- [STATUS](#) function

Caché MultiValue Basic Functions

ABS

Returns the absolute value of a number.

```
ABS ( number )
```

Arguments

<i>number</i>	Any valid numeric expression, specified as a number or a numeric string.
---------------	--

Description

The absolute value of a number is its unsigned magnitude. For example, **ABS**(-1) and **ABS**(1) both return 1. **ABS** removes plus and minus signs and leading and trailing zeros from *number*. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7. If *number* is an uninitialized variable or a non-numeric value, **ABS** returns 0 (zero).

The **ABS** function gives the absolute value of a number: all numbers become positive. The **NEG** function inverts the sign of a number: negative numbers become positive and positive numbers become negative.

Examples

The following example uses the **ABS** function to compute the absolute value of a number:

```
PRINT ABS(0050.300);    ! Returns 50.3
PRINT ABS(-50.3);      ! Returns 50.3
PRINT ABS(+50.3);      ! Returns 50.3
PRINT ABS(0);          ! Returns 0
PRINT ABS(-0);         ! Returns 0
```

See Also

- [ABSS](#) function
- [NEG](#) function

ABSS

Returns the absolute value of each element in a dynamic array.

```
ABSS(dynarray)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
-----------------	---

Description

The **ABSS** function returns the absolute value of each numeric element of *dynarray*. The absolute value of a number is its unsigned magnitude. **ABSS** removes signs, and leading and trailing zeros from elements in a dynamic array. If a *dynarray* element is an uninitialized variable, a null string, or a non-numeric value, **ABSS** returns a value of 0 (zero) for that element.

Examples

The following example uses the **ABSS** function to return the absolute value of each of the numbers in a dynamic array:

```
a=11:@VM:-22:@VM:-33:@VM:44
PRINT a;           ! returns 11v-22v-33v44
PRINT ABSS(a);    ! returns 11v22v33v44
```

See Also

- [ABS](#) function
- [NEGS](#) function
- [Dynamic Arrays](#)

ACOS

Returns the arc-cosine of an angle.

```
ACOS ( number )
```

Arguments

<i>number</i>	Any valid numeric expression in the range -1 to 1. Values outside of this range generate a syntax error.
---------------	--

Description

The **ACOS** function returns the trigonometric arc-cosine of *number*. An arc-cosine is the inverse of a cosine. The result is given in radians.

ACOS(-1) returns the value of pi. To convert degrees to radians, multiply degrees by pi/180. To convert radians to degrees, multiply radians by 180/pi.

Examples

The following example uses the **ACOS** function to return the arc-cosine of an angle:

```
PRINT ACOS(-0.5):" in radians"  
PRINT ACOS(-0.5)*(180/ACOS(-1)):" in degrees"
```

See Also

- [ATAN](#) function
- [COS](#) function
- [SIN](#) function
- [TAN](#) function
- Derived Math Functions
- ObjectScript: \$ZARCCOS function

ADDS

Adds the values of corresponding elements in two dynamic arrays.

```
ADDS(dynarray1, dynarray2)
```

Arguments

<i>dynarray</i>	Any valid dynamic array of numeric values.
-----------------	--

Description

The **ADDS** function adds the value of each element in *dynarray1* to the corresponding element in *dynarray2*. It then returns a dynamic array containing the results of these additions. If a *dynarray* element value is an uninitialized variable, a null string, or a non-numeric value, **ADDS** parses its value as 0 (zero).

You can use the **NUMS** function to determine if the elements in a dynamic array are numeric. You can use the **SUBS**, **MULS**, **DIVS**, and **MODS** functions to perform other arithmetic operations on the corresponding elements of two dynamic arrays.

To add together the element values within a single dynamic array, use either the **SUM** function (for single-level dynamic arrays) or the **SUMMATION** function (for multi-level dynamic arrays),

Examples

The following example uses the **ADDS** function to add the elements of two dynamic arrays:

```
a=11:@VM:22:@VM:33:@VM:44
b=10:@VM:9:@VM:8:@VM:7
PRINT a;           ! returns 11v22v33v44
PRINT ADDS(a,b);  ! returns 21v31v41v51
```

See Also

- [CATS](#) function
- [DIVS](#) function
- [MODS](#) function
- [MULS](#) function
- [SUM](#) function
- [SUMMATION](#) function

- [SUBS](#) function
- [Dynamic Arrays](#)

ALPHA

Determines if a string is alphabetic or not.

```
ALPHA(string)
```

Arguments

<i>string</i>	Any valid string.
---------------	-------------------

Description

If *string* consists entirely of alphabetic characters, **ALPHA** returns 1. Otherwise, **ALPHA** returns 0. Note that blank spaces are non-alphabetic characters.

Examples

The following example uses the **ALPHA** function to determine if a string consists of only alphabetic characters:

```
PRINT ALPHA("abcdefg");      ! Returns 1
PRINT ALPHA("AbCdeFG");     ! Returns 1
PRINT ALPHA("my string");   ! Returns 0 (space not allowed)
PRINT ALPHA("half-wit");    ! Returns 0 (hyphen not allowed)
PRINT ALPHA("");           ! Returns 0
PRINT ALPHA(123);          ! Returns 0
```

See Also

- [NUM](#) function

ANDS

Returns the logical AND of corresponding elements of two dynamic arrays.

```
ANDS(dynarray1, dynarray2)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
-----------------	---

Description

The **ANDS** function performs a logical AND test on the corresponding element values of *dynarray1* and *dynarray2*. If both element values are non-zero numeric values, **ANDS** returns 1 for that element. Otherwise, **ANDS** returns 0. If a *dynarray* element value is an uninitialized variable, a null string, or a string containing any non-numeric value, **ANDS** parses its value as 0.

Caché MVBasic also supports the logical AND operators & and AND. These can be applied to the elements of dynamic arrays by using the **REUSE** function.

Examples

The following example uses the **ANDS** function to compare two dynamic arrays. It returns 1 when both element values are non-zero:

```
a=1:@VM:0:@VM:33:@VM:0
b=10:@VM:9:@VM:1:@VM:0
PRINT ANDS(a,b)
! returns 1v0v1v0
```

See Also

- [ORS](#) function
- [NOTS](#) function
- [Dynamic Arrays](#)
- [Operators](#)

ASCII

Converts a string from EBCDIC to ASCII.

```
ASCII(string)
```

Arguments

<i>string</i>	A string of characters in EBCDIC representation.
---------------	--

Description

The **ASCII** function takes one or more EBCDIC codes and returns the corresponding ASCII character(s). This is the inverse of the **EBCDIC** function.

The **CHAR** function takes an ASCII code and returns the corresponding character. The **SEQ** function takes a character and returns the corresponding ASCII code.

Examples

The following example uses the **ASCII** function to return the characters associated with the specified EBCDIC code string:

```
estring=EBCDIC("ABCDEFGG")
astring=ASCII(estring)
PRINT astring
! returns "ABCDEFGG"
```

The following example shows the use of the **SEQ** and **CHAR** functions with the **ASCII** function:

```
PRINT SEQ(EBCDIC("A"))
! returns 193
PRINT ASCII(CHAR(193))
! returns "A"
```

See Also

- [EBCDIC](#) function
- [CHAR](#) function
- [SEQ](#) function
- [Strings](#)

ASIN

Returns the arc-sine of an angle.

```
ASIN(number)
```

Arguments

<i>number</i>	Any valid numeric expression in the range -1 to 1. Values outside of this range generate a syntax error.
---------------	--

Description

The **ASIN** function returns the trigonometric arc-sine of *number*. An arc-sine is the inverse of a sine. The result is given in radians.

To convert degrees to radians, multiply degrees by $\pi/180$. To convert radians to degrees, multiply radians by $180/\pi$.

Examples

The following example uses the **ASIN** function to return the arc-sine of an angle:

```
PRINT ASIN(-0.5):" in radians"  
PRINT ASIN(-0.5)*(180/ACOS(-1)):" in degrees"
```

See Also

- [ATAN](#) function
- [COS](#) function
- [SIN](#) function
- [TAN](#) function
- Derived Math Functions
- ObjectScript: \$ZARCSIN function

ASSIGNED

Determines if a variable is assigned.

```
ASSIGNED(var)
```

Arguments

<i>var</i>	Any valid variable name . If <i>var</i> is not a valid variable name, MVBasic issues a syntax error.
------------	--

Description

The **ASSIGNED** function determines whether a variable is assigned or not assigned. If *var* is assigned a value, **ASSIGNED** returns 1. If *var* is not assigned a value, **ASSIGNED** returns 0. An assigned value can be a single value or a dynamic array value. **ASSIGNED** also returns 1 if *var* is assigned the null string or is assigned an unassigned variable.

The **UNASSIGNED** function is the functional opposite of the **ASSIGNED** function.

Examples

The following example tests the assignment of several variables. **ASSIGNED** returns 1 (assigned) for variables *a* through *f*. **ASSIGNED** returns 0 (unassigned) for variable *g*.

```
a=123
b="fred"
c=1:@VM:2:@VM:3
d=""
e=NULL
f=g
PRINT ASSIGNED(a)
PRINT ASSIGNED(b)
PRINT ASSIGNED(c)
PRINT ASSIGNED(d)
PRINT ASSIGNED(e)
PRINT ASSIGNED(f)
PRINT ASSIGNED(g)
```

Note that variable *f* is considered assigned, even though it is assigned to an unassigned variable.

See Also

- [UNASSIGNED](#) function

ATAN

Returns the arctangent of a number.

```
ATAN(number)
```

Arguments

<i>number</i>	Any valid numeric expression, specified as a number or a numeric string.
---------------	--

Description

The **ATAN** function takes the ratio of two sides of a right triangle (*number*) and returns the corresponding angle in radians. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle. The range of the result is $-\pi/2$ to $\pi/2$ radians.

To convert degrees to radians, multiply degrees by $\pi/180$. To convert radians to degrees, multiply radians by $180/\pi$.

Examples

The following example returns the arctangents of the integers from -4 through 4:

```
FOR x = -4 TO 4
PRINT "Arctangent of ":x:" is ":ATAN(x)
NEXT
```

The following example uses **ATAN** to calculate the value of pi:

```
PRINT ATAN(1)*4;    ! Calculate the value of pi.
```

Notes

Arctangent (**ATAN**) is the inverse trigonometric function of tangent (**TAN**), which takes an angle as its argument and returns the ratio of two sides of a right triangle. Do not confuse the arctangent with the cotangent; a cotangent is the simple inverse of a tangent ($1/\text{tangent}$).

See Also

- [COS](#) function
- [SIN](#) function
- [TAN](#) function
- [Derived Math Functions](#)

- ObjectScript: \$ZARCTAN function

BITAND

Returns the bitwise AND for two bit strings.

```
BITAND(bitstring1,bitstring2)
```

Arguments

<i>bitstring</i>	An integer that specifies a bit string. For example, the integer 64 specifies the bitstring 1000000.
------------------	--

Description

The **BITAND** function compares two bit strings bit-by-bit, and returns a bitstring that is the logical AND bitwise comparison of the two strings. Both *bitstring* values are specified as positive integers. The returned value is also expressed as a positive integer.

The following is the truth table for **BITAND**:

	<i>bitstring1</i> = 0	<i>bitstring1</i> = 1
<i>bitstring2</i> = 0	0	0
<i>bitstring2</i> = 1	0	1

A *bitstring* can be expressed as either a number or as a string. A number are converted to canonical form, with leading plus signs and leading and trailing zeros omitted. If either argument evaluates to the null string, a non-numeric string, or an undefined variable, it is assumed to have a value of 0. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7.

Examples

The following example specifies a *bitstring1* of 14 (binary 1110), and a *bitstring2* of 9 (binary 1001). Bitwise AND comparison results in the binary string 1000, the integer value of which is 8:

```
PRINT BITAND(14,9); ! Returns 8
```

The following example specifies a *bitstring1* of 14 (binary 1110), and a *bitstring2* of 6 (binary 110). Bitwise AND comparison results in the binary string 0110, the integer value of which is 6:

```
PRINT BITAND(14,6); ! Returns 6
```

The following example specifies a *bitstring1* of 65 (binary 1000001), and a *bitstring2* of 62 (binary 111110). Bitwise AND comparison results in the binary string 0000000, the integer value of which is 0:

```
PRINT BITAND(65,62); ! Returns 0
```

The following example specifies two bitstrings with the same integer value. Bitwise AND comparison of a number with itself always results in the number:

```
PRINT BITAND(64,64); ! Returns 64
```

See Also

- [BITOR](#) function
- [BITXOR](#) function
- [BITNOT](#) function
- [BITSET](#) function
- [BITRESET](#) function
- [BITTEST](#) function

BITNOT

Sets the specified bit in a bitstring to its opposite value.

```
BITNOT(bitstring,bitno)
```

Arguments

<i>bitstring</i>	An integer that specifies a bit string. For example, the integer 64 specifies the bitstring 1000000.
<i>bitno</i>	An integer that specifies the bit position in <i>bitstring</i> to set to its opposite value. Bit positions are counted right to left, beginning with position 0.

Description

The **BITNOT** function defines a bit string using *bitstring* and changes (flips) one bit of that bit string at the location specified by *bitno*. Both values are specified as positive integers. If the bit specified by *bitno* has a value of 0, **BITNOT** sets it to 1. If the bit specified by *bitno* has a value of 1, **BITNOT** sets it to 0.

Both *bitstring* and *bitno* can be expressed as either numbers or as strings. These numbers are converted to canonical form, with leading plus signs and leading and trailing zeros omitted. If either argument evaluates to the null string, a non-numeric string, or an undefined variable, it is assumed to have a value of 0. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7.

The **BITNOT** function always changes the specified bit. The **BITSET** function only sets the specified bit if its value is 0. The **BITRESET** function only sets the specified bit if its value is 1.

Examples

The following example specifies a *bitstring* of 64 (binary 1000000), and *bitno* sets bit position 0 to its opposite. This results in the binary string 1000001, the integer value of which is 65:

```
PRINT BITNOT(64,0); ! Returns 65
```

The following example specifies a *bitstring* of 64 (binary 1000000), and *bitno* sets bit position 4 to its opposite. This results in the binary string 1010000, the integer value of which is 80:

```
PRINT BITNOT(64,4); ! Returns 80
```

The following example specifies a *bitstring* of 65 (binary 1000001), and *bitno* specifies setting bit position 0 to its opposite. This results in the binary string 1000000, the integer value of which is 64:

```
PRINT BITNOT(65,0); ! Returns 64
```

The following example specifies a *bitstring* of 8 (binary 1000), and *bitno* specifies setting bit position 4 to its opposite. The *bitstring* has an implicit bit position of 4 with a value of 0. Setting this bit to 1 returns the binary string 11000, the integer value of which is 24:

```
PRINT BITNOT(8,4); ! Returns 24
```

The following example specifies a *bitstring* of 1 (binary 1), and *bitno* sets bit position 0 to its opposite. This results in the binary string 0, the integer value of which is 0:

```
PRINT BITNOT(1,0); ! Returns 0
```

See Also

- [BITSET](#) function
- [BITRESET](#) function
- [BITTEST](#) function

BITOR

Returns the bitwise OR for two bit strings.

```
BITOR(bitstring1,bitstring2)
```

Arguments

<i>bitstring</i>	An integer that specifies a bit string. For example, the integer 64 specifies the bitstring 1000000.
------------------	--

Description

The **BITOR** function compares two bit strings bit-by-bit, and returns a bitstring that is the logical OR bitwise comparison of the two strings. Both *bitstring* values are specified as positive integers. The returned value is also expressed as a positive integer.

The following is the truth table for **BITOR**:

	<i>bitstring1</i> = 0	<i>bitstring1</i> = 1
<i>bitstring2</i> = 0	0	1
<i>bitstring2</i> = 1	1	1

A *bitstring* can be expressed as either a number or as a string. A number are converted to canonical form, with leading plus signs and leading and trailing zeros omitted. If either argument evaluates to the null string, a non-numeric string, or an undefined variable, it is assumed to have a value of 0. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7.

Examples

The following example specifies a *bitstring1* of 14 (binary 1110), and a *bitstring2* of 9 (binary 1001). Bitwise OR comparison results in the binary string 1111, the integer value of which is 15:

```
PRINT BITOR(14,9); ! Returns 15
```

The following example specifies a *bitstring1* of 14 (binary 1110), and a *bitstring2* of 6 (binary 110). Bitwise OR comparison results in the binary string 1110, the integer value of which is 14:

```
PRINT BITOR(14,6); ! Returns 14
```

The following example specifies a *bitstring1* of 65 (binary 1000001), and a *bitstring2* of 62 (binary 111110). Bitwise OR comparison results in the binary string 111111, the integer value of which is 127:

```
PRINT BITOR(65,62); ! Returns 127
```

The following example specifies two bitstrings with the same integer value. Bitwise OR comparison of a number with itself always results in the number:

```
PRINT BITOR(64,64); ! Returns 64
```

See Also

- [BITAND](#) function
- [BITXOR](#) function
- [BITNOT](#) function
- [BITSET](#) function
- [BITRESET](#) function
- [BITTEST](#) function

BITRESET

Sets the specified bit in a bitstring to 0.

```
BITRESET(bitstring,bitno)
```

Arguments

<i>bitstring</i>	An integer that specifies a bit string. For example, the integer 64 specifies the bitstring 1000000.
<i>bitno</i>	An integer that specifies the bit position in <i>bitstring</i> to set to 0. Bit positions are counted right to left, beginning with position 0.

Description

The **BITRESET** function defines a bit string using *bitstring* and resets to 0 one bit of that bit string at the location specified by *bitno*. Both values are specified as positive integers. If the bit specified by *bitno* has a value of 1, **BITRESET** sets it to 0. If the bit specified by *bitno* already has a value of 0, **BITRESET** leaves it unchanged.

Both *bitstring* and *bitno* can be expressed as either numbers or as strings. These numbers are converted to canonical form, with leading plus signs and leading and trailing zeros omitted. If either argument evaluates to the null string, a non-numeric string, or an undefined variable, it is assumed to have a value of 0. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7.

The **BITRESET** function sets a specified bit to 0. The **BITSET** function sets a specified bit to 1. The **BITNOT** function sets a specified bit to its opposite value.

Examples

The following example specifies a *bitstring* of 65 (binary 1000001), and *bitno* resets bit position 0 to the bit value 0. This results in the binary string 1000000, the integer value of which is 64:

```
PRINT BITRESET(65,0); ! Returns 64
```

The following example specifies a *bitstring* of 64 (binary 1000000), and *bitno* resets bit position 6 to the bit value 0. This results in the binary string 0000000, the integer value of which is 0:

```
PRINT BITRESET(64,6); ! Returns 0
```

The following example specifies a *bitstring* of 64 (binary 1000000), and *bitno* specifies resetting bit position 0 to the bit value 0. But because bit position 0 already has a bit value of 0, the binary string 1000000 (integer value 64) is returned unchanged:

```
PRINT BITRESET(64,0); ! Returns 64
```

The following example specifies a *bitstring* of 8 (binary 1000), and *bitno* specifies resetting bit position 4 to the bit value 0. The *bitstring* has an implicit bit position of 4, which already has a value of 0. Thus the original binary string 1000 (integer value 8) is returned unchanged:

```
PRINT BITRESET(8,4); ! Returns 8
```

The following example specifies a *bitstring* of 0 (binary 0), and *bitno* sets bit position 0 to the bit value 0. This results in the binary string 0, the integer value of which is 0:

```
PRINT BITRESET(0,0); ! Returns 0
```

See Also

- [BITSET](#) function
- [BITNOT](#) function
- [BITTEST](#) function

BITSET

Sets the specified bit in a bitstring to 1.

```
BITSET(bitstring,bitno)
```

Arguments

<i>bitstring</i>	An integer that specifies a bit string. For example, the integer 64 specifies the bitstring 1000000.
<i>bitno</i>	An integer that specifies the bit position in <i>bitstring</i> to set to 1. Bit positions are counted right to left, beginning with position 0.

Description

The **BITSET** function defines a bit string using *bitstring* and sets one bit of that bit string at the location specified by *bitno*. Both values are specified as positive integers. If the bit spec-

ified by *bitno* has a value of 0, **BITSET** sets it to 1. If the bit specified by *bitno* already has a value of 1, **BITSET** leaves it unchanged.

Both *bitstring* and *bitno* can be expressed as either numbers or as strings. These numbers are converted to canonical form, with leading plus signs and leading and trailing zeros omitted. Decimal values are truncated to integers. If either argument evaluates to the null string, a non-numeric string, or an undefined variable, it is assumed to have a value of 0. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7.

The **BITSET** function sets a specified bit to 1. The **BITRESET** function sets a specified bit to 0. The **BITNOT** function sets a specified bit to its opposite value.

Examples

The following example specifies a *bitstring* of 64 (binary 1000000), and *bitno* sets bit position 0 to the bit value 1. This results in the binary string 1000001, the integer value of which is 65:

```
PRINT BITSET(64,0); ! Returns 65
```

The following example specifies a *bitstring* of 64 (binary 1000000), and *bitno* sets bit position 4 to the bit value 1. This results in the binary string 1010000, the integer value of which is 80:

```
PRINT BITSET(64,4); ! Returns 80
```

The following example specifies a *bitstring* of 65 (binary 1000001), and *bitno* specifies setting bit position 0 to the bit value 1. But because bit position 0 already has a bit value of 1, the binary string 1000001 (integer value 65) is returned unchanged:

```
PRINT BITSET(65,0); ! Returns 65
```

The following example specifies a *bitstring* of 8 (binary 1000), and *bitno* specifies setting bit position 4 to the bit value 1. The *bitstring* has an implicit bit position of 4 with a value of 0. Setting this bit to 1 returns the binary string 11000, the integer value of which is 24:

```
PRINT BITSET(8,4); ! Returns 24
```

The following example specifies a *bitstring* of 0 (binary 0), and *bitno* sets bit position 0 to the bit value 1. This results in the binary string 1, the integer value of which is 1:

```
PRINT BITSET(0,0); ! Returns 1
```

The following examples specify *bitstring* and *bitno* with null string values. The null string is assigned a value of 0:

```
PRINT BITSET("",1); ! Returns 2
PRINT BITSET(1,""); ! Returns 1
```

See Also

- [BITRESET](#) function
- [BITNOT](#) function
- [BITTEST](#) function

BITTEST

Tests the value of the specified bit in a bitstring.

```
BITTEST(bitstring,bitno)
```

Arguments

<i>bitstring</i>	An integer that specifies a bit string. For example, the integer 64 specifies the bitstring 1000000.
<i>bitno</i>	An integer that specifies the bit position in <i>bitstring</i> to test. Bit positions are counted right to left, beginning with position 0.

Description

The **BITTEST** function defines a bit string using *bitstring* and tests the value of one bit of that bit string at the location specified by *bitno*. If the bit specified by *bitno* has a value of 0, **BITTEST** returns 0. If the bit specified by *bitno* has a value of 1, **BITTEST** returns 1.

Both *bitstring* and *bitno* are specified as positive integers. These arguments can be expressed as either numbers or as strings. Numbers are converted to canonical form, with leading plus signs and leading and trailing zeros omitted. If either argument evaluates to the null string, a non-numeric string, or an undefined variable, it is assumed to have a value of 0. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7.

You can use the **BITSET** function to set individual bits.

Examples

The following examples specify a *bitstring* of 14 (binary 1110), and use *bitno* to specify each bit in turn, returning the value of the bit:

```
x = BITSET(14,3);      ! Returns 14
PRINT BITTEST(x,0);   ! Returns 0
PRINT BITTEST(x,1);   ! Returns 1
PRINT BITTEST(x,2);   ! Returns 1
PRINT BITTEST(x,3);   ! Returns 1
```

The following example specifies a *bitstring* of 8 (binary 1000), and *bitno* specifies bit position 4. The *bitstring* has an implicit bit position of 4 with a value of 0.

```
PRINT BITTEST(8,4);   ! Returns 0
```

See Also

- [BITRESET](#) function
- [BITSET](#) function

BITXOR

Returns the bitwise XOR for two bit strings.

```
BITXOR(bitstring1,bitstring2)
```

Arguments

<i>bitstring</i>	An integer that specifies a bit string. For example, the integer 64 specifies the bitstring 1000000.
------------------	--

Description

The **BITXOR** function compares two bit strings bit-by-bit, and returns a bitstring that is the logical exclusive or (XOR) bitwise comparison of the two strings. Both *bitstring* values are specified as positive integers. The returned value is also expressed as a positive integer.

The following is the truth table for **BITXOR**:

	<i>bitstring1</i> = 0	<i>bitstring1</i> = 1
<i>bitstring2</i> = 0	0	1
<i>bitstring2</i> = 1	1	0

A *bitstring* can be expressed as either a number or as a string. A number are converted to canonical form, with leading plus signs and leading and trailing zeros omitted. If either argument evaluates to the null string, a non-numeric string, or an undefined variable, it is assumed to have a value of 0. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7.

Examples

The following example specifies a *bitstring1* of 14 (binary 1110), and a *bitstring2* of 9 (binary 1001). Bitwise XOR comparison results in the binary string 0111, the integer value of which is 7:

```
PRINT BITXOR(14,9); ! Returns 7
```

The following example specifies a *bitstring1* of 14 (binary 1110), and a *bitstring2* of 6 (binary 110). Bitwise XOR comparison results in the binary string 1000, the integer value of which is 8:

```
PRINT BITXOR(14,6); ! Returns 8
```

The following example specifies a *bitstring1* of 65 (binary 1000001), and a *bitstring2* of 62 (binary 111110). Bitwise XOR comparison results in the binary string 1111111, the integer value of which is 127:

```
PRINT BITXOR(65,62); ! Returns 127
```

The following example specifies two bitstrings with the same integer value. Bitwise XOR comparison of a number with itself always results in 0:

```
PRINT BITXOR(64,64); ! Returns 0
```

See Also

- [BITAND](#) function
- [BITOR](#) function
- [BITNOT](#) function
- [BITSET](#) function

- [BITRESET](#) function
- [BITTEST](#) function

CATS

Concatenates the values of corresponding elements in two dynamic arrays.

```
CATS(dynarray1, dynarray2)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
-----------------	---

Description

The **CATS** function concatenates the value of each element in *dynarray1* to the corresponding element in *dynarray2*. It then returns a dynamic array containing the results of these concatenations. If a dynamic array element contains an empty string or an uninitialized variable, no concatenation is performed for that element, and the element value from the other dynamic array is returned.

You can use the **SPLICE** function to concatenate the elements of two dynamic arrays, supplying a delimiter character. You can use the **REUSE** function to concatenate the same value to all of the elements of a dynamic array, or to concatenate a default value when the dynamic arrays differ in length.

You can use the **ADDS**, **SUBS**, **MULS**, **DIVS**, and **MODS** functions to perform arithmetic operations on the corresponding elements of two dynamic arrays.

Examples

The following example uses the **CATS** function to concatenate the elements of two dynamic arrays:

```
a=11:@VM:22:@VM:33:@VM:44
b=10:@VM:9:@VM:8:@VM:7
PRINT CATS(a,b)
! returns 1110v229v338v447
```

See Also

- [REUSE](#) function

- [SPLICE](#) function
- [Dynamic Arrays](#)

CHANGE

Replaces a substring in a string.

```
CHANGE(string,subout,subin[,occurrences[,begin]])
```

Arguments

<i>string</i>	The string in which substring substitutions are made. Any expression that resolves to a valid variable. <i>string</i> may be a dynamic array .
<i>subout</i>	The substring to be replaced. Any expression that resolves to a valid string or numeric.
<i>subin</i>	The substring to be inserted in place of <i>subout</i> . Any expression that resolves to a valid string or numeric.
<i>occurrences</i>	<i>Optional</i> — A positive integer specifying the number of occurrences of <i>subout</i> to replace with <i>subin</i> . If omitted, all occurrences are replaced. If used with <i>begin</i> , you can specify an <i>occurrences</i> value of -1 indicating that all occurrences of <i>subout</i> from the <i>begin</i> point to the end of the string are to be replaced.
<i>begin</i>	<i>Optional</i> — An integer specifying which occurrence of <i>subout</i> to begin replacement with. If omitted, or specified as 0 or 1, replacement begins with the first occurrence of <i>subout</i> .

Description

The **CHANGE** function edits the value of *string* by replacing some or all instances of *subout* with *subin*. The *subout* and *subin* values may be of different lengths. Matching of strings is case-sensitive.

The value of *subout* and *subin* can be a string or a numeric. If numeric, the value is converted to canonical form (plus sign, leading and trailing zeros removed) before performing the **CHANGE** operation.

To remove all instances of *subout* from *string*, specify the null string ("") as the *subin* value. The null string ("") cannot be used as the *subout* value.

The value of *occurrences* may be larger than the actual number of occurrences. If *occurrences* is omitted, or set to a value of 0, a negative number, the null string, a non-numeric string, or an undefined variable, all occurrences are replaced. If *occurrences* is set to a decimal number, it is truncated to an integer; if set to a [mixed numeric string](#), it resolves to the numeric portion of the string.

Note: Caché MVBasic supports both the UniVerse **CHANGE** function and the UniData **SWAP** statement, both of which perform substring replacement.

You can use the **CONVERT** function to perform character-for-character substitutions.

Examples

The following example illustrates use of the **CHANGE** function, replacing a substring value in all the elements of a dynamic array:

```
cities="Pittsburg Penn.":@VM:"Philadelphia Penn."  
CHANGE(cities,"Penn.,"PA")
```

The following example illustrates use of the **CHANGE** function, replacing the third and fourth occurrences of a substring value:

```
teststr=123test123test123test123test123test123test  
CHANGE(teststr,"test","RETRY",2,3)  
! Returns "123test123test123RETRY123RETRY123test123test"
```

See Also

- [SWAP](#) statement
- [CONVERT](#) function

CHAR

Returns the character corresponding to the specified character code.

```
CHAR(charcode)
```

Arguments

<i>charcode</i>	A decimal integer that identifies a character. For 8-bit characters, the value in <i>charcode</i> must evaluate to a positive integer in the range 0 to 255. For 16-bit characters, specify integers in the range 256 through 65534.
-----------------	--

Description

The **CHAR** function takes a character code and returns the corresponding character. The **SEQ** function takes a character and returns the corresponding ASCII code.

Numbers from 0 to 31 are the same as standard, nonprintable ASCII codes. For example, **CHAR**(10) returns a linefeed character.

Note: **CHAR** and **UNICHAR** are functionally identical. On Unicode systems both can be used to return 16-bit Unicode characters. On 8-bit systems, these functions return a null string for character codes beyond 255.

The Caché MVBasic **CHAR** function returns a single character. The corresponding Caché ObjectScript **\$CHAR** function can return a string of multiple characters by specifying a comma-separated list of ASCII codes. The Caché MVBasic **CHARS** function takes a dynamic array of ASCII codes and returns the corresponding single characters as a dynamic array.

Examples

The following example uses the **CHAR** function to return the character associated with the specified character code:

```
PRINT CHAR(65);      ! Returns A.
PRINT CHAR(97);      ! Returns a.
PRINT CHAR(37);      ! Returns %.
PRINT CHAR(62);      ! Returns >.
```

The following example uses the **CHAR** function to return the lowercase letter characters of the Russian alphabet on a Unicode version of Caché. On an 8-bit version of Caché it returns a null string for each letter:

```
letter=1072
FOR x=1 TO 32
  PRINT CHAR(letter)
  letter=letter+1
NEXT
```

See Also

- [UNICHAR](#) function
- [CHARS](#) function
- [SEQ](#) function
- ObjectScript: [\\$CHAR](#) function

CHARS

Returns the character corresponding to the specified character code for each element of a dynamic array.

```
CHARS(dynarray)
```

Arguments

<i>dynarray</i>	A valid dynamic array of decimal integers that identify characters . For 8-bit characters, these element values must evaluate to positive integers in the range 0 to 255. For 16-bit characters, these element values must evaluate to positive integers in the range 256 through 65534.
-----------------	--

Description

The **CHARS** function takes a dynamic array of character codes and returns the corresponding characters. It returns these values as a dynamic array. The **SEQS** function takes a dynamic array of characters and returns the corresponding character codes.

Numbers from 0 to 31 are the same as standard, nonprintable ASCII codes. For example, **CHARS**(10) returns a linefeed character.

Note: **CHARS** and **UNICHARS** are functionally identical. On Unicode systems both can be used to return 16-bit Unicode characters. On 8-bit systems, these functions return a null string for character codes greater than 255.

The Caché MVBasic **CHARS** function returns a dynamic array of characters. The corresponding Caché ObjectScript **\$CHAR** function returns a string of characters by specifying a comma-separated list of character codes.

Examples

The following example uses the **CHARS** function to return the characters associated with each specified character code:

```
a=65:@VM:66:@VM:67:@VM:68
PRINT CHARS(a); ! returns AvBvCvD
```

The following example uses the **CHARS** function to return the first four letters of the Greek alphabet. On a Unicode version of Caché it returns the Greek letters in a dynamic array; on an 8-bit version of Caché it returns a dynamic array with a null string for each letter:

```
b=945:@VM:946:@VM:947:@VM:948
PRINT CHARS(b)
```

See Also

- [UNICHARS](#) function
- [CHAR](#) function
- [SEQS](#) function
- [Dynamic Arrays](#)
- ObjectScript: [\\$CHAR](#) function

CHECKSUM

Returns a checksum number for a string.

```
CHECKSUM(string)
```

Arguments

<i>string</i>	Any valid string .
---------------	------------------------------------

Description

The **CHECKSUM** function generates a cyclic redundancy code (also called a checksum) corresponding to *string*. It returns this checksum as a positive integer. A checksum can be used to determine if data has been modified or if it was incompletely transmitted.

CHECKSUM returns the same checksum number for a numeric and the corresponding numeric string. However, numerics are converted to canonical form before checksum processing, whereas numeric strings are not converted to canonical form.

A checksum is calculated using multiplication. Therefore, if *string* is 0 a checksum of 0 is returned. If *string* is a null string or an undefined variable, these values are parsed as 0, and a checksum of 0 is returned.

Examples

The following examples all return the same checksum:

```
PRINT CHECKSUM(123.4)
PRINT CHECKSUM("123.4")
PRINT CHECKSUM(+00123.400)
```

The following examples *do not* return the same checksum:

```
PRINT CHECKSUM(123.400)
PRINT CHECKSUM("123.400")
```

See Also

- [Strings](#)

COL1

Returns the **FIELD** substring start position.

```
COL1 ( )
```

Arguments

The **COL1** function takes no arguments. The parentheses are mandatory.

Description

The **COL1** function returns the starting position for the most recently called **FIELD** function. **FIELD** extracts a substring from a string by specifying a delimiter character. The specified delimiter immediately precedes the extracted substring. **COL1** returns the string position (counting from 1) of this delimiter character.

If the **FIELD** *count* is 1, **COL1** returns 0. If the **FIELD** *count* is greater than the number of delimited substrings, **COL1** returns 0. If the **FIELD** *delimiter* is not located in *string*, **COL1** returns 0

The initial **COL1** value is 0. The **COL1** value is preserved until it is overwritten by the next **FIELD** function call.

COL1 returns a substring's start delimiter position. **COL2** returns a substring's end delimiter position.

Examples

The following example shows the use of the **COL1** function:

```
colors="Red^Green^Blue^Yellow^Orange^Black"
FOR x=1 TO 5
  PRINT FIELD(colors,"^",x)
  PRINT "Start delimiter position: ":COL1()
  ! Returns: 0, 4, 10, 15, 22
NEXT
```

See Also

- [FIELD](#) function
- [COL2](#) function

COL2

Returns the **FIELD** substring end position.

```
COL2 ( )
```

Arguments

The **COL2** function takes no arguments. The parentheses are mandatory.

Description

The **COL2** function returns the ending position for the most recently called **FIELD** function. **FIELD** extracts a substring from a string by specifying a delimiter character. This substring is limited by encountering the next delimiter character. **COL2** returns the string position (counting from 1) of this substring-ending delimiter character.

If the **FIELD** *delimiter* is not located in *string* and *count*=1, **COL2** returns the full length of *string*. If **FIELD** returns a null string, **COL2** returns 0.

The initial **COL2** value is 0. The **COL2** value is preserved until it is overwritten by the next **FIELD** function call.

COL2 returns a substring's end delimiter position. **COL1** returns a substring's start delimiter position.

Examples

The following example shows the use of the **COL2** function:

```
colors="Red^Green^Blue^Yellow^Orange^Black"
FOR x=1 TO 5
  PRINT FIELD(colors,"^",x)
  PRINT "End delimiter position: ":COL2()
  ! Returns: 4, 10, 15, 22, 29
NEXT
```

See Also

- [FIELD](#) function

- [COL1](#) function

CONVERT

Replaces single characters in a string.

```
CONVERT(charsout,charsin,string)
```

Arguments

<i>charsout</i>	One or more characters to be replaced. Any expression that resolves to a valid string or numeric.
<i>charsin</i>	The character or characters to be inserted in place of the corresponding characters in <i>charsout</i> . Any expression that resolves to a valid string or numeric.
<i>string</i>	The string in which character substitutions are made. Any expression that resolves to a valid string . <i>string</i> may be a dynamic array .

Description

The **CONVERT** function edits the value of *string* by replacing all instances of single characters in *charsout* with single characters from *charsin* and returns the resulting string. **CONVERT** performs a character-for-character substitution. Matching of characters is case-sensitive.

CONVERT can be used as follows:

- To remove all instances of a character from a string, specify the character to be removed in *charsout* and a null string in *charsin*. For example, to remove the # character from *mystring*: `CONVERT("#", "", mystring)`
- To replace all instances of a character in a string with another character, specify the character to be replaced in *charsout* and the replacement character in *charsin*. For example, to replace all instances of the # character with the * character in *mystring*: `CONVERT("#", "*", mystring)`
- To replace all instances of a list of single characters with corresponding other single characters, specify those characters to be replaced in *charsout* and the corresponding replacement characters in *charsin*. For example, to replace all instances in *mystring* of

the each lowercase letter a, b, c, and d with the corresponding uppercase letter:
`CONVERT("abcd", "ABCD", mystring)`

- To both replace some single characters and remove others, specify those characters to be replaced or removed in *charsout*. First specify those to be replaced, then those to be removed. Specify the corresponding replacement characters in *charsin*, and nothing for the characters to be removed. For example, to replace all instances of + with &, and to remove all instances of # in *mystring*: `CONVERT("+#", "&", mystring)`

The value of *charsout* and *charsin* can be a string or a numeric. If numeric, the value is converted to canonical form (plus sign, leading and trailing zeros removed) before performing the **CONVERT** operation.

If *charsout* contains more characters than *charsin*, the unpaired characters are deleted from the returned string. If *charsin* contains more characters than *charsout*, the unpaired characters are ignored and have no effect.

Note: **CONVERT** performs single character one-for-one substitution for all instances in a string. The **CHANGE** function performs substring replacement, and can specify how many instances to replace and where to begin replacement.

The **CONVERT** statement and the **CONVERT** function perform the same operation, with the following difference: the **CONVERT** statement changes the supplied string; the **CONVERT** function returns a new string with the specified changes and leaves the supplied string unchanged.

Examples

The following example illustrates use of the **CONVERT** function in converting a string to a dynamic array by replacing the # character with a Value Mark level delimiter character:

```
cities="New York#Chicago#Boston#Los Angeles"
dynacities=CONVERT("#",CHAR(253),cities)
PRINT cities
PRINT dynacities
```

See Also

- [CONVERT](#) statement
- [CHANGE](#) function
- [SWAP](#) statement
- [Strings](#)

COS

Returns the cosine of an angle.

```
COS(number)
```

Arguments

<i>number</i>	Any valid numeric expression that expresses an angle in degrees.
---------------	--

Description

The **COS** function takes an angle in degrees and returns the ratio of two sides of a right triangle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse. The result is in radians, in the range -1 to 1.

To convert degrees to radians, multiply degrees by $\pi/180$. To convert radians to degrees, multiply radians by $180/\pi$.

Examples

The following example uses the **COS** function to return the cosine of an angle:

```
Dim MyAngle
MyAngle = 1.3;           ! Define angle in degrees.
PRINT COS(MyAngle);     ! Returns cosine in radians.
```

The following example uses the **COS** function to return the secant of an angle:

```
Dim MyAngle, MySecant
MyAngle = 1.3;           ! Define angle in degrees.
MySecant = 1 / Cos(MyAngle); ! Calculate secant.
Print MySecant;         ! Secant in radians.
```

See Also

- [ATAN](#) function
- [COSH](#) function
- [SIN](#) function
- [TAN](#) function
- Derived Math Functions
- ObjectScript: [\\$ZCOS](#) function

COSH

Returns the hyperbolic cosine of an angle.

```
COSH(number)
```

Arguments

<i>number</i>	Any valid numeric expression that expresses an angle in degrees.
---------------	--

Description

The **COSH** function takes an angle in degrees and returns the ratio of two sides of a right triangle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse. The result is in radians, in the range -1 to 1.

To convert degrees to radians, multiply degrees by $\pi/180$. To convert radians to degrees, multiply radians by $180/\pi$.

Examples

The following example uses the **COSH** function to return the hyperbolic cosine of an angle:

```
Dim MyAngle
MyAngle = 1.3;           ! Define angle in degrees.
PRINT COSH(MyAngle);   ! Returns hyperbolic cosine in radians.
```

See Also

- [ATAN](#) function
- [COS](#) function
- [SIN](#) function
- [TAN](#) function
- [Derived Math Functions](#)

COUNT

Returns the number of instances of a substring in a string.

```
COUNT(string, substring)
```

Arguments

<i>string</i>	Any valid string expression .
<i>substring</i>	A string to match against <i>string</i> .

Description

The **COUNT** function returns the number of times a specified *substring* appears in *string*.

String matching is case-sensitive. Numbers are converted to canonical form, with leading and trailing zeroes and plus signs removed. Numeric strings are not converted to canonical form. If *string* is an empty string ("") or an undefined variable, **COUNT** returns a count of 0.

If *substring* is the null string, **COUNT** returns the length of *string*. If *string* is an empty string ("") or an undefined variable, **COUNT** returns a length of 0. This use of **COUNT** is functionally identical to the **LEN** function.

Examples

The following example uses the **COUNT** function to return the number of appearance of a substring in a string:

```
PRINT COUNT("InterSystems", "s");    ! Returns 2
PRINT COUNT("InterSystems", "S");    ! Returns 1
PRINT COUNT("InterSystems", "te");   ! Returns 2
PRINT COUNT(+0099.900, 0);           ! Returns 0
PRINT COUNT("0099.900", 0);          ! Returns 4
PRINT COUNT("InterSystems", "");     ! Returns 12
```

The following example shows that overlapping substrings are only counted once:

```
PRINT COUNT("AAAAA", "AA");          ! Returns 2
```

See Also

- [LEN](#) function
- [DCOUNT](#) function

- [COUNTS](#) function

COUNTS

Returns the number of instances of a substring in each element of a dynamic array.

```
COUNTS(dynarray, substring)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
<i>substring</i>	A string to match against each element in <i>dynarray</i> .

Description

The **COUNTS** function returns the number of times a specified *substring* appears in each element of *dynarray*. These values are returned as a dynamic array of integer counts. A *dynarray* element containing an empty string ("") or an undefined variable returns a count of 0.

String matching is case-sensitive. Numbers are converted to canonical form, with leading and trailing zeroes and plus signs removed. Numeric strings are not converted to canonical form.

If *substring* is the null string, **COUNTS** returns the length of each dynamic array element. A *dynarray* element containing an empty string ("") or an undefined variable returns a length of 0. This use of **COUNTS** is functionally identical to the **LENS** function.

Examples

The following example uses the **COUNTS** function to return the number of appearance of a substring in each element of a dynamic array:

```
citystate="Springfield IL":@VM:"Springfield MA":@VM:
"Somerville MA":@VM:"Somerville NJ":@VM:"Somerville ME"
PRINT COUNTS(citystate,"Somerville")
PRINT COUNTS(citystate,"Springfield")
PRINT COUNTS(citystate,"MA")
PRINT COUNTS(citystate,"VA")
```

See Also

- [COUNT](#) function

- [LENS](#) function
- [Dynamic Arrays](#)

DATE

Returns the current local system date in internal format.

```
DATE( )
```

Arguments

None. The parentheses are mandatory.

Description

The **DATE** function returns the current date in a format such as the following:

```
14122
```

This represents the elapsed number of days since December 31, 1967. **DATE** returns the current date at the moment when the function is executed.

Caché MV Basic also supplies `@DATE`, `@DAY`, `@MONTH`, `@YEAR`, and `@YEAR4` system variables. These values are set when the process is initialized, and are only updated when a program is initiated from the MV shell. For further details, see the [Variables](#) page of this manual.

Examples

The following example calls the **DATE** function to return the current date in internal format, then uses the **CONV** function to convert date from internal format to display format.

```
PRINT DATE( )  
PRINT CONV( DATE( ) , "D" )
```

See Also

- [TIMEDATE](#) function
- [CONV](#) function
- Caché ObjectScript: \$HOROLOG special variable
- SQL: NOW function

DCOUNT

Returns the number of delimited substrings in a string.

```
DCOUNT(string,delimiter)
```

Arguments

<i>string</i>	Any valid string expression .
<i>delimiter</i>	One or more characters used as a delimiter in <i>string</i> .

Description

The **DCOUNT** function returns the number of delimited substrings that appears in *string*.

String matching is case-sensitive. Numbers are converted to canonical form, with leading and trailing zeroes and plus signs removed. Numeric strings are not converted to canonical form.

If *delimiter* doesn't appear in *string*, **DCOUNT** returns 1. If *delimiter* is the null string, **DCOUNT** returns the length of *string* plus 1.

If *string* is an empty string ("") or an undefined variable, **DCOUNT** returns a count of 0.

Examples

The following example uses the **DCOUNT** function to return the number of Value Mark delimited substrings in a dynamic array:

```
colors="Red":@VM:"Green":@VM:"Blue":@VM:"Yellow"
PRINT DCOUNT(colors,CHAR(253)); ! Returns 4
```

See Also

- [LEN](#) function
- [COUNT](#) function

DELETE

Deletes an element from a dynamic array.

```
DELETE(dynarray, f[, v[, s]])
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
<i>f</i>	An integer specifying the Field level of the dynamic array on which to perform the deletion. Fields are counted from 1.
<i>v</i>	<i>Optional</i> — An integer specifying the Value level of the dynamic array on which to perform the deletion. Values are counted from 1 within a Field.
<i>s</i>	<i>Optional</i> — An integer specifying the Subvalue level of the dynamic array on which to perform the deletion. Subvalues are counted from 1 within a Value.

Description

The **DELETE** function returns a dynamic array with one element deleted. It deletes both the data and the dynamic array delimiter. Which element to delete is specified by the *f*, *v*, and *s* integers. For example, if *f*=2 and *v*=3, this means delete the third value from the second field. If *f*=2 and *v* is not specified, this means to delete the entire second field.

The **DELETE** function and the **DEL** statement perform the same operation, with the following difference: **DEL** changes the supplied dynamic array; **DELETE** creates a new dynamic array with the specified change and leaves the supplied dynamic array unchanged.

Examples

The following example uses the **DELETE** function to delete the second value from the first field of a dynamic array:

```
cities="New York":@VM:"London":@VM:
"Chicago":@VM:"Boston":@VM:"Los Angeles"
PRINT cities
! Returns: "New YorkvLondonvChicagovBostonvLos Angeles"
PRINT DELETE(cities,1,2)
! Returns: "New YorkvChicagovBostonvLos Angeles"
```

See Also

- [DEL](#) statement
- [COUNTS](#) function
- [EXTRACT](#) function
- [Dynamic Arrays](#)

DIV

Integer division of two values.

```
DIV(numstr1,numstr2)
```

Arguments

<i>numstr1</i>	The dividend. Any valid numeric or numeric string .
<i>numstr2</i>	The divisor. Any valid non-zero numeric or numeric string .

Description

The **DIV** function divides the value of *numstr1* by *numstr2*, and returns the integer product. It discards the fractional remainder. If a *numstr* value is an uninitialized variable, a null string, or a non-numeric value, **DIV** parses its value as 0 (zero).

Attempting to divide by zero issues a “Division by zero” error and returns a value of 0.

To perform exact division with a fractional product, use the division operator (/). To perform modulo division, use the **MOD** or **REM** function.

You can use the **DIVS** and **MODS** functions to perform division on the elements of a dynamic array.

Examples

The following examples use the **DIV** function to return the integer product of a division operation:

```
PRINT DIV(10,5);      ! returns 2
PRINT DIV(10,4);      ! returns 2
PRINT DIV(10,3.3);    ! returns 3
PRINT DIV(10,3.4);    ! returns 2
PRINT DIV(10.2,3.4);  ! returns 3
PRINT DIV(10,-3);     ! returns -3
PRINT DIV(-10,3);     ! returns -3
```

See Also

- [MOD](#) function
- [REM](#) function
- [DIVS](#) function
- [MODS](#) function
- [Operators](#)

DIVS

Divides the values of the corresponding elements in two dynamic arrays.

```
DIVS(dynarray1,dynarray2)
```

Arguments

<i>dynarray1</i>	The dividend. Any valid dynamic array of numeric values.
<i>dynarray2</i>	The divisor. Any valid dynamic array of non-zero numeric values.

Description

The **DIVS** function divides the value of each element in *dynarray1* by the corresponding element in *dynarray2*. It then returns a dynamic array containing the results of these divisions. If an element value is an uninitialized variable, a null string, or a non-numeric value, **DIVS** parses its value as 0 (zero).

Attempting to divide by zero issues a “Division by zero” error and returns a value of 0 for that element.

Examples

The following example uses the **DIVS** function to divide the elements of two dynamic arrays:

```
a=11:@VM:22:@VM:0:@VM:-7
b=10:@VM:.5:@VM:10:@VM:42
PRINT DIVS(a,b)
! returns 1.1v44v0v-.1666666666667
```

See Also

- [ADDS](#) function
- [MODS](#) function
- [MULS](#) function
- [SUBS](#) function
- [Dynamic Arrays](#)

DOWNCASE

Converts alphabetic characters to lowercase.

```
DOWNCASE(string)
```

Arguments

<i>string</i>	Any valid string or numeric expression.
---------------	---

Description

The **DOWNCASE** function returns a string of characters with all uppercase letters converted to lowercase. Characters other than uppercase letters are passed through unchanged. If you specify a null string, **DOWNCASE** returns a null string.

By default, **DOWNCASE** performs case conversion on ANSI Latin-1 letters. To perform case conversion on letters in other character sets, you must set the appropriate locale.

The **OCNV** function with the “MCL” option is functionally identical to the **DOWNCASE** function. To convert lowercase to uppercase, use the **UPCASE** function.

Examples

The following example uses the **DOWNCASE** function to return a string in all lowercase:

```
PRINT DOWNCASE("InterSystems"); ! Returns "intersystems"
```

See Also

- [UPCASE](#) function
- [OCONV](#) function

DQUOTE

Encloses a value in double quotation marks.

```
DQUOTE(string)
```

Arguments

<i>string</i>	Any expression that resolves to a string or a numeric. <i>string</i> may be a dynamic array .
---------------	---

Description

The **DQUOTE** function returns *string* enclosed in double quotation marks. Using **DQUOTE** increases the length of *string* by 2 characters. If *string* is the null string ("") or an undefined variable, **DQUOTE** returns a string consisting of two quotation mark characters, a string with a length of 2. This should not be confused with the null string (""), which has a length of 0.

The **DQUOTE** function converts a numeric to canonical form before enclosing it in quotation marks. **DQUOTE** does not convert a numeric string to canonical form.

The **QUOTE** function is functionally identical to **DQUOTE**. The **SQUOTE** function is similar, except that it encloses *string* with single quotation marks, rather than double quotation marks.

Examples

The following example uses the **DQUOTE** function to convert a numeric to a string enclosed in double quotation marks:

```
quoted = DQUOTE(+007.000)
PRINT quoted;           ! Returns "7"
PRINT LEN(quoted);     ! Returns 3
```

See Also

- [QUOTE](#) function
- [SQUOTE](#) function
- [LEN](#) function
- [PRINT](#) statement

DTX

Converts a number from decimal to hexadecimal.

```
DTX(decnum[,width])
```

Arguments

<i>decnum</i>	A decimal integer.
<i>width</i>	<i>Optional</i> — A positive integer specifying the number of digits of the returned value, for the purpose of zero-padding.

Description

The **DTX** function returns a decimal integer converted to hexadecimal. The *decnum* value can be a positive or negative integer. If positive, **DTX** returns the number of hexadecimal digits needed to express it. If *width* is larger than the needed number of hexadecimal digits, **DTX** pads the returned hexadecimal number with leading zeros. If *decnum* is a negative integer, **DTX** returns high values. For example, `DTX(-1)` returns `FFFFFFFFFFFFFFFF`. If *decnum* is zero, the null string, an undefined variable, or a non-numeric string, **DTX** returns 0. If *decnum* is a mixed numeric string, the numeric part is parsed until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7.

Use **XTD** to convert from hexadecimal to decimal.

Examples

The following examples use the **DTX** function to return an integer converted to hexadecimal:

```
PRINT DTX(12);           ! Returns "C"
PRINT DTX(12,4);       ! Returns "000C"
PRINT DTX(199);        ! Returns "C7"
PRINT DTX(199,1);      ! Returns "C7"
```

See Also

- [XTD](#) function

EBCDIC

Converts a string from ASCII to EBCDIC.

```
EBCDIC(string)
```

Arguments

<i>string</i>	A string of characters in ASCII representation.
---------------	---

Description

The **EBCDIC** function takes one or more ASCII codes and returns the corresponding EBCDIC character(s). This is the inverse of the **ASCII** function.

The **CHAR** function takes an ASCII code and returns the corresponding character. The **SEQ** function takes a character and returns the corresponding ASCII code.

Examples

The following example uses the **EBCDIC** function to return the characters associated with the specified EBCDIC code string:

```
estring=EBCDIC("ABCDEFGG")
astring=ASCII(estring)
PRINT astring
! returns "ABCDEFGG"
```

The following example shows the use of the **SEQ** and **CHAR** functions with the **EBCDIC** function:

```
PRINT SEQ(ASCII("A"))
! returns 159
PRINT EBCDIC(CHAR(159))
! returns "A"
```

See Also

- [ASCII](#) function
- [CHAR](#) function

- [SEQ](#) function
- [Strings](#)

EQS

Performs an equality comparison on elements of two dynamic arrays.

```
EQS(dynarray1, dynarray2)
```

Arguments

<i>dynarray</i>	Any valid dynamic array of numeric values.
-----------------	--

Description

The **EQS** function compares each corresponding numeric element from two dynamic arrays for equality. It returns a dynamic array of boolean values, in which each element comparison is represented by a 1 (equal) or a 0 (not equal). **EQS** removes signs and leading and trailing zeros from element values before making the comparison. If an element is an uninitialized variable, a null string, or a non-numeric value, **EQS** assigns it a value of 0 for the purpose of this comparison.

For two elements to be compared, they must be on the same dynamic array level. For example, you cannot compare a value mark (@VM) dynamic array element to a subvalue mark (@SM) dynamic array element.

If an element in one dynamic array has no corresponding element in the other dynamic array, a 0 (not equal) comparison is returned for that element.

The **EQS** function is the functional opposite of the **NES** function.

Examples

The following example uses the **EQS** function to return an equality comparison for each of the elements in dynamic arrays *a* and *b*:

```
a=11:@VM:-22:@VM:-33:@VM:44
b=11:@VM:-24:@VM:0:@VM:44
PRINT EQS(a,b)
! returns 1v0v0v1
```

See Also

- [GES](#) function
- [GTS](#) function
- [LES](#) function
- [LTS](#) function
- [NES](#) function
- [Dynamic Arrays](#)

EREPLACE

Replaces a substring in a string.

```
EREPLACE(string,substring,replacement[,occurrence[,begin]])
```

Arguments

<i>string</i>	An expression that resolves to a string .
<i>substring</i>	A expression that resolves to a substring found within <i>string</i> .
<i>replacement</i>	A expression that resolves to the substring used to replace <i>substring</i> .
<i>occurrence</i>	<i>Optional</i> — An integer count specifying how many occurrences of <i>substring</i> to replace. The default is to replace all occurrences. When using <i>begin</i> , you can set this argument to 0 to replace all occurrences found.
<i>begin</i>	<i>Optional</i> — An integer count specifying the instance of <i>substring</i> with which to begin replacement. The default is to begin with the first instance of <i>substring</i> .

Description

The **EREPLACE** function replaces each occurrence of substring in a string with a new value. Whether to replace all instances of *substring* is specified by the optional *occurrence* and *begin* arguments. If these are omitted, all occurrences of *substring* are replaced by *replacement*. The *replacement* string can be longer or shorter than the *substring* it replaces.

If *substring* is not found in *string*, **EREPLACE** returns *string* unchanged. If *substring* is the empty string ("") the *replacement* string is appended to the beginning of *string*.

If *replacement* is the null string, **EREPLACE** removes *substring* from *string*.

Examples

The following example uses the **EREPLACE** function to replace all instances of a substring:

```
x="The slow brown fox slowly leapt"
PRINT EREPLACE(x,"slow","quick")
! Returns "The quick brown fox quickly leapt"
```

The following example also replaces all instances of a substring:

```
x="The slow brown fox slowly leapt"
PRINT EREPLACE(x,"slow","quick",0,1)
! Returns "The quick brown fox quickly leapt"
```

The following example appends the *replacement* value to the string:

```
x="there was a slow brown fox"
PRINT EREPLACE(x,"","Once upon a time ")
! Returns "Once upon a time there was a slow brown fox"
```

See Also

- [REMOVE](#) statement
- [EXTRACT](#) function
- [Strings](#)

EXP

Returns e (the base of natural logarithms) raised to a power.

```
EXP (number)
```

Arguments

<i>number</i>	Any valid number within the following range: On a Windows system, if the value of <i>number</i> is greater than 335, a runtime error occurs; if the value of <i>number</i> is less than -295, EXP returns zero (0).
---------------	--

Description

The **EXP** function takes the natural log constant e and raises it to the power specified by the *number* argument. The constant e (**EXP(1)**) is approximately 2.718282.

The **EXP** function complements the action of the **LN** function and is sometimes referred to as the antilogarithm.

In Caché ObjectScript, the corresponding function is \$ZEXP.

Examples

The following example uses the **EXP** function to calculate e raised to the power of each of the integers -10 through 10:

```
FOR x = -10 TO 10
PRINT "Natural log to the power of ",x," = ",EXP(x)
NEXT
```

The following example uses the **EXP** function to return the hyperbolic sine of an angle:

```
MyAngle = 1.3
! Define angle in radians.
MyHSin = (EXP(MyAngle) - EXP(-1 * MyAngle)) / 2
! Calculate hyperbolic sine.
PRINT MyHSin
```

See Also

- [LN function](#)
- [Derived Math Functions](#)

EXTRACT

Finds the data value of an element of a dynamic array by delimiter position.

```
EXTRACT(dynarray, f[, v[, s]])
```

Arguments

<i>dynarray</i>	Any valid dynamic array.
<i>f</i>	An integer specifying the Field level of the dynamic array from which to access the data. Fields are counted from 1.
<i>v</i>	<i>Optional</i> — An integer specifying the Value level of the dynamic array from which to access the data. Values are counted from 1 within a Field.
<i>s</i>	<i>Optional</i> — An integer specifying the Subvalue level of the dynamic array from which to access the data. Subvalues are counted from 1 within a Value.

Description

The **EXTRACT** function returns the data value from one element of a dynamic array. Which element to access is specified by the *f*, *v*, and *s* integers. For example, if *f*=2 and *v*=3, this means access the third value from the second field. If *f*=2 and *v* is not specified, this means to access the entire second field.

If lower level delimiters exist in *dynarray*, setting an upper level to 0, the null string, a non-numeric value, or an undefined variable is equivalent to setting it to 1.

If lower level delimiters do not exist in *dynarray*, setting this non-existent lower level to 1, 0, the null string, a non-numeric value, or an undefined variable has no effect on retrieving the data value in the level above it.

You can also use the $\langle \rangle$ operator to extract an element value from a dynamic array. For further details, see the [Dynamic Arrays](#) page of this manual.

Examples

The following example uses the **EXTRACT** function to access the second value from the first field of a dynamic array:

```
cities="New York":@VM:"London":@VM:
"Chicago":@VM:"Boston":@VM:"Los Angeles"
PRINT EXTRACT(cities,1,2)
! Returns: "London"
```

The following examples all return “London”, because the higher level Field Mark value is equivalent to 1:

```
cities="New York":@VM:"London":@VM:  
"Chicago":@VM:"Boston":@VM:"Los Angeles"  
PRINT EXTRACT(cities,1,2)  
PRINT EXTRACT(cities,0,2)  
PRINT EXTRACT(cities,"",2)
```

The following examples all return “London”, because the lower Subvalue Mark level does not exist:

```
cities="New York":@VM:"London":@VM:  
"Chicago":@VM:"Boston":@VM:"Los Angeles"  
PRINT EXTRACT(cities,1,2,0)  
PRINT EXTRACT(cities,1,2,1)  
PRINT EXTRACT(cities,1,2,"")
```

See Also

- [FIND](#) statement
- [FINDSTR](#) statement
- [REMOVE](#) statement
- [REPLACE](#) function
- [Dynamic Arrays](#)
- [Variables](#)

FADD

Adds two floating point numbers.

```
FADD(num1 , num2 )
```

Arguments

<i>num</i>	Any expression that evaluates to a valid numeric value.
------------	---

Description

The **FADD** function adds two numbers and returns the result. If a *num* value is an uninitialized variable, a null string, or a non-numeric value, **FADD** parses its value as 0 (zero).

You can perform the same operation using the addition operator (+). Refer to the [Operators](#) page of this manual.

Arithmetic Operations

- To perform arithmetic operations on floating point numbers, use the **FADD**, **FSUB**, **FMUL**, and **FDIV** functions, or use the standard arithmetic operators.
- To perform arithmetic operations on numeric strings, use the **SADD**, **SSUB**, **SMUL**, and **SDIV** functions.
- To perform integer division, use the **DIV** function. To perform modulo division, use the **MOD** function.
- To perform arithmetic operations on corresponding elements of dynamic arrays, use the **ADDS**, **SUBS**, **MULS**, **DIVS**, and **MODS** functions.
- To add together the element values within a single dynamic array, use either the **SUM** function (for single-level dynamic arrays) or the **SUMMATION** function (for multi-level dynamic arrays).
- To perform numeric comparison operations, use the **SCMP** function, or use the standard comparison operators.

Examples

The following example uses the **FADD** function to add two floating point numbers:

```
a=11.95
b=10.25
PRINT FADD(a,b); ! returns 22.2
```

See Also

- [SADD](#) function
- [ADDS](#) function
- [SUM](#) function
- [SUMMATION](#) function
- [Operators](#)

FDIV

Divides two floating point numbers.

```
FDIV ( num1 , num2 )
```

Arguments

<i>num1</i>	The dividend. Any valid numeric or numeric string .
<i>num2</i>	The divisor. Any valid non-zero numeric or numeric string .

Description

The **FDIV** function divides the value of *num1* by *num2*, and returns the product. If a value is 0, an uninitialized variable, a null string, or a non-numeric value, **FDIV** parses its value as 0 (zero). If *num1* is 0, **FDIV** returns a result of 0. If *num2* is 0, **FDIV** generates a syntax error.

You can perform the same operation using the Division operator (/). Refer to the [Operators](#) page of this manual.

Arithmetic Operations

- To perform arithmetic operations on floating point numbers, use the [FADD](#), [FSUB](#), [FMUL](#), and **FDIV** functions, or use the standard arithmetic operators.
- To perform arithmetic operations on numeric strings, use the [SADD](#), [SSUB](#), [SMUL](#), and [SDIV](#) functions.
- To perform integer division, use the [DIV](#) function. To perform modulo division, use the [MOD](#) function.
- To perform arithmetic operations on corresponding elements of dynamic arrays, use the [ADDS](#), [SUBS](#), [MULS](#), [DIVS](#), and [MODS](#) functions.
- To perform numeric comparison operations, use the [SCMP](#) function, or use the standard comparison operators.

Examples

The following example uses the **FDIV** and the **DIV** functions to divide the same two floating point numbers:

```

a=11.95
b=10.25
PRINT FDIV(a,b); ! returns 1.165853658536585366
PRINT DIV(a,b); ! returns 1

```

See Also

- [SDIV](#) function
- [DIVS](#) function
- [DIV](#) function
- [MOD](#) function
- [MODS](#)
- [Operators](#)

FIELD

Returns the specified substring, based on a delimiter.

```
FIELD(string,delimiter,count[,range])
```

Arguments

<i>string</i>	The target string from which a substring is to be returned. If you specify a null string ("") as the target string, FIELD always returns a null string.
<i>delimiter</i>	A single character, specified as a number or a quoted string. This character is used as a delimiter to identify substrings. This character cannot also be used as a data value within <i>string</i> . The delimiter characters used in dynamic arrays are listed in the Dynamic Arrays general concepts page of this manual.
<i>count</i>	An integer that specifies which substring to return from the target string. Substrings are separated by a <i>delimiter</i> , and counted from 1. A decimal number is truncated to an integer. A string is parsed as a number until a non-numeric character is encountered. Thus "7dwarves" is parsed as 7. A <i>count</i> value of 0, a negative number, the null string, or a non-numeric string is the same as <i>count</i> =1.
<i>range</i>	<i>Optional</i> — An integer specifying the number of delimited substrings to return, starting with <i>count</i> . If omitted, the default is 1.

Description

The **FIELD** function returns the substring which is the *n*th piece of *string*, where the integer *n* is specified by the *count* parameter, and substrings are separated by a *delimiter* character. The delimiter itself is not returned.

If *count* is 1, **FIELD** returns the first piece of the string. This is the piece of the string from the beginning of the string to the first delimiter. If the first character of the string is a delimiter, *count*=1 returns the null string.

You can follow the **FIELD** function with the **COL1** function to determine the string position of the start delimiter for the returned substring. If *count* is 1, **COL1** returns 0. You can determine the end delimiter position by calling the **COL2** function.

If *count* is greater than the number of delimited substrings, **FIELD** returns the null string. In this case, **COL1** and **COL2** both return 0.

If you specify a *delimiter* that is not located in *string* and *count*=1, **FIELD** returns the entire *string*. If *count*>1, **FIELD** returns the null string.

If you specify the null string as a *delimiter*, **FIELD** returns the entire *string*, regardless of the value of *count*.

If the optional *range* argument is set to an integer value greater than 1, that number of sequential delimited substrings is returned as a single string. Delimiters within the string are included. If *range* is a decimal number, it is truncated to its integer value. Setting *range* to any value other than a numeric 2 or greater is treated as setting it to 1. If *range* is larger than the number of remaining substrings in the string, the remaining substrings are returned.

Note: The **FIELD** and **GROUP** functions are functionally identical.

Examples

The following example uses the **FIELD** function to return the first five delimited items in a string:

```
colors="Red^Green^Blue^Yellow^Orange^Black"  
FOR x=1 TO 5  
    PRINT FIELD(colors,"^",x)  
NEXT
```

The following example uses the **FIELD** function to return the first three elements in a dynamic array:

```
colors="Red":@VM:"Green":@VM:"Blue":@VM:"Yellow"  
FOR x=1 TO 3  
    PRINT FIELD(colors,CHAR(253),x)  
NEXT
```

See Also

- [FIELDS](#) function
- [GROUP](#) function
- [COL1](#) function
- [COL2](#) function
- [Strings](#)
- [Dynamic Arrays](#)

FIELDS

Returns a dynamic array of substrings, based on a delimiter.

```
FIELDS(dynarray, delimiter, count[, range])
```

Arguments

<i>dynarray</i>	The target dynamic array from which a dynamic array of substrings is to be returned.
<i>delimiter</i>	A single character, specified as a number or a quoted string. This character is used as a delimiter to identify substrings within elements. This character cannot also be used as a data value within <i>dynarray</i> . The delimiter characters used in dynamic arrays are listed in the Dynamic Arrays general concepts page of this manual.
<i>count</i>	An integer that specifies which substring to return from each element of <i>dynarray</i> . Substrings are separated by a <i>delimiter</i> , and counted from 1. A decimal number is truncated to an integer. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7. A <i>count</i> value of 0, a negative number, the null string, or a non-numeric string is the same as <i>count</i> =1.
<i>range</i>	<i>Optional</i> — An integer specifying the number of delimited substrings to return for each element, starting with <i>count</i> . If omitted, the default is 1.

Description

The **FIELDS** function returns a dynamic array of substrings. Each substring is the *n*th piece of each element, where the integer *n* is specified by the *count* parameter, and substrings are separated by a *delimiter* character. The delimiter itself is not returned.

If *count* is 1, **FIELDS** returns the first piece of each element. This is the piece of the string from the beginning of the element to the first delimiter. If the first character of the element is a delimiter, *count*=1 returns the null string.

If *count* is greater than the number of delimited substrings in an element, **FIELDS** returns the null string for that element.

If you specify a *delimiter* that is not located in *dynarray* and *count*=1, **FIELDS** returns the entire *dynarray* as a single element. If *count*>1, **FIELDS** returns the null string.

If you specify the null string as a *delimiter*, **FIELDS** returns the entire *dynarray*, regardless of the value of *count*.

If the optional *range* argument is set to an integer value greater than 1, that number of sequential delimited substrings is returned as a single string. Delimiters within the string are included. If *range* is a decimal number, it is truncated to its integer value. Setting *range* to any value other than a numeric 2 or greater is treated as setting it to 1. If *range* is larger than the number of remaining substrings in the element, the remaining substrings are returned.

The **FIELDS** function returns delimited substrings from a dynamic array. The **FIELD** and **GROUP** functions can be used to return a delimited substring from a string.

Examples

The following example uses the **FIELDS** function to return the area code from each telephone number element in an array, using the hyphen (-) as a delimiter:

```
tele="617-123-4567":@VM:"401-555-4321":@VM:"603-987-6543":@VM:"508-246-8024"
areacodes=FIELDS(tele,"-",1)
PRINT areacodes
! Returns: 617v401v603v508
```

See Also

- [FIELD](#) function
- [GROUP](#) function
- [Strings](#)
- [Dynamic Arrays](#)

FIELDSTORE

Replaces data in a delimited string.

```
FIELDSTORE(string, delimiter, count, multiple, newval)
```

Arguments

<i>string</i>	The name of the string to be modified. <i>string</i> can be a dynamic array.
<i>delimiter</i>	A character that serves as a delimiter within <i>string</i>
<i>count</i>	An integer that specifies which delimited string to use as the starting point for the replacement operation.
<i>multiple</i>	An integer specifying how many delimited strings to replace with <i>newval</i> .
<i>newval</i>	The data to be inserted.

Description

The **FIELDSTORE** function replaces one or more delimited substrings in *string* with a specified *newval*, then returns the resulting string. **FIELDSTORE** adds and removes delimiters as needed.

- To replace a delimited substring with another substring, specify a *count* that corresponds to an existing delimited string and *multiple*=1.
- To replace more than one delimited substrings with a single delimited substring, specify a *count* that corresponds to an existing delimited string and a *multiple* greater than one. Both the replaced substrings and their delimiters are removed.
- To append a delimited substring to the end of *string*, specify a *count* greater than the number of existing delimited strings. **FIELDSTORE** adds the appropriate number of delimiter characters, if necessary, before the *newval* substring.
- To prepend a delimited substring to the beginning of *string*, specify a *count*=1 and *multiple*=0. **FIELDSTORE** appends a delimiter character and the *newval* substring.
- To delete a delimited substring, specify a *count* that corresponds to an existing delimited string and specify *newval* as the empty string.

If *delimiter* is not found in *string*, and *count* is 1, 0, or the null string, *newval* replaces *string* and is returned with no delimiters. If *delimiter* is not found in *string*, and *count* is > 1, the

specified delimiter and *newval* are appended to *string*. The number of delimiters appended being *count* minus 1.

Examples

The following example uses the **FIELDSTORE** function to replace the first delimited substring in a string:

```
cities="New York^London^Chicago^Boston^Los Angeles"
PRINT FIELDSTORE(cities,"^",1,1,"Providence")
! Returns: "Providence^London^Chicago^Boston^Los Angeles"
```

The following example uses the **FIELDSTORE** function to replace the second Value Mark delimited substring in a dynamic array:

```
cities="New York":@VM:"London":@VM:"Chicago":@VM:"Boston":@VM:"Los Angeles"
PRINT cities
! Returns: "New YorkvLondonvChicagovBostonvLos Angeles"
PRINT FIELDSTORE(cities,CHAR(253),2,1,"Providence")
! Returns: "New YorkvProvidencevChicagovBostonvLos Angeles"
```

The following example uses the **FIELDSTORE** function to replace the second Value Mark delimited substring and the next two substrings in a dynamic array:

```
cities="New York":@VM:"London":@VM:"Chicago":@VM:"Boston":@VM:"Los Angeles"
PRINT cities
! Returns: "New YorkvLondonvChicagovBostonvLos Angeles"
PRINT FIELDSTORE(cities,CHAR(253),2,3,"Providence")
! Returns: "New YorkvProvidencevLos Angeles"
```

See Also

- [INS](#) statement
- [COUNTS](#) function
- [DELETE](#) function
- [INSERT](#) function
- [EXTRACT](#) function
- [Dynamic Arrays](#)

FILEINFO

Returns information about an open file.

```
FILEINFO( filevar , key )
```

Arguments

<i>filevar</i>	A file variable name used to refer to the file in Caché MVBasic.
<i>key</i>	An integer code used to specify what file information to return. Available values are 0 through 20.

Description

The **FILEINFO** function returns various types of information about an open file. You must specify a *filevar* supplied by an open statement, such as **OPEN** or **OPENSEQ**. You can use `FILEINFO(filevar , 0)` to determine if *filevar* is valid.

The following are the available *key* options and return values:

0	File variable: 1 if <i>filevar</i> is valid. Otherwise 0.
1	VOC name: The VOC name of a MultiValue file. For example, <code>myfile</code> . If <i>filevar</i> does not refer to a MultiValue file, returns a null string.
2	Pathname: For a MultiValue file, the name of a Caché global variable. For example, <code>^ "USER" myfile</code> . For a sequential file, the fully-qualified pathname of the file, as specified in the OPENSEQ statement. If the file does not exist, this is returned as a directory path.
3	File type: 0=file specified in open does not exist. 1=Static hashed file (the default for MultiValue data files). 5=sequential file
4	Hashing algorithm: If none, returns a null string.
5	Current modulus: always 1.
6	Minimum modulus: If none, returns a null string.
7	Group size: If none, returns a null string.
8	Large record size: If none, returns a null string.
9	Merge load parameter: If none, returns a null string.
10	Split load parameter: If none, returns a null string.

11	Current loading: Expressed as a percentage. If none, returns a null string.
12	Node name: If the file is on the local system returns a null string.
13	Secondary indexes: 1=exist. 0=do not exist.
14	Current line number: 0 if beginning of file.
15	Part number: For a distributed file, a list of currently open part numbers. If none, returns a null string.
16	Status codes: For a distributed file, a list of status codes for I/O operations on each part. If none, returns a null string.
17	Recoverable: 1=file marked as recoverable. 0=file not marked as recoverable.
18	Always returns a null string.
19	Always returns a null string.
20	NLS enabled: If enabled returns the file map name. Otherwise, returns a null string.

Examples

The following example opens a sequential file, tests the file variable, then uses the file variable to return the file's pathname and the file type (in this case, type 5):

```
OPENSEQ "C:\temp\file1" TO myfile
IF FILEINFO(myfile,0)=1
  THEN PRINT "valid file variable"
  ELSE PRINT "file variable not valid"
END
PRINT "File pathname is:",FILEINFO(myfile,2)
PRINT "File type is:",FILEINFO(myfile,3)
CLOSESEQ myfile
```

See Also

- [OPEN](#) statement
- [OPENSEQ](#) statement

FIX

Returns a floating point number with the specified number of decimal digits.

```
FIX(number[,precision[,mode]])
```

Arguments

<i>number</i>	Any valid numeric expression, specified as a number or a numeric string.
<i>precision</i>	<i>Optional</i> — An integer specifying the number of decimal digits of precision. The default is 4.
<i>mode</i>	<i>Optional</i> — A boolean flag that specifies whether to round or truncate <i>number</i> . 0=round; 1=truncate. The default is 0.

Description

The **FIX** function takes a floating point number and returns this number rounded or truncated to the specified number of decimal digits. The *precision* is the maximum number of decimal digits. **FIX** does not pad a number with trailing zeros, and removes trailing zeros that result from the rounding process. Thus `FIX(12.99,1)` returns 13, not 13.0.

The *precision* argument is optional. If not specified, **FIX** either takes its precision from a preceding **PRECISION** command, or takes the default precision of 4. A value of 0, the null string, or a non-numeric string does not set *precision*, and the default precision is taken. You must specify a *precision* value to specify a *mode* value.

Examples

The following example shows the uses of the **FIX** function:

```
PRINT FIX(123.987654);      ! Returns 123.9877
PRINT FIX(123.987654,2);   ! Returns 123.99
PRINT FIX(123.987654,1);   ! Returns 124
PRINT FIX(123.987654,0);   ! Returns 123.9877
PRINT FIX(123.987654,2,0); ! Returns 123.99
PRINT FIX(123.987654,2,1); ! Returns 123.98
```

See Also

- [PRECISION](#) command

FMT

Formats a value for display.

```
FMT(string, format)
```

Arguments

<i>string</i>	A string expression to be formatted for display.
<i>format</i>	A quoted string consisting of positional letter and number codes specifying the display format for <i>string</i> .

Description

The **FMT** function returns the *string* value formatted as specified by *format*. This formatting may include padding or rounding/truncating of *string*. The most common use for **FMT** is to provide a uniform display format for decimal numbers.

The *format* string has the following format:

```
wfRn
```

w	<i>Optional</i> — The overall width of the display field, specified as a positive integer.
f	<i>Optional</i> — A fill character, specified as a single character. The default fill character is a blank space.
R	The letter “R” or “L” specifying right or left justification.
n	The number of digits to the right of the decimal place, specified as a positive integer.

The most basic *format* is a string of the format "*wRn*", where “R” specifies right justification and *n* is the number of digits to the right of the decimal point to display. If *string* has fewer digits than *n*, zero padding is added. If *string* has more digits than *n*, the number is rounded to the specified number of digits.

You can use *w* (width) and *f* (fill) formatting to make a display field a standard width. For example: "*10#R5*", where “10” is the overall width of the display field, “#” is the fill character to use to fill out the display field. Because “R” indicates right justification, these fill characters will appear to the left of the *string* value. The “5” indicates that the *string* value is to have 5 digits to the right of the decimal place. The width value must be larger than or equal to the

number of characters (including the decimal point) of *string* after adjusting the number of digits to the right of the decimal place. The fill character is optional; if omitted, filling is done with blank spaces. If the fill character is a number or the letters “L”, “R” or “T” it must be enclosed in single quotes. For example: 10'0'R2.

Examples

The following examples use “Rn” formatting to format a numeric values so that it displays 4 decimal digits. Note that both zero padding and rounding are performed as needed:

```
PRINT FMT(1.2, "R4");      ! Returns 1.2000
PRINT FMT(1.77777, "R4"); ! Returns 1.7778
PRINT FMT(.4, "R4");      ! Returns 0.4000
PRINT FMT(0, "R4");       ! Returns 0.0000
```

See Also

- [FMTS](#) function
- [LEN](#) function
- [RIGHT](#) function

FMTS

Formats each element of a dynamic array for display.

```
FMTS(dynarray, format)
```

Arguments

<i>dynarray</i>	A dynamic array to be formatted for display.
<i>format</i>	A quoted string consisting of positional letter and number codes specifying the display format for <i>dynarray</i> .

Description

The **FMTS** function returns the *dynarray* value with each element formatted as specified by *format*. This formatting may include padding or rounding/truncating of element values. The most common use for **FMTS** is to provide a uniform display format for decimal numbers.

The *format* string has the following format:

wfRn	
w	<i>Optional</i> — The overall width of the display field, specified as a positive integer.
f	<i>Optional</i> — A fill character, specified as a single character. The default fill character is a blank space.
R	The letter “R” or “L” specifying right or left justification.
n	The number of digits to the right of the decimal place, specified as a positive integer.

The most basic *format* is a string of the format "Rn", where “R” specifies right justification and *n* is the number of digits to the right of the decimal point to display. If *string* has fewer digits than *n*, zero padding is added. If *string* has more digits than *n*, the number is rounded to the specified number of digits.

You can use *w* (width) and *f* (fill) formatting to make a display field a standard width. For example: "10#R5", where “10” is the overall width of the display field, “#” is the fill character to use to fill out the display field. Because “R” indicates right justification, these fill characters will appear to the left of the element value. The “5” indicates that the element value is to have 5 digits to the right of the decimal place. The width value must be larger than or equal to the number of characters (including the decimal point) of the element value after adjusting the number of digits to the right of the decimal place. The fill character is optional; if omitted, filling is done with blank spaces. If the fill character is a number or the letters “L”, “R” or “T” it must be enclosed in single quotes. For example: 10'0'R2.

Examples

The following example uses “Rn” formatting to format the elements of a dynamic array so that all elements display 4 decimal digits. Note that both zero padding and rounding are performed as needed:

```
nums=1.2:@VM:2.45:@VM:3:@VM:4.9999999:@VM:0
PRINT FMFS(nums,"R4")
! Returns: 1.2000v2.4500v3.0000v5.0000v0.0000
```

See Also

- [FMT](#) function
- [LEN](#) function

- [RIGHT](#) function
- [Dynamic Arrays](#)

FMUL

Multiplies two floating point numbers.

```
FMUL ( num1 , num2 )
```

Arguments

<i>num</i>	Any valid numeric or numeric string .
------------	---

Description

The **FMUL** function multiplies two numbers and returns a numeric value. If a *num* value is an uninitialized variable, a null string, or a non-numeric value, **FMUL** parses its value as 0 (zero).

You can perform the same operation using the multiplication operator (*). Refer to the [Operators](#) page of this manual.

Arithmetic Operations

- To perform arithmetic operations on floating point numbers, use the [FADD](#), [FSUB](#), **FMUL**, and [FDIV](#) functions, or use the standard arithmetic operators.
- To perform arithmetic operations on numeric strings, use the [SADD](#), [SSUB](#), [SMUL](#), and [SDIV](#) functions.
- To perform integer division, use the [DIV](#) function. To perform modulo division, use the [MOD](#) function.
- To perform arithmetic operations on corresponding elements of dynamic arrays, use the [ADDS](#), [SUBS](#), [MULS](#), [DIVS](#), and [MODS](#) functions.
- To perform numeric comparison operations, use the [SCMP](#) function, or use the standard comparison operators.

Examples

The following examples use the **FMUL** function to multiply two floating point numbers:

```
PRINT FMUL(3.33,78.0);    ! returns 259.74
```

See Also

- [SMUL](#) function
- [MULS](#) function
- [Operators](#)

FSUB

Subtracts two floating point numbers.

```
FSUB(num1 , num2)
```

Arguments

<i>num1</i>	The minuend. Any valid numeric or numeric string .
<i>num2</i>	The subtrahend. Any valid numeric or numeric string .

Description

The **FSUB** function subtracts *num2* from *num1*, expressed as either numbers or as strings, and returns the result. Leading plus signs and leading and trailing zeros are ignored. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7. If a *num* value is an uninitialized variable, a null string, or a non-numeric value, **FSUB** parses its value as 0 (zero).

The **FSUB** function performs a subtraction on two numbers and returns the result. You can perform the same operation using the subtraction operator (-). Refer to the [Operators](#) page of this manual.

Arithmetic Operations

- To perform arithmetic operations on floating point numbers, use the [FADD](#), **FSUB**, [FMUL](#), and [FDIV](#) functions, or use the standard arithmetic operators.
- To perform arithmetic operations on numeric strings, use the [SADD](#), [SSUB](#), [SMUL](#), and [SDIV](#) functions.
- To perform integer division, use the [DIV](#) function. To perform modulo division, use the [MOD](#) function.

- To perform arithmetic operations on corresponding elements of dynamic arrays, use the [ADDS](#), [SUBS](#), [MULS](#), [DIVS](#), and [MODS](#) functions.
- To perform numeric comparison operations, use the [SCMP](#) function, or use the standard comparison operators.

Examples

The following example uses the **FSUB** function to subtract two floating point numbers:

```
a=11.95
b=10.25
PRINT FSUB(a,b); ! returns 1.7
```

See Also

- [SSUB](#) function
- [SUBS](#) function
- [Operators](#)

GES

Performs a greater than or equal to comparison on elements of two dynamic arrays.

```
GES(dynarray1 , dynarray2)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
-----------------	---

Description

The **GES** function compares each corresponding numeric element from two dynamic arrays and determines if the first value is greater than or equal to the second. It returns a dynamic array of boolean values in which each element comparison is represented. It returns a 1 if the *dynarray1* element value is greater than or equal to the *dynarray2* element value. It returns a 0 if the *dynarray1* element value is less than the *dynarray2* element value.

GES removes signs and leading and trailing zeros from element values before making the comparison. If an element value is an uninitialized variable, a null string, or a non-numeric value, **GES** assigns it a value of 0 for the purpose of this comparison.

For two elements to be compared, they must be on the same dynamic array level. For example, you cannot compare a value mark (@VM) dynamic array element to a subvalue mark (@SM) dynamic array element.

Examples

The following example uses the **GES** function to return a greater than comparison for each of the elements in dynamic arrays *a* and *b*:

```
a=10:@VM:-22:@VM:-33:@VM:45
b=10:@VM:-23:@VM:0:@VM:44
PRINT GES(a,b)
! returns 1v1v0v1
```

See Also

- [EQS](#) function
- [GTS](#) function
- [LES](#) function
- [LTS](#) function
- [Dynamic Arrays](#)

GETREM

Returns the position of the Remove pointer in a dynamic array.

```
GETREM(dynarray)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
-----------------	---

Description

The **GETREM** function is called following a **REMOVE** function, a **REMOVE** statement, or a **REVMREMOVE** statement. These statements extract successive data elements from a dynamic array. They establish a pointer specifying the position for the next extract from that dynamic array. **GETREM** returns this pointer value. This value is an integer count character position within *dynarray*.

Following a **REMOVE** operation, **GETREM** returns the character position, counting from 1, of the delimiter following the extracted data. If the **REMOVE** attempts to extract data past the end of the dynamic array, **GETREM** returns the length of the dynamic array, plus 1. Subsequent **REMOVE** operations do not change this end-of-dynamic-array pointer value.

A **REVREMOVE** operation decrements this pointer to the delimiter preceding the extracted data. If the **REVREMOVE** attempts to extract data past the beginning of the dynamic array, **GETREM** returns 0.

If *dynarray* has never been accessed by a **REMOVE** or **REVREMOVE**, **GETREM** returns 0. If *dynarray* is changed, its pointer is reset to 0. If *dynarray* does not exist, **GETREM** issues a syntax error.

Examples

The following example uses the **GETREM** function to return the Remove pointer position:

```
cities="Newark":@VM:"New York":@VM:"Boston"
PRINT cities;           ! returns NewarkvNew YorkvBoston
REMOVE val FROM cities SETTING 3
  PRINT val;           ! Returns "Newark"
  PRINT GETREM(cities); ! returns 7
REMOVE val FROM cities SETTING 3
  PRINT val;           ! Returns "New York"
  PRINT GETREM(cities); ! returns 16
REMOVE val FROM cities SETTING 3
  PRINT val;           ! Returns "Boston"
  PRINT GETREM(cities); ! returns 23
REMOVE val FROM cities SETTING 3
  PRINT val;           ! Returns ""
  PRINT GETREM(cities); ! returns 23
```

See Also

- [REMOVE](#) statement
- [REVREMOVE](#) statement
- [REMOVE](#) function
- [Dynamic Arrays](#)

GROUP

Returns the specified substring, based on a delimiter.

```
GROUP(string,delimiter,count[,range])
```

Arguments

<i>string</i>	The target string from which a substring is to be returned. If you specify a null string ("") as the target string, GROUP always returns a null string.
<i>delimiter</i>	A single character, specified as a number or a quoted string. This character is used as a delimiter to identify substrings. This character cannot also be used as a data value within <i>string</i> . The delimiter characters used in dynamic arrays are listed in the Dynamic Arrays general concepts page of this manual.
<i>count</i>	An integer that specifies the substring to return from the target string. Substrings are separated by a <i>delimiter</i> , and counted from 1. A decimal number is truncated to an integer. A string is parsed as a number until a non-numeric character is encountered. Thus "7dwarves" is parsed as 7. A <i>count</i> value of 0, a negative number, the null string, or a non-numeric string is the same as <i>count</i> =1.
<i>range</i>	<i>Optional</i> — An integer specifying the number of delimited substrings to return, starting with <i>count</i> . If omitted, the default is 1.

Description

The **GROUP** function returns the substring which is the *n*th piece of *string*, where the integer *n* is specified by the *count* parameter, and substrings are separated by a *delimiter* character. The delimiter itself is not returned.

If *count* is 1, **GROUP** returns the first piece of the string. This is the piece of the string from the beginning of the string to the first delimiter. If the first character of the string is a delimiter, *count*=1 returns the null string.

You can follow the **GROUP** function with the **COL1** function to determine the string position of the start delimiter for the returned substring. If *count* is 1, **COL1** returns 0. You can determine the end delimiter position by calling the **COL2** function.

If *count* is greater than the number of delimited substrings, **GROUP** returns the null string. In this case, **COL1** and **COL2** both return 0.

If you specify a *delimiter* that is not located in *string* and *count*=1, **GROUP** returns the entire *string*. If *count*>1, **GROUP** returns the null string.

If you specify the null string as a *delimiter*, **GROUP** returns the entire *string*, regardless of the value of *count*.

If the optional *range* argument is set to an integer value greater than 1, that number of sequential delimited substrings is returned as a single string. Delimiters within the string are included. If *range* is a decimal number, it is truncated to its integer value. Setting *range* to any value other than a numeric 2 or greater is treated as setting it to 1. If *range* is larger than the number of remaining substrings in the string, the remaining substrings are returned.

Note: The **GROUP** and **FIELD** functions are functionally identical.

Examples

The following example uses the **GROUP** function to return the first five delimited items in a string:

```
colors="Red^Green^Blue^Yellow^Orange^Black"
FOR x=1 TO 5
    PRINT GROUP(colors,"^",x)
NEXT
```

The following example uses the **GROUP** function to return the first three elements in a dynamic array:

```
colors="Red":@VM:"Green":@VM:"Blue":@VM:"Yellow"
FOR x=1 TO 3
    PRINT GROUP(colors,CHAR(253),x)
NEXT
```

See Also

- [FIELD](#) function
- [COL1](#) function
- [Strings](#)
- [Dynamic Arrays](#)

GTS

Performs a greater than comparison on elements of two dynamic arrays.

```
GTS(dynarray1, dynarray2)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
-----------------	---

Description

The **GTS** function compares each corresponding numeric element from two dynamic arrays and determines which value is greater. It returns a dynamic array of boolean values in which each element comparison is represented. It returns a 1 if the *dynarray1* element value is greater than the *dynarray2* element value. It returns a 0 if the *dynarray1* element value is equal to or less than the *dynarray2* element value.

GTS removes signs and leading and trailing zeros from element values before making the comparison. If an element value is an uninitialized variable, a null string, or a non-numeric value, **GTS** assigns it a value of 0 for the purpose of this comparison.

For two elements to be compared, they must be on the same dynamic array level. For example, you cannot compare a value mark (@VM) dynamic array element to a subvalue mark (@SM) dynamic array element.

Examples

The following example uses the **GTS** function to return a greater than comparison for each of the elements in dynamic arrays *a* and *b*:

```
a=10:@VM:-22:@VM:-33:@VM:45
b=10:@VM:-23:@VM:0:@VM:44
PRINT GTS(a,b)
! returns 0v1v0v1
```

See Also

- [EQS](#) function
- [GES](#) function
- [LES](#) function
- [LTS](#) function

- [Dynamic Arrays](#)

ICONV

Converts a value from external format to internal format.

```
ICONV(ostring, codes)
```

Arguments

<i>ostring</i>	An expression that resolves to a string . It specifies a value in external (output) format.
<i>codes</i>	A quoted string containing one or more code characters that govern the conversion from external format to internal format.

Description

The **ICONV** function is a general-purpose conversion function used to convert from external (output) format to internal (storage) format. Conversions are governed by a set of code characters specific to the type of data to be converted. You must specify at least one mandatory code character. Additional code characters may be included or omitted, but must follow the specified order. The code characters are as follows:

Time conversion: Internal times are stored as the number of seconds elapsed since midnight. ICONV accepts fractional seconds.	“MT”
Date conversion: Internal dates are stored as the number of days elapsed since December 31, 1967. Dates prior to this are stored using a negative number of days.	“D”

The **DATE** and **TIME** functions return internal format values. The **TIMEDATE** function returns external format values.

If an *ostring* is not valid, a null string is returned. If a *codes* value is not valid, the *ostring* is returned unchanged.

You can use the **STATUS** function to determine the success of an **ICONV** conversion. The following status codes are supported: 0=successful conversion; 1=invalid date; 2=invalid *codes* value.

The **ICONV** function converts from external format to internal format. The **OCNV** function converts from internal format to external format.

You can use the **ICONVS** function to convert the elements of a dynamic array from external format to internal format.

Examples

The following examples show date conversions from external to internal format. All of these **ICONV** functions return the internal date 14143:

```
DateConversions:
PRINT ICONV("20 SEP 2006","D")
PRINT ICONV("09-20-2006","D")
PRINT ICONV("09/20/2006","D")
```

The following example shows time conversions from external to internal format:

```
TimeConversions:
PRINT ICONV("13:21","MT");           ! Returns 48060
PRINT ICONV("1:21PM","MT");         ! Returns 48060
PRINT ICONV("13:21:01","MT");       ! Returns 48061
PRINT ICONV("13:21:01.65","MT");    ! Returns 48061.65
```

See Also

- [ICONVS](#) function
- [OCNV](#) function
- [STATUS](#) function
- [DATE](#) function
- [TIME](#) function
- [TIMEDATE](#) function
- [Strings](#)

ICONVS

Converts a dynamic array from external format to internal format.

```
ICONVS (odynarray, codes)
```

Arguments

<i>odynarray</i>	A dynamic array , each element of which specifies a value in external (output) format.
<i>codes</i>	A quoted string containing one or more code characters that govern the conversion from external format to internal format.

Description

The **ICONVS** function is a general-purpose conversion function used to convert the elements of a dynamic array from external (output) format to internal (storage) format. It returns a dynamic array of values. Conversions are governed by a set of code characters specific to the type of data to be converted. You must specify at least one mandatory code character. Additional code characters may be included or omitted, but must follow the specified order. The code characters are as follows:

Time conversion: Internal times are stored as the number of seconds elapsed since midnight. ICONVS accepts fractional seconds.	"MT"
Date conversion: Internal dates are stored as the number of days elapsed since December 31, 1967. Dates prior to this are stored using a negative number of days.	"D"

The **DATE** and **TIME** functions return internal format values. The **TIMEDATE** function returns external format values.

Note: The **ICONVS** function converts dynamic array element values from external format to internal format. The **OCONVS** function converts dynamic array element values from internal format to external format.

You can use the **ICONV** function to convert a single value from external format to internal format.

Examples

The following examples show date conversions from external to internal format. **ICONVS** returns a dynamic array with all of its elements containing the internal date 14143:

```
DateConversions:
  x="20 SEP 2006":@VM:"09-20-2006":@VM:"09/20/2006"
  PRINT ICONVS(x,"D")
```

You can then apply **OCONVS** to the result to return all of these dates in the same output format.

See Also

- [ICONV](#) function
- [OCONVS](#) function
- [DATE](#) function
- [TIME](#) function
- [TIMEDATE](#) function
- [Dynamic Arrays](#)

INDEX

Returns starting position of a substring in a string.

```
INDEX(string,substring,occurs)
```

Arguments

<i>string</i>	The string to be searched for <i>substring</i> . An expression that resolves to a string or a numeric value.
<i>substring</i>	A substring to locate within <i>string</i> .
<i>occurs</i>	A positive integer specifying which occurrence of <i>substring</i> to locate.

Description

The **INDEX** function returns the starting character position, counting from 1, of *substring* within *string*. Matching of *substring* is case-sensitive. You use the *occurs* argument to specify which occurrence of *substring* to return the location of.

INDEX returns 0 for any of the following:

- If *substring* does not occur in *string*.
- If *string* or *substring* is the null string ("").
- If *occurs* specifies more occurrences of *substring* than appear in *string*.
- If *occurs* is 0, a negative number, a decimal number, the null string, a non-numeric string, or an undefined variable.

If *occurs* is a mixed numeric string, the numeric part is parsed until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7.

See Also

- [INDEXS](#) function
- [Strings](#)

INDEXS

Returns the starting position of a substring for each element of a dynamic array.

```
INDEXS (dynarray, substring, occurs)
```

Arguments

<i>dynarray</i>	A dynamic array of elements to be searched for <i>substring</i> . Any valid dynamic array .
<i>substring</i>	A substring to locate within <i>string</i> .
<i>occurs</i>	A positive integer specifying which occurrence of <i>substring</i> to locate.

Description

The **INDEXS** function returns a dynamic array of integers, each integer element containing the starting character position, counting from 1, of *substring* within the corresponding *dynarray* element. Matching of *substring* is case-sensitive. You use the *occurs* argument to specify which occurrence of *substring* to return the location of.

INDEXS returns 0 for a dynamic array element for any of the following:

- If *substring* does not occur in the element.

- If *dynarray* or *substring* is the null string ("").
- If *occurs* specifies more occurrences of *substring* than appear in the element.
- If *occurs* is 0, a negative number, a decimal number, the null string, a non-numeric string, or an undefined variable.

If *occurs* is a mixed numeric string, the numeric part is parsed until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7.

See Also

- [INDEX](#) function
- [Dynamic Arrays](#)
- [Strings](#)

INDICES

Returns information about a file's secondary key indexes.

```
INDICES(file[,indexname])
```

Arguments

<i>file</i>	The name of an open file.
<i>indexname</i>	<i>Optional</i> — The name of a secondary index in <i>file</i> .

Description

The **INDICES** function returns a dynamic array that contains secondary index information for a file.

- If *indexname* is omitted, **INDICES** returns a dynamic array containing the index names of all secondary indexes in *file*. The secondary index names are separated by field marks (@FM).
- If *indexname* is specified, **INDICES** returns a dynamic array containing information about this secondary index. The secondary index information items are separated by field marks (@FM).

See Also

- [Dynamic Arrays](#)

INMAT

Returns the number of array elements.

```
INMAT (array)
```

Arguments

<i>array</i>	<i>Optional</i> — The name of an array. If omitted, the most recently parsed array.
--------------	---

Description

The **INMAT** function returns the number of elements in an array that has been loaded using a **MATREAD**, **MATREADU**, or **MATPARSE** statement.

An array must be dimensioned using the **DIM** (or **DIMENSION**) statement.

See Also

- [DIM](#) statement
- [MATPARSE](#) statement
- [MATREAD](#) statement
- [MATREADU](#) statement

INSERT

Inserts data in a dynamic array.

```
INSERT(dynarray, f[, v[, s]]; expression)
```

Arguments

<i>dynarray</i>	The name of a valid dynamic array.
<i>f</i>	An integer specifying the Field level of the dynamic array in which to insert the data. Fields are counted from 1.
<i>v</i>	<i>Optional</i> — An integer specifying the Value level of the dynamic array in which to insert the data. Values are counted from 1 within a Field.
<i>s</i>	<i>Optional</i> — An integer specifying the Subvalue level of the dynamic array in which to insert the data. Subvalues are counted from 1 within a Value.
<i>expression</i>	The data to be inserted.

Description

The **INSERT** function inserts a data value at the specified dynamic array location, then returns the full dynamic array including this insertion. Which element to insert is specified by the *f*, *v*, and *s* integers. For example, if *f*=2 and *v*=3, this means insert the new data value as the third value in the second field. The **INSERT** function does not overwrite; if there already was a third value, the insert increments its location to the fourth value.

Note that a semicolon (;) is used before *expression* as an argument separator. This is because the *v* and *s* arguments are optional and can be omitted.

If lower level delimiters exist in *dynarray*, setting an upper level to 0, the null string, a non-numeric value, or an undefined variable is equivalent to setting it to 1.

Examples

The following example uses the **INSERT** function to insert the second value in the first field of a dynamic array:

```
cities="New York":@VM:"London":@VM:
"Chicago":@VM:"Boston":@VM:"Los Angeles"
PRINT INSERT(cities,1,2;"Providence")
! Returns: "New YorkvProvidencevLondonvChicagovBostonvLos Angeles"
```

See Also

- [INS](#) statement
- [COUNTS](#) function
- [DELETE](#) function
- [EXTRACT](#) function
- [FIELDSTORE](#) function
- [Dynamic Arrays](#)

ITYPE

Returns the I-type value from the file dictionary.

```
ITYPE(itype)
```

Arguments

<i>itype</i>	The contents of the I-descriptor. Any valid expression.
--------------	---

Description

The **ITYPE** function returns the contents of the compiled I-descriptor. You can read the I-descriptor from a file dictionary into the *itype* variable, then use the **ITYPE** function to return its contents.

An I-descriptor can reference a record ID (@ID) or a field value in a data record @RECORD).

See Also

- [READ](#) statement

KEYIN

Receives a single character of user input.

```
KEYIN( )
```

Arguments

None. The parentheses are mandatory.

Description

The **KEYIN** function is used in interactive programs to receive a single input character from the user. **KEYIN** pauses program execution while awaiting user input. By default, it displays a blinking prompt to receive a single input character. Program execution continues immediately upon input of a character. No Enter key is required.

You can use the **INPUT** statement for user input of more than one character or for other user input options. You cannot use the **DATA** statement to supply a character to **KEYIN**.

Examples

The following example calls the **KEYIN** function to input a single character:

```
x=KEYIN()  
PRINT "input character:",x
```

See Also

- [INPUT](#) statement

LEFT

Returns a specified number of characters from the left end of a string.

```
LEFT(string, length)
```

Arguments

<i>string</i>	String expression from which the leftmost characters are returned.
<i>length</i>	Numeric expression that evaluates to a positive integer indicating how many characters to return. If 0, a zero-length string ("") is returned. Fractional numbers are truncated to an integer. If greater than or equal to the number of characters in string, the entire string is returned. No padding is performed.

Description

The **LEFT** function returns the specified number of characters from the beginning (left end) of a string. If you specify a *length* greater than the string length, the entire string is returned. To determine the number of characters in string, use the **LEN** function.

The **RIGHT** function returns the specified number of characters from the end (right end) of a string.

Examples

The following example uses the **LEFT** function to return the first three characters of MyString:

```
MyString = "InterSystems"
PRINT LEFT(MyString,3);    ! Returns "Int"
```

See Also

- [LEN](#) function
- [RIGHT](#) function

LEN

Returns the number of characters in a string.

```
LEN(string)
```

Arguments

<i>string</i>	Any valid string or numeric expression.
---------------	---

Description

The **LEN** function returns the number of characters in a specified string. **LEN** counts characters, not bytes.

For numerics, prior to determining the length MVBASIC performs all arithmetic operations and converts numbers to canonical form, with leading and trailing zeroes, a trailing decimal point, and all signs removed except a single minus sign. Note that **LEN** does count the decimal point and the minus sign. Numeric strings are not converted to canonical form. An empty string ("") returns a length of 0. An undefined variable returns a length of 0.

Examples

The following example uses the **LEN** function to return the number of characters in a string:

```
PRINT LEN("InterSystems");    ! Returns 12
PRINT LEN(+0099.900);        ! Returns 4
PRINT LEN("0099.900");       ! Returns 8
PRINT LEN(CHAR(960));        ! Returns 1
PRINT LEN("");               ! Returns 0
```

See Also

- [COUNT](#) function
- [LENS](#) function

LENS

Returns the length of each element of a dynamic array.

```
LENS (dynarray)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
-----------------	---

Description

The **LENS** function returns the number of characters in each element of a dynamic array. **LENS** counts characters, not bytes. Results are returned as a dynamic array of integers.

For numerics, prior to determining the length MVBASIC performs all arithmetic operations and converts numbers to canonical form, with leading and trailing zeroes, a trailing decimal point, and all signs removed except a single minus sign. Note that **LENS** does count the decimal point and the minus sign. Numeric strings are not converted to canonical form. An empty string ("") returns a length of 0 for that element. An undefined variable returns a length of 0 for that element.

Examples

The following example uses the **LENS** function to return the number of characters in each element of a dynamic array. Numbers are converted to canonical form:

```
nums=123:@VM:12.300:@VM:++0123.00:@VM:"+123.00":@VM:" "
PRINT LENS(nums); ! Returns 3v4v3v7v0
```

See Also

- [COUNTS](#) function
- [LEN](#) function
- [Dynamic Arrays](#)

LES

Performs a less than or equal to comparison on elements of two dynamic arrays.

```
LES(dynarray1, dynarray2)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
-----------------	---

Description

The **LES** function compares each corresponding numeric element from two dynamic arrays and determines if the first value is less than or equal to the second value. It returns a dynamic array of boolean values in which each element comparison is represented. It returns a 1 if the *dynarray1* element value is less than or equal to the *dynarray2* element value. It returns a 0 if the *dynarray1* element value is greater than the *dynarray2* element value.

LES removes signs and leading and trailing zeros from element values before making the comparison. If an element value is an uninitialized variable, a null string, or a non-numeric value, **LES** assigns it a value of 0 for the purpose of this comparison.

For two elements to be compared, they must be on the same dynamic array level. For example, you cannot compare a value mark (@VM) dynamic array element to a subvalue mark (@SM) dynamic array element.

Examples

The following example uses the **LES** function to return a less than or equal to comparison for each of the elements in dynamic arrays *a* and *b*:

```
a=10:@VM:-22:@VM:-33:@VM:45
b=10:@VM:-23:@VM:0:@VM:44
PRINT LES(a,b)
! returns 1v0v1v0
```

See Also

- [EQS](#) function
- [GES](#) function
- [GTS](#) function
- [LTS](#) function

- [Dynamic Arrays](#)

LN

Returns the natural logarithm of a number.

```
LN ( number )
```

Arguments

<i>number</i>	Any valid positive number.
---------------	----------------------------

Description

The **LN** function returns the natural logarithm of *number*.

The **LN** function complements the action of the **EXP** function, which is sometimes referred to as the antilogarithm.

Examples

The following example uses the **LN** function to calculate the natural logarithm of each of the integers 1 through 10:

```
FOR x=1 TO 10
PRINT "Natural log of ",x," = ",LN(x)
NEXT
```

See Also

- [EXP](#) function
- [Derived Math Functions](#)

LOWER

Lowers dynamic array delimiters to next level.

```
LOWER(dynarray)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
-----------------	---

Description

The **LOWER** function returns a dynamic array with its delimiters converted to the next lower-level delimiters. For example, @VM value mark delimiters become @SM subvalue mark delimiters. When a delimiter cannot be lowered any further, it is returned unchanged.

The available levels, in descending order, are: @IM (CHAR(255)); @FM (CHAR(254)); @VM (CHAR(253)); @SM (CHAR(252)); @TM (CHAR(251)); and CHAR(250).

The **RAISE** function performs the opposite operation, raising the level of dynamic array delimiters to the next higher level.

Examples

The following example uses the **LOWER** function to convert dynamic array delimiters to the next lower level. It then uses the **RAISE** function to reverse this operation:

```
numvm=123:@VM:456:@VM:789:@VM:"10":@VM:"11"
PRINT numvm;           ! Returns 123v456v10v11
numlower = LOWER(numvm)
PRINT numlower;       ! Returns 123s456s10s11
numraise = RAISE(numlower)
PRINT numraise;      ! Returns 123v456v10v11
```

See Also

- [RAISE](#) function
- [Dynamic Arrays](#)

LTS

Performs a less than comparison on elements of two dynamic arrays.

```
LTS(dynarray1, dynarray2)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
-----------------	---

Description

The **LTS** function compares each corresponding numeric element from two dynamic arrays and determines which value is lesser. It returns a dynamic array of boolean values in which each element comparison is represented. It returns a 1 if the *dynarray1* element value is less than the *dynarray2* element value. It returns a 0 if the *dynarray1* element value is equal to or greater than the *dynarray2* element value.

LTS removes signs and leading and trailing zeros from element values before making the comparison. If an element value is an uninitialized variable, a null string, or a non-numeric value, **LTS** assigns it a value of 0 for the purpose of this comparison.

For two elements to be compared, they must be on the same dynamic array level. For example, you cannot compare a value mark (@VM) dynamic array element to a subvalue mark (@SM) dynamic array element.

Examples

The following example uses the **LTS** function to return a less than comparison for each of the elements in dynamic arrays *a* and *b*:

```
a=10:@VM:-22:@VM:-33:@VM:45
b=10:@VM:-23:@VM:0:@VM:44
PRINT LTS(a,b)
! returns 0v0v1v0
```

See Also

- [EQS](#) function
- [GES](#) function
- [GTS](#) function
- [LES](#) function

- [Dynamic Arrays](#)

MAXIMUM

Returns the largest numeric value from the elements of a dynamic array.

```
MAXIMUM(dynarray)
```

Arguments

<i>dynarray</i>	Any valid dynamic array of numeric values.
-----------------	--

Description

The **MAXIMUM** function compares the values of all of the elements in a dynamic array and returns the largest numeric value. The **MAXIMUM** function compares all dynamic array values, regardless of the dynamic array levels of the elements. If an element value is an uninitialized variable, a null string, or a non-numeric value, **MAXIMUM** parses its value as 0 (zero).

Examples

The following example uses the **MAXIMUM** function to return the largest numeric value in a dynamic array:

```
a=10:@FM:9:@VM:8:@SM:7  
PRINT MAXIMUM(a);      ! returns 10
```

See Also

- [ADDS](#) function
- [MINIMUM](#) function
- [SUMMATION](#) function
- [Dynamic Arrays](#)

MINIMUM

Returns the smallest numeric value from the elements of a dynamic array.

```
MINIMUM( dynarray )
```

Arguments

<i>dynarray</i>	Any valid dynamic array of numeric values.
-----------------	--

Description

The **MINIMUM** function compares the values of all of the elements in a dynamic array and returns the smallest numeric value. The **MINIMUM** function compares all dynamic array values, regardless of the dynamic array levels of the elements. If an element value is an uninitialized variable, a null string, or a non-numeric value, **MINIMUM** parses its value as 0 (zero).

Examples

The following example uses the **MINIMUM** function to return the smallest numeric value in a dynamic array:

```
a=10:@FM:9:@VM:8:@SM:7  
PRINT MINIMUM(a);      ! returns 7
```

See Also

- [ADDS](#) function
- [MAXIMUM](#) function
- [SUMMATION](#) function
- [Dynamic Arrays](#)

MOD

Modulo division of two values.

```
MOD(numstr1,numstr2)
```

Arguments

<i>numstr1</i>	The dividend. Any valid numeric or numeric string .
<i>numstr2</i>	The divisor. Any valid non-zero numeric or numeric string .

Description

The **MOD** function divides the value of *numstr1* by *numstr2*, and returns the modulo (remainder following integer division) that results from this division. If a *numstr* value is an uninitialized variable, a null string, or a non-numeric value, **MOD** parses its value as 0 (zero).

Attempting to divide by zero issues a “Division by zero” error and returns a value of 0.

The **REM** function is functionally identical to the **MOD** function. To perform exact division with a fractional product, use the division operator (/). To perform integer division, use the **DIV** function.

You can use the **MODS** function to perform modulo division on the elements of a dynamic array.

Examples

The following examples use the **MOD** function to return the modulo value for a division operation:

```
PRINT MOD(10,5);    ! returns 0
PRINT MOD(10,4);    ! returns 2
PRINT MOD(10,3);    ! returns 1
PRINT MOD(10,6);    ! returns 4
PRINT MOD(10,-6);   ! returns 4
PRINT MOD(10,11);   ! returns 10
```

See Also

- [REM](#) function
- [DIV](#) function
- [MODS](#) function
- [DIVS](#) function

- [Operators](#)

MODS

Modulo division of the values of corresponding elements in two dynamic arrays.

```
MODS(dynarray1,dynarray2)
```

Arguments

<i>dynarray</i>	Any valid dynamic array of numeric values.
-----------------	--

Description

The **MODS** function divides the value of each element in *dynarray1* by the corresponding element in *dynarray2*. It then returns a dynamic array containing the modulo (remainder) for each element that results from these divisions. If an element value is an uninitialized variable, a null string, or a non-numeric value, **MODS** parses its value as 0 (zero).

Attempting to divide by zero issues a “Division by zero” error and returns a value of 0 for that element.

You can use the **MOD** function or the **REM** function to perform modulo division on single values.

Examples

The following example uses the **MODS** function to return the modulo value for each division operation on the elements of two dynamic arrays:

```
a=11:@VM:22:@VM:0:@VM:-7
b=10:@VM:.5:@VM:10:@VM:42
PRINT MODS(a,b); ! returns 1v0v0v-7
```

See Also

- [ADDS](#) function
- [DIVS](#) function
- [MOD](#) function
- [MULS](#) function
- [REM](#) function

- [SUBS](#) function
- [Dynamic Arrays](#)

MULS

Multiplies the values of corresponding elements in two dynamic arrays.

```
MULS(dynarray1, dynarray2)
```

Arguments

<i>dynarray</i>	The <i>dynarray</i> arguments can be any dynamic array of numeric values. If a dynamic array element contains a non-numeric value, MULS treats this value as 0 (zero).
-----------------	---

Description

The **MULS** function multiplies the value of each element in *dynarray1* by the corresponding element in *dynarray2*. It then returns a dynamic array containing the results of these multiplications. If a *dynarray* element value is an uninitialized variable, a null string, or a non-numeric value, **MULS** parses its value as 0 (zero). If the two dynamic arrays are of unequal length, the elements with no corresponding value are multiplied by 0, thus returning a 0 value for that element.

Examples

The following example uses the **MULS** function to multiply the elements of two dynamic arrays:

```
a=3:@VM:22:@VM:33:@VM:4
b=10:@VM:0.5:@VM:0:@VM:-4
PRINT MULS(a,b); ! returns 30v11v0v-16
```

The following example multiplies the elements of two dynamic arrays of different length:

```
a=3:@VM:22:@VM:33:@VM:4
b=10:@VM:0.5
PRINT MULS(a,b); ! returns 30v11v0v0
```

See Also

- [SMUL](#) function

- [ADDS](#) function
- [DIVS](#) function
- [MODS](#) function
- [SUBS](#) function
- [Dynamic Arrays](#)

NEG

Returns the inverse sign of a number.

```
NEG(number)
```

Arguments

<i>number</i>	Any valid numeric expression, specified as a number or a numeric string .
---------------	---

Description

The **NEG** function returns the inverse sign of a number. For example, **NEG**(-1) returns 1, and **NEG**(1) returns -1. **NEG** removes multiple signs and leading and trailing zeros from *number*. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7. If *number* is an uninitialized variable or a non-numeric value, **NEG** returns 0 (zero).

The **NEG** function inverts the sign of a number: negative numbers become positive and positive numbers become negative. The **ABS** function gives the absolute value of a number: all numbers become positive.

You can use the **NEGS** function to invert the sign for each element of a dynamic array.

Examples

The following example uses the **NEG** function to invert the sign of a number:

```
PRINT NEG(0050.300); ! Returns -50.3
PRINT NEG(-50.3); ! Returns 50.3
PRINT NEG(++50.3); ! Returns -50.3
PRINT NEG(0); ! Returns 0
PRINT NEG(-0); ! Returns 0
```

See Also

- [ABS](#) function
- [NEGS](#) function

NES

Performs an inequality comparison on elements of two dynamic arrays.

```
NES(dynarray1,dynarray2)
```

Arguments

<i>dynarray</i>	Any valid dynamic array of numeric values.
-----------------	--

Description

The **NES** function (not equals) compares each corresponding numeric element from two dynamic arrays for inequality. It returns a dynamic array of boolean values, in which each element comparison is represented by a 1 (not equal) or a 0 (equal). **NES** removes signs and leading and trailing zeros from element values before making the comparison. If an element is an uninitialized variable, a null string, or a non-numeric value, **NES** assigns it a value of 0 for the purpose of this comparison.

For two elements to be compared, they must be on the same dynamic array level. For example, you cannot compare a value mark (@VM) dynamic array element to a subvalue mark (@SM) dynamic array element.

If an element in one dynamic array has no corresponding element in the other dynamic array, a 1 (not equal) comparison is returned for that element.

The **NES** function is the functional opposite of the **EQS** function.

Examples

The following example uses the **NES** function to return a not equals comparison for each of the elements in dynamic arrays *a* and *b*:

```
a=11:@VM:-22:@VM:-33:@VM:44
b=11:@VM:-24:@VM:0:@VM:44
PRINT NES(a,b)
! returns 0v1v1v0
```

See Also

- [EQS](#) function
- [GES](#) function
- [GTS](#) function
- [LES](#) function
- [LTS](#) function
- [Dynamic Arrays](#)

NOT

Returns the logical complement of an expression.

```
NOT(expression)
```

Arguments

<i>expression</i>	Any valid expression.
-------------------	-----------------------

Description

The **NOT** function returns the logical complement (inverse) of an expression. Thus all expressions that evaluate to 0 become 1, and all expressions that evaluate to a non-zero numeric value become 0. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7, and thus returns 0. If *expression* is an uninitialized variable, the null string, or a non-numeric value, **NOT** parses it as 0, and thus returns 1.

You can use the **ANDS** and **ORS** functions to perform logical comparisons on two values (either single expressions or arrays).

Examples

The following example uses the **NOT** function to return the logical complement of an expression:

```
PRINT NOT(7);           ! Returns 0
PRINT NOT(-7);         ! Returns 0
PRINT NOT("7dwarves"); ! Returns 0
PRINT NOT(0);          ! Returns 1
PRINT NOT("fred");     ! Returns 1
PRINT NOT("");        ! Returns 1
```

See Also

- [ANDS](#) function
- [ORS](#) function
- [NOTS](#) function
- [Operators](#)

NOTS

Returns the logical complement of each element of a dynamic array.

```
NOTS(dynarray)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
-----------------	---

Description

The **NOTS** function returns the logical complement (inverse) of each element of a dynamic array. It returns a dynamic array of boolean values corresponding to the elements of *dynarray*.

All expressions that evaluate to 0 become 1, and all expressions that evaluate to a non-zero numeric value become 0. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7, and thus returns 0. If an element contains an uninitialized variable, the null string, or a non-numeric value, **NOTS** parses it as 0, and thus returns 1 for that element.

You can use the **ANDS** and **ORS** functions to perform logical comparisons on two dynamic arrays.

Examples

The following example uses the **NOTS** function to return the logical complement of the elements of a dynamic array:

```
a=7:@VM: "-7.1" :@VM: "7dwarves" :@VM: 0 :@VM: " " :@VM: "fred"  
PRINT NOTS(a)  
! Returns 0v0v0v1v1v1
```

See Also

- [ANDS](#) function
- [ORS](#) function
- [NOT](#) function
- [Dynamic Arrays](#)
- [Operators](#)

NUM

Returns whether a value is numeric.

```
NUM(string)
```

Arguments

<i>string</i>	Any valid string or numeric.
---------------	--

Description

The **NUM** function determines whether a value is numeric or non-numeric. If *string* is a non-numeric value, **NUM** returns 0. If *string* is a numeric value, **NUM** returns 1. A numeric value can contain the numerals 0 through 9, plus and minus signs, and the decimal point. **NUM** also returns 1 for the null string, or for an undefined variable.

You can use the **NUMS** function to make the same determination of each element of a dynamic array.

See Also

- [NUMS](#) function
- [SCMP](#) function

NUMS

Returns whether each element in a dynamic array is numeric.

```
NUMS(dynarray)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
-----------------	---

Description

The **NUMS** function determines whether each element in a dynamic array is numeric or non-numeric. It returns a dynamic array of boolean values corresponding to the elements in *dynarray*. If an element contains a non-numeric value, **NUMS** returns 0 for that element. If an element contains a numeric value, **NUMS** returns 1 for that element. **NUMS** also returns 1 for the null string, or for an undefined variable.

A numeric value can be a numeric or a string numeric. A valid numeric value can contain the numbers 0 through 9, plus and minus signs, and the decimal point; it cannot contain any other characters.

You can use the **NUM** function to determine whether a single value is numeric or non-numeric.

See Also

- [NUM](#) function
- [SCMP](#) function
- [Dynamic Arrays](#)

OCONV

Converts a value from internal format to external format.

```
OCONV(istring, codes)
```

Arguments

<i>istring</i>	An expression that resolves to a string . It specifies a value represented in internal (storage) format.
<i>codes</i>	A quoted string containing one or more code characters that govern the conversion from internal format to external format.

Description

The **OCONV** function is a general-purpose conversion function used to convert from internal (storage) format to external (output) format. Conversions are governed by a set of code characters specific to the type of data to be converted. You must specify at least one mandatory code character. Additional code characters may be included or omitted, but must follow the specified order. Most valid CMQL conversion codes can be used. The following are some of the most commonly used code characters:

<p>Time conversion: Internal times are specified as the number of seconds elapsed since midnight. OCONV accepts, but truncates, fractional seconds.</p>	<p>“MT” plus one or more optional codes: “H”=12 hour clock with AM and PM suffixes. “S”=include seconds. A character to be used as the time separator character, replacing the default colons (letters are always displayed uppercase).</p>
<p>Date conversion: Internal dates are specified as the number of days elapsed since December 31, 1967. Dates prior to this are specified using a negative number of days.</p>	<p>“D” plus one or more optional codes in the following sequence: The integers 0, 2, or 4 to specify the number of year digits (default is 4). A character to be used as the date separator character (such as / or –). Dates are formatted according to the current Caché locale, so that numeric dates are in the correct date/month order: American format mm/dd/yyyy or European (E) format dd/mm/yyyy.</p>
<p>Case conversion: Converts the case of alphabetic characters; has no effect on non-letter characters.</p>	<p>“MC” plus one of the following codes: “L”=convert to lowercase. “U”=convert to uppercase. “T”=convert to title case (initial letter of each word uppercase, all other letters lowercase).</p>

The **DATE** and **TIME** functions return internal format values. The **TIMEDATE** function returns external format values.

You can use the **STATUS** function to determine the success of an **OCONV** conversion.

The **OCONV** function converts from internal format to external format. The **ICONV** function converts from external format to internal format.

You can use the **OCONVS** function to convert the elements of a dynamic array from internal format to external format.

Date Conversion

You can display a date in any of the following formats:

- Abbreviated month format (“08 AUG 2006”), with either four-digit years (code "D" or "D4") or two-digit years (code "D2").
- Numeric format ("08/08/2006"), with your choice of separator character (code "D/", "D-", etc.) and either four-digit years (code "D/" or "D4/"), two-digit years (code "D2/"), or

no year (code "D0/"). The day/month order (American or European) is determined by the current Caché locale.

- Day of the Week format (code "DW") returning an integer from 0 (Sunday) through 6 (Saturday).

You can use the [DATE](#) function to supply the current date in internal format.

Internal dates are an integer count of days, with 0 representing December 31, 1967. Dates earlier than December 31, 1967 can be represented using negative numbers. The largest permitted internal date is 2933628, which represents December 31, 9999. The smallest permitted internal date is -46385, which represents December 31, 1840.

The expansion of two-digit years to four digits is governed by the MultiValue CEN-TURY.PIVOT verb, described in *Operational Differences Between MultiValue and Caché*.

Examples

The following example shows date conversions:

```
DateConversions:
! Month Abbreviation Formats:
PRINT OCONV(0,"D");           ! "31 DEC 1967"
PRINT OCONV(14100,"D");       ! "08 AUG 2006"
PRINT OCONV(14100,"D2");      ! "08 AUG 06"
PRINT OCONV(DATE(),"D");      ! current date in above format
PRINT OCONV(@DATE,"D");       ! current date in above format
PRINT OCONV(14120,"D-");      ! "08-28-2006"
PRINT OCONV(14120,"D/");      ! "08/28/2006"
PRINT OCONV(14120,"DE");      ! "28/08/2006"
PRINT OCONV(14120,"D2/");     ! "08/28/06"
PRINT OCONV(14120,"D2-E");    ! "28-08-06"
```

The following example shows time conversions:

```
TimeConversions:
PRINT OCONV(0,"MT")
PRINT OCONV(TIME(),"MT")
PRINT OCONV(TIME(),"MTH")
PRINT OCONV(TIME(),"MTS")
PRINT OCONV(TIME(),"MTS.")
PRINT OCONV(TIME(),"MTHS*")
```

The following example shows case conversions:

```
CaseConversions:
mystr="The qUICK BrOwn foX"
PRINT OCONV(mystr,"MCU")
! Returns: THE QUICK BROWN FOX
PRINT OCONV(mystr,"MCL")
! Returns: the quick brown fox
PRINT OCONV(mystr,"MCT")
! Returns: The Quick Brown Fox
```

See Also

- [ICONV](#) function
- [OCONVS](#) function
- [STATUS](#) function
- [DATE](#) function
- [TIME](#) function
- [TIMEDATE](#) function
- [DOWNCASE](#) function
- [UPCASE](#) function
- [Strings](#)

OCONVS

Converts a dynamic array from internal format to external format.

```
OCONVS( idynarray, codes )
```

Arguments

<i>idynarray</i>	A dynamic array , each element of which specifies a value in internal (storage) format.
<i>codes</i>	A quoted string containing one or more code characters that govern the conversion from internal format to external format.

Description

The **OCONVS** function is a general-purpose conversion function used to convert the elements of a dynamic array from internal (storage) format to external (output) format. Conversions are governed by a set of code characters specific to the type of data to be converted. You must specify at least one mandatory code character. Additional code characters may be included or omitted, but must follow the specified order. The code characters are as follows:

<p>Time conversion: Internal times are specified as the number of seconds elapsed since midnight. OCONVS accepts, but truncates, fractional seconds.</p>	<p>“MT” plus one or more optional codes: “H”=12 hour clock with AM and PM suffixes. “S”=include seconds. A character to be used as the time separator character, replacing the default colons (letters are always displayed uppercase).</p>
<p>Date conversion: Internal dates are specified as the number of days elapsed since December 31, 1967. Dates prior to this are specified using a negative number of days.</p>	<p>“D” plus one or more optional codes: The integers 2 or 4 specifying the number of year digits. A character to be used as the date separator character (such as / or –), replacing the default spaces. “E”=European format (dd/mm/yyyy).</p>
<p>Case conversion: Converts the case of alphabetic characters; has no effect on non-letter characters.</p>	<p>“MC” plus one of the following codes: “L”=convert to lowercase. “U”=convert to uppercase. “T”=convert to title case (initial letter of each word uppercase, all other letters lowercase).</p>

The **DATE** and **TIME** functions return internal format values. The **TIMEDATE** function returns external format values.

The expansion of two-digit years to four digits is governed by the MultiValue **CENTURY.PIVOT** verb, described in *Operational Differences Between MultiValue and Caché*.

Note: The **OCONVS** function converts dynamic array element values from internal format to external format. The **ICONVS** function converts dynamic array element values from external format to internal format.

You can use the **OCONV** function to convert a single value from internal format to external format.

Examples

The following example shows date conversions:

```
DateConversions:
  x=14143:@VM:14144:@VM:14145
  PRINT OCONVS(x,"D")
  ! Returns "20 SEP 2006v21 SEP 2006v22 SEP 2006"
```

See Also

- [OCONV](#) function

- [ICONVS](#) function
- [DATE](#) function
- [TIME](#) function
- [TIMEDATE](#) function
- [DOWNCASE](#) function
- [UPCASE](#) function
- [Dynamic Arrays](#)

ORS

Returns the logical OR of corresponding elements of two dynamic arrays.

```
ORS(dynarray1, dynarray2)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
-----------------	---

Description

The **ORS** function performs a logical OR test on the corresponding element values of *dynarray1* and *dynarray2*. If either element value is a non-zero numeric value, **ORS** returns 1 for that element. Otherwise, **ORS** returns 0. If an element value is an uninitialized variable, a null string, or a string containing any non-numeric value, **ORS** parses its value as 0.

Caché MVBasic also supports the logical OR operators ! and OR. These can be applied to the elements of dynamic arrays by using the **REUSE** function.

Examples

The following example uses the **ORS** function to compare two dynamic arrays. It returns 1 when either element value is non-zero:

```
a=1:@VM:0:@VM:33:@VM:0
b=10:@VM:9:@VM:1:@VM:0
PRINT ORS(a,b)
! returns 1v1v1v0
```

See Also

- [ANDS](#) function
- [NOTS](#) function
- [Dynamic Arrays](#)
- [Operators](#)

PWR

Returns a number raised to a power.

```
PWR ( num , exponent )
```

Arguments

<i>num</i>	The base number. Any valid numeric expression, specified either as a numeric or as a string.
<i>exponent</i>	The exponent. Any valid numeric expression, specified either as a numeric or as a string.

Description

The **PWR** function raises *num* to the power specified by *exponent*. Both numeric values can be expressed as either numbers or as strings. Leading plus signs and leading and trailing zeros are ignored. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7. Non-numeric strings and null strings are parsed as 0. Any *num* raised to an *exponent* of 0 returns 1.

The same operation can be performed using the exponentiation operator: **.

See Also

- [SQRT](#) function
- [SCMP](#) function
- [SADD](#) function
- [SDIV](#) function
- [Operators](#)

QUOTE

Encloses a value in double quotation marks.

```
QUOTE(string)
```

Arguments

<i>string</i>	Any expression that resolves to a string or a numeric. <i>string</i> may be a dynamic array .
---------------	---

Description

The **QUOTE** function returns *string* enclosed in double quotation marks. Using **QUOTE** increases the length of *string* by 2 characters. If *string* is the null string ("") or an undefined variable, **QUOTE** returns a string consisting of two quotation mark characters, a string with a length of 2. This should not be confused with the null string (""), which has a length of 0.

The **QUOTE** function converts a numeric to canonical form before enclosing it in quotation marks. **QUOTE** does not convert a numeric string to canonical form.

The **DQUOTE** function is functionally identical to **QUOTE**. The **SQUOTE** function is similar, except that it encloses *string* with single quotation marks, rather than double quotation marks.

Examples

The following example uses the **QUOTE** function to convert a numeric to a string enclosed in double quotation marks:

```
quoted = QUOTE(+007.000)
PRINT quoted;           ! Returns "7"
PRINT LEN(quoted);     ! Returns 3
```

See Also

- [DQUOTE](#) function
- [SQUOTE](#) function
- [LEN](#) function
- [PRINT](#) statement

RAISE

Raises dynamic array delimiters to next level.

```
RAISE(dynarray)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
-----------------	---

Description

The **RAISE** function returns a dynamic array with its delimiters converted to the next higher-level delimiters. For example, @SM subvalue mark delimiters become @VM value mark delimiters. When a delimiter cannot be raised any further, it is returned unchanged.

The available levels, in ascending order, are: CHAR(250); @TM (CHAR(251)); @SM (CHAR(252)); @VM (CHAR(253)); @FM (CHAR(254)); and @IM (CHAR(255)).

The **LOWER** function performs the opposite operation, lowering the level of dynamic array delimiters to the next lower level.

Examples

The following example uses the **RAISE** function to convert dynamic array delimiters to the next higher level. It then uses the **LOWER** function to reverse this operation:

```
numsm=123:@SM:456:@SM:789:@SM:"10":@SM:"11"
PRINT numsm;           ! Returns 123s456s10s11
numraise = RAISE(numsm)
PRINT numraise;       ! Returns 123v456v10v11
numlower = LOWER(numraise)
PRINT numlower;      ! Returns 123s456s10s11
```

See Also

- [LOWER](#) function
- [Dynamic Arrays](#)

REM

Remainder after integer division of two values.

```
REM(numstr1,numstr2)
```

Arguments

<i>numstr1</i>	The dividend. Any valid numeric or numeric string .
<i>numstr2</i>	The divisor. Any valid non-zero numeric or numeric string .

Description

The **REM** function divides the value of *numstr1* by *numstr2*, and returns the remainder following integer division (modulo) that results from this division. If a *numstr* value is an uninitialized variable, a null string, or a non-numeric value, **REM** parses its value as 0 (zero).

Attempting to divide by zero issues a “Division by zero” error and returns a value of 0.

The **MOD** function is functionally identical to the **REM** function. You can use the **MODS** function to perform modulo division on the elements of a dynamic array.

Note: Caché MVBasic contains both a REM (remarks) statement and a REM (remainder) function. These are completely unrelated and should not be confused.

Examples

The following examples use the **REM** function to return the remainder value for an integer division operation:

```
PRINT REM(10,5);    ! returns 0
PRINT REM(10,4);    ! returns 2
PRINT REM(10,3);    ! returns 1
PRINT REM(10,6);    ! returns 4
PRINT REM(10,-6);   ! returns 4
PRINT REM(10,11);   ! returns 10
```

See Also

- [DIVS](#) function
- [MOD](#) function
- [MODS](#) function

REMOVE

Extracts sequential elements of a dynamic array.

```
REMOVE (dynarray, delimcode)
```

Arguments

<i>dynarray</i>	A dynamic array from which successive data values are to be extracted.
<i>delimcode</i>	A variable used to receive an integer code for the dynamic array delimiter type.

Description

The **REMOVE** function efficiently extracts successive data values from a dynamic array. The extracted element value is returned. The delimiter type is placed in the *delimcode* variable. The **REMOVE** function operates on all dynamic array delimiter levels; in contrast, the **REMOVE** statement operates on a specified delimiter level.

REMOVE maintains an internal pointer so that repeated calls return successive element values. If **REMOVE** is called after the last element value has been extracted, it returns the empty string.

You can use the **GETREM** function to return the character position in *dynarray* of the **REMOVE** pointer.

Note: The **REMOVE** function, **REMOVE** statement, and **REVREMOVE** statement all share the same character position pointer. It is incremented by Remove operations and decremented by Revremove operations.

The *delimcode* integer code values are as follows:

0	End of file
1	@IM Item Mark CHAR(255)
2	@FM Field Mark CHAR(254)
3	@VM Value Mark CHAR(253)
4	@SM Subvalue Mark CHAR(252)
5	@TM Text Mark CHAR(251)

Examples

The following example successively extracts the first five elements from a dynamic array:

```
names="Fred":@VM:"Barney":@VM:"Wilma":@VM:"Betty"
FOR x=1 TO 5
  PRINT REMOVE(names,lv1)
  PRINT lv1
  ! Returns:
  !   Fred
  !   3
  !   Barney
  !   3
  !   Wilma
  !   3
  !   Betty
  !   0
  !   ""
  !   0
NEXT
```

See Also

- [REVREMOVE](#) statement
- [EXTRACT](#) function
- [GETREM](#) function
- [REMOVE](#) function

REPLACE

Replaces the data in an element of a dynamic array.

```
REPLACE (dynarray, f[, v[, s]] ; replacement )
```

Arguments

<i>dynarray</i>	Any valid dynamic array.
<i>f</i>	An integer specifying the Field level of the dynamic array from which to access the data. Fields are counted from 1.
<i>v</i>	<i>Optional</i> — An integer specifying the Value level of the dynamic array from which to access the data. Values are counted from 1 within a Field.
<i>s</i>	<i>Optional</i> — An integer specifying the Subvalue level of the dynamic array from which to access the data. Subvalues are counted from 1 within a Value.
<i>replacement</i>	A data value used to replace the element data value specified by <i>f</i> , <i>v</i> , and <i>s</i> . Note the semicolon (;) that precedes <i>replacement</i> .

Description

The **REPLACE** function replaces the data value in one element of a dynamic array with a new value. Which element to replace is specified by the *f*, *v*, and *s* integers. For example, if *f*=2 and *v*=3, this means replace the third value from the second field. If *f*=2 and *v* is not specified, this means to replace the entire second field.

If *f*, *v*, or *s* is higher than the current number of elements at that location, **REPLACE** appends the *replacement* value with the appropriate number of level delimiter characters.

If *replacement* is the null string, **REPLACE** removes the current data value, but does not remove the level delimiter character.

If lower level delimiters exist in *dynarray*, setting an upper level to 0, the null string, a non-numeric value, or an undefined variable is equivalent to setting it to 1.

If lower level delimiters do not exist in *dynarray*, setting this non-existent lower level to 1, 0, the null string, a non-numeric value, or an undefined variable has no effect on the data value in the level above it.

You can also use the <> operator to replace an element value in a dynamic array. For further details, see the [Dynamic Arrays](#) page of this manual.

Examples

The following example uses the **REPLACE** function to replace the second value from the first field of a dynamic array:

```
cities="New York":@VM:"London":@VM:
"Chicago":@VM:"Boston":@VM:"Los Angeles"
PRINT REPLACE(cities,1,2;"Minneapolis")
```

See Also

- [REMOVE](#) statement
- [EXTRACT](#) function
- [Dynamic Arrays](#)

REUSE

Reuses a value when comparing two dynamic arrays of different lengths.

```
REUSE (dynarray)
```

Arguments

<i>dynarray</i>	Any valid dynamic array . This argument can be a dynamic array of one element—a string or numeric expression.
-----------------	---

Description

The **REUSE** function is used in combination with MVBasic functions that compare the elements of two dynamic arrays. Its most common use is to provide a corresponding element value when comparing dynamic arrays of different lengths. **REUSE** provides the needed element values for the shorter of the two dynamic arrays by reusing the last element value as the value for all subsequent element comparisons.

Specifying **REUSE** has no effect when the two dynamic arrays are of the same size, or if **REUSE** is specified for the larger of the two dynamic arrays.

If *dynarray* is set to a literal, it is treated as a dynamic array with one element. In other words, the literal is compared to every element in the other dynamic array.

If **REUSE** is not used when comparing dynamic arrays of different lengths, a value is provided for the elements without a match. In most cases these elements are compared with either the

null string (for string comparisons) or with 0 (for numeric comparisons). Note however that the **DIVS** function supplies a value of 1 for missing divisor elements to prevent division by zero errors.

Examples

The following example gives the shipping weight of various items. The items (widget) vary in weight, but the packaging (box) is always the same weight:

```
widget=4:@VM:3:@VM:4.5:@VM:2.5:@VM:5:@VM:4:@VM:3
box=1.3
shipwt=ADDS(widget,REUSE(box))
PRINT shipwt
! Returns 5.3v4.3v5.8v3.8v6.3v5.3v4.3
```

The following example concatenates the string value elements of two dynamic arrays. In this case, the `qrtrs` dynamic array is static; it always has four values, while the `qpaid` dynamic array grows as quarterly payments are posted. By making its last element value “unpaid”, the resulting `paidstatus` dynamic array always has a payment status for each quarter:

```
qrtrs="Q1-":@VM:"Q2-":@VM:"Q3-":@VM:"Q4-"
qpaid="$100":@VM:"$150":@VM:"unpaid"
paidstatus = CATS(qrtrs,REUSE(qpaid))
PRINT paidstatus
! returns Q1-$100vQ2-$150vQ3-unpaidvQ4-unpaid
```

The following example uses **REUSE** to calculate bonuses based on salary. The policy of this organization is to give its three highest-paid employees (the partners) a bonus of 1.5% of salary, and all other employees a bonus of 2% of salary:

```
BonusPct=1.5:@VM:1.5:@VM:1.5:@VM:2
SalInThou=160:@VM:150:@VM:150:@VM:105:@VM:100:@VM:95:@VM:70:@VM:65
BonusAmt=MULS(SalInThou,REUSE(BonusPct))
```

See Also

- [ADDS](#) function
- [CATS](#) function
- [DIVS](#) function
- [MODS](#) function
- [MULS](#) function
- [SUM](#) function
- [SUMMATION](#) function
- [SUBS](#) function

- [Dynamic Arrays](#)

RIGHT

Returns a specified number of characters from the right end of a string.

```
RIGHT(string, length)
```

Arguments

<i>string</i>	String expression from which the rightmost characters are returned.
<i>length</i>	Numeric expression that evaluates to a positive integer indicating how many characters to return. If 0, a zero-length string ("") is returned. Fractional numbers are truncated to an integer. If greater than or equal to the number of characters in string, the entire string is returned. No padding is performed.

Description

The **RIGHT** function returns the specified number of characters counting backwards from the end (right end) of a string. If you specify a *length* greater than the string length, the entire string is returned. To determine the number of characters in string, use the **LEN** function.

The **LEFT** function returns the specified number of characters from the beginning (left end) of a string.

Examples

The following example uses the **RIGHT** function to return a specified number of characters from the right side of a string:

```
AnyString = "Hello World"  
PRINT RIGHT(AnyString,1);      ! Returns "d"  
PRINT RIGHT(AnyString,5);     ! Returns "World"  
PRINT RIGHT(AnyString,20);    ! Returns "Hello World"
```

See Also

- [LEFT](#) function
- [LEN](#) function

RND

Returns a random number.

```
RND ( number )
```

Arguments

<i>number</i>	Any valid numeric expression, specified as a number or a numeric string.
---------------	--

Description

The **RND** function returns a random value between zero and the specified *number*, inclusive of zero but exclusive of *number*. Thus the available range of returned numbers is 0 through *number*-1.

If *number* is a decimal number it is truncated to its integer portion. If *number* is a negative number, a negative number is returned.

A *number* string value is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7. If *number* resolves to 1, 0, or -1, **RND** always returns 0. If *number* is a non-numeric string, the empty string (“”), or an undefined variable it is parsed as 0, and thus **RND** always returns 0.

Examples

The following example generates twenty random numbers between 0 and 100:

```
FOR x=1 TO 20
  PRINT RND(100)
NEXT
```

See Also

- [RANDOMIZE](#) statement

SADD

Adds two numeric strings.

```
SADD(numstr1,numstr2)
```

Arguments

<i>numstr</i>	Any valid numeric or numeric string .
---------------	---

Description

The **SADD** function adds two numeric values, expressed as either numbers or as strings, and returns the result. Leading plus signs and leading and trailing zeros are ignored. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7. Non-numeric strings and null strings are parsed as 0.

Arithmetic Operations

- To perform arithmetic operations on numeric strings, use the **SADD**, **SSUB**, **SMUL**, and **SDIV** functions.
- To perform arithmetic operations on floating point numbers, use the **FADD**, **FSUB**, **FMUL**, and **FDIV** functions, or use the standard arithmetic operators.
- To perform integer division, use the **DIV** function. To perform modulo division, use the **MOD** function.
- To perform arithmetic operations on corresponding elements of dynamic arrays, use the **ADDS**, **SUBS**, **MULS**, **DIVS**, and **MODS** functions.
- To add together the element values within a single dynamic array, use either the **SUM** function (for single-level dynamic arrays) or the **SUMMATION** function (for multi-level dynamic arrays).
- To perform numeric comparison operations, use the **SCMP** function, or use the standard comparison operators.

Examples

The following examples use the **SADD** function to add two numeric strings. All of these examples return 10:

```
PRINT SADD(7,3)
PRINT SADD("7","3")
PRINT SADD("+7.00","003")
PRINT SADD("7dwarves","3wishes")
```

All of the following examples return 7:

```
PRINT SADD(7,0)
PRINT SADD("7","")
PRINT SADD("7","three")
```

See Also

- [FADD](#) function
- [ADDS](#) function
- [SUM](#) function
- [SUMMATION](#) function
- [Operators](#)

SCMP

Performs a string comparison of two numbers.

```
SCMP(num1, num2)
```

Arguments

<i>num1</i>	Any valid numeric expression, specified as a number or a numeric string.
<i>num2</i>	Any valid numeric expression, specified as a number or a numeric string.

Description

The **SCMP** function compares two numeric values, expressed as either numbers or as strings. Leading plus signs and leading and trailing zeros are ignored. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7. Non-numeric strings and null strings are parsed as 0.

The comparison return values are as follows:

- -1: $num1 < num2$
- 0: $num1 = num2$

- 1: *num1* > *num2*

See Also

- [SADD](#) function
- [SDIV](#) function
- [Operators](#)

SDIV

Divides two numeric strings.

```
SDIV(numstr1,numstr2)
```

Arguments

<i>numstr1</i>	The dividend. Any valid numeric or numeric string .
<i>numstr2</i>	The divisor. Any valid numeric or numeric string .

Description

The **SDIV** function divides *numstr1* by *numstr2* and returns the result. The two numeric values can be expressed as either numbers or as strings. Leading plus signs and leading and trailing zeros are ignored. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7. Non-numeric strings and null strings are parsed as 0. Division by 0 is illegal, and generates a syntax error.

For compatibility, a third numeric argument is accepted, but ignored.

Arithmetic Operations

- To perform arithmetic operations on numeric strings, use the [SADD](#), [SSUB](#), [SMUL](#), and **SDIV** functions.
- To perform arithmetic operations on floating point numbers, use the [FADD](#), [FSUB](#), [FMUL](#), and [FDIV](#) functions, or use the standard arithmetic operators.
- To perform integer division, use the [DIV](#) function. To perform modulo division, use the [MOD](#) function.

- To perform arithmetic operations on corresponding elements of dynamic arrays, use the [ADDS](#), [SUBS](#), [MULS](#), [DIVS](#), and [MODS](#) functions.
- To perform numeric comparison operations, use the [SCMP](#) function, or use the standard comparison operators.

Examples

The following examples use the **SDIV** function to divide a numeric string by another numeric string. All of these examples return 2.333333333:

```
PRINT SDIV(7,3)
PRINT SDIV("7","3")
PRINT SDIV("+7.00","003")
PRINT SDIV("7dwarves","3wishes")
```

All of the following examples return 0:

```
PRINT SDIV(0,7)
PRINT SDIV("", "0")
PRINT SDIV("seven", "3")
```

All of the following examples generate a syntax error:

```
PRINT SDIV(7,0)
PRINT SDIV("7","")
PRINT SDIV("7","three")
```

See Also

- [FDIV](#) function
- [DIVS](#) function
- [DIV](#) function
- [MOD](#) function
- [MODS](#)
- [Operators](#)

SEQ

Returns the character code corresponding to a specified character.

```
SEQ(char)
```

Arguments

<i>char</i>	Any valid character. If <i>char</i> is a string, SEQ returns the value of the first character.
-------------	---

Description

The **SEQ** function takes a character and returns the corresponding character code, an integer value. Its inverse, the **CHAR** function takes a numeric code and returns the corresponding character.

The Caché MVBasic **SEQ** function returns the numeric value for a single character. The corresponding Caché ObjectScript **\$ASCII** function can take a string of characters and return the numeric value for a specific character by specifying its position in the string.

Note: **SEQ** and **UNISEQ** are functionally identical.

Examples

The following example uses the **SEQ** function to return the numeric code associated with the specified character:

```
PRINT SEQ('A');      ! Returns 65.  
PRINT SEQ('a');      ! Returns 97.  
PRINT SEQ('%');      ! Returns 37.  
PRINT SEQ('>');      ! Returns 62.
```

The following example uses the **SEQ** function to return lowercase letter characters and associated numeric codes of the Russian alphabet. On a Unicode version of Caché it returns the Russian letters; on an 8-bit version of Caché it returns a -1 (indicating a null string) for each letter:

```
letter=1072  
FOR x=1 TO 32  
  glyph=CHAR(letter)  
  PRINT SEQ(glyph),glyph  
  letter=letter+1  
NEXT
```

See Also

- [CHAR](#) function
- [UNISEQ](#) function
- ObjectScript: \$ASCII function

SEQS

Returns the character code for the first character of each element in a dynamic array.

```
SEQS(dynarray)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
-----------------	---

Description

The **SEQS** function takes a dynamic array and returns the corresponding numeric codes for the first character in each element. It returns these character codes as a dynamic array. If an element consists of a string of more than one character, **SEQS** returns the numeric value of the first character of that element. If an element contains an uninitialized variable or a null string, **SEQS** returns -1 for that element.

If the first character of a dynamic array element is one of the following dynamic array level delimiters: CHAR(252), CHAR(253), or CHAR(254), **SEQS** treats this character as a level delimiter, and returns -1 for the null element(s) established by parsing this character as a level delimiter.

Note: **UNISEQS** and **SEQS** are functionally identical. On Unicode systems both can be used to return character codes for 16-bit Unicode characters. On 8-bit systems, these functions return that character code of the first 8 bits of a 16-bit Unicode character.

The **CHARS** function is the inverse of **SEQS**. It takes a dynamic array of numeric codes and returns the corresponding characters.

The **SEQ** function (or **UNISEQ** function) takes the first character of a string and returns the corresponding numeric code.

The **SEQS** function returns the numeric value for the first character of each element as a dynamic array element. The corresponding Caché ObjectScript **\$ASCII** function can take a string of characters and return the numeric value for a specific character by specifying its position in the string.

Examples

The following example uses the **SEQS** function to return the numeric codes associated with each character in a dynamic array:

```
alpha="A":@VM:"B":@VM:"C":@VM:"D"  
PRINT SEQS(alpha)  
! returns 65v66v67v68
```

The following example returns the numeric codes associated with four lowercase Russian letters in a dynamic array. On a Unicode system, it returns the Russian character codes. On an 8-bit system, characters beyond 255 are treated as null strings, so **SEQS** returns -1 for each element.

```
russian=CHAR(1072):@VM:CHAR(1073):@VM:CHAR(1074):@VM:CHAR(1075)  
PRINT SEQS(russian)
```

See Also

- [UNISEQS](#) function
- [CHARS](#) function
- [SEQ](#) function
- [Dynamic Arrays](#)
- ObjectScript: [\\$ASCII](#) function

SIN

Returns the sine of an angle.

```
SIN(number)
```

Arguments

<i>number</i>	Any valid numeric expression, that expresses an angle in degrees.
---------------	---

Description

The **SIN** function takes an angle in degrees and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the hypotenuse. The result is in radians, in the range -1 to 1.

To convert degrees to radians, multiply degrees by $\pi/180$. To convert radians to degrees, multiply radians by $180/\pi$.

Examples

The following example uses the **SIN** function to return the sine of an angle:

```
DIM MyAngle
MyAngle = 1.3;           ! Define angle in degrees.
PRINT SIN(MyAngle);    ! Return sine in radians.
```

The following example uses the **SIN** function to return the cosecant of an angle:

```
DIM MyAngle, MyCosecant
MyAngle = 1.3;           ! Define angle in degrees.
MyCosecant = 1 / SIN(MyAngle); ! Calculate cosecant.
PRINT MyCosecant
```

See Also

- [ATAN](#) function
- [COS](#) function
- [SINH](#) function
- [TAN](#) function
- [Derived Math Functions](#)
- [ObjectScript: \\$ZSIN](#) function

SINH

Returns the hyperbolic sine of an angle.

```
SINH(number)
```

Arguments

<i>number</i>	Any valid numeric expression, that expresses an angle in degrees.
---------------	---

Description

The **SINH** function takes an angle in degrees and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the hypotenuse. The result is in radians, in the range -1 to 1.

To convert degrees to radians, multiply degrees by $\pi/180$. To convert radians to degrees, multiply radians by $180/\pi$.

Examples

The following example uses the **SINH** function to return the sine of an angle:

```
DIM MyAngle
MyAngle = 1.3;           ! Define angle in degrees.
PRINT SINH(MyAngle);    ! Return hyperbolc sine in radians.
```

See Also

- [ATAN](#) function
- [COS](#) function
- [SIN](#) function
- [TAN](#) function
- [Derived Math Functions](#)

SMUL

Multiplies two numeric strings.

```
SMUL(numstr1,numstr2)
```

Arguments

<i>numstr</i>	Any valid numeric or numeric string .
---------------	---

Description

The **SMUL** function multiplies the value of two numeric strings and returns a numeric value. If a *numstr* value is an uninitialized variable, a null string, or a non-numeric value, **SMUL** parses its value as 0 (zero).

Arithmetic Operations

- To perform arithmetic operations on numeric strings, use the [SADD](#), [SSUB](#), [SMUL](#), and [SDIV](#) functions.
- To perform arithmetic operations on floating point numbers, use the [FADD](#), [FSUB](#), [FMUL](#), and [FDIV](#) functions, or use the standard arithmetic operators.
- To perform integer division, use the [DIV](#) function. To perform modulo division, use the [MOD](#) function.
- To perform arithmetic operations on corresponding elements of dynamic arrays, use the [ADDS](#), [SUBS](#), [MULS](#), [DIVS](#), and [MODS](#) functions.
- To perform numeric comparison operations, use the [SCMP](#) function, or use the standard comparison operators.

Examples

The following examples use the [SMUL](#) function to multiply two numeric strings. All of these examples return 21:

```
PRINT SMUL(3,7)
PRINT SMUL("3","7")
PRINT SMUL("003","+7.00")
PRINT SMUL("3wishes","7dwarves")
```

All of the following examples return 0:

```
PRINT SMUL(3,0)
PRINT SMUL("3","")
PRINT SMUL("3","seven")
```

See Also

- [FMUL](#) function
- [MULS](#) function
- [Operators](#)

SPACE

Returns a string consisting of the specified number of spaces.

```
SPACE ( number )
```

Arguments

<i>number</i>	The number of spaces you want in the string. Any valid numeric expression, specified as a number or a numeric string.
---------------	---

Description

The **SPACE** function returns a string of the specified number of spaces.

If *number* is 0, a negative number, a null string, or a non-numeric string, no spaces are returned. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7. If *number* is a decimal number, MVBasic truncates it to an integer.

You can use the **SPACES** function to return a dynamic array, each element of which contains the number of spaces specified for that element.

You can also insert spaces using tabbing. MVBasic sets default tab stops at 10-column intervals; this default is modifiable using the **TABSTOP** statement.

Examples

The following example uses the **SPACE** function to return a string with four spaces inserted in it:

```
PRINT "Hello":SPACE(4):"World"
```

See Also

- [LEN](#) function
- [SPACES](#) function
- [PRINT](#) statement

SPACES

Returns a dynamic array consisting of the specified number of spaces for each element.

```
SPACES (dynarray)
```

Arguments

<i>dynarray</i>	A valid dynamic array of positive integers, specifying the number of spaces you want in each corresponding element of the output array.
-----------------	---

Description

The **SPACES** function returns a dynamic array, each element of which contains the number of spaces specified for that element.

If an element value is 0, a negative number, a null string or a non-numeric string, no spaces are returned. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7. If an element value is a decimal number, **SPACES** truncates it to an integer.

You can use the **SPACE** function to return a single string of spaces. You can also insert spaces using tabbing. MVBasic sets default tab stops at 10-column intervals.

Examples

The following example uses the **SPACES** function to return a dynamic array, each element of which contains one additional space. It concatenates the string of spaces in each of these elements to a single-letter string from the *letters* dynamic array:

```
letters="A":@VM:"B":@VM:"C":@VM:"D":@VM:"E"
spaces=1:@VM:2:@VM:3:@VM:4:@VM:5
PRINT CATS(letters,SPACES(spaces))
```

See Also

- [CATS](#) function
- [LENS](#) function
- [SPACE](#) function
- [PRINT](#) statement
- [Dynamic Arrays](#)

SPLICE

Combines two dynamic arrays into a new dynamic array.

```
SPLICE(dynarray1,separator,dynarray2)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
<i>separator</i>	A character or string of characters inserted when concatenating two elements from different dynamic arrays. If you specify a null string, no separator is inserted between concatenated elements.

Description

The **SPLICE** function concatenates two dynamic arrays on an element-by-element basis. It returns a dynamic array containing all of the element values of *dynarray1* and *dynarray2*.

For two elements to be concatenated, they must be on the same dynamic array level. For example, you cannot concatenate a value mark (@VM) dynamic array element to a subvalue mark (@SM) dynamic array element.

Examples

The following example uses **SPLICE** to return a concatenated dynamic array including all of the elements in dynamic arrays *a* and *b*:

```
a=10:@VM:20:@VM:30:@VM:40
b=15:@VM:25:@VM:35:@VM:45
PRINT SPLICE(a,'/',b)
! returns 10/15v20/25v30/35v40/45
```

The following example uses **SPLICE** to concatenate a dynamic array with itself, specifying a null string *separator*:

```
a=10:@VM:20:@VM:30:@VM:40
PRINT SPLICE(a,'',a)
! returns 1010v2020v3030v4040
```

The following example uses **SPLICE** to concatenate two dynamic arrays with different delimited levels:

```
a=10:@VM:20:@VM:30:@VM:40
c=11:@SM:12:@SM:13:@SM:14
PRINT SPLICE(a,'/',c)
! returns 10/11s/12s/13s/14v20v30v40
```

See Also

- [CATS](#) function
- [REUSE](#) function
- [Dynamic Arrays](#)

SPOOLER

Returns information on queued print jobs.

```
SPOOLER(n[,user])
```

Arguments

<i>n</i>	An integer flag specifying what category of information to return. Available values are 1 through 5, inclusive.
<i>user</i>	<i>Optional</i> — Limits information returned to jobs created by the specified <i>user</i> . This is the same value as the “user name” value. Applicable to <i>n</i> =2 and <i>n</i> =4 only. The default is to return information on all jobs, regardless of the creator.

Description

The SPOOLER function returns information about form queues, jobs, and assignments. It returns a dynamic array in which print jobs are separated by Field Marks, and information items for each print job are separated by Value Marks. Which jobs (Fields) are returned depends on the value of *user*. The type of information (Values) returned for each job depends on the *n* flag value. The following *n* values are supported:

1	Form queue information, consisting of the following elements: 1=form queue name, 3=device name, 5=status, 6=number of jobs, 7=page skip.
2	Print job information, consisting of the following elements: 1=form queue name, 2=print job number, 3=user name (OS login name), 4=port number of creator of job, 5=creation date (internal format), 6=creation time (internal format), 7=job status, 8=options (in legacy format), 9=print job size (in pages), 10=number of copies, 14=user name (same as 3), 15=user name (same as 3), 17=MV account name, 18=page size (in lines), 19=options (in long format).
3	Current assignments, consisting of the following elements: 1=channel number (0 to 255), 2=form queue name, 3=options (in legacy format), 4=number of copies, 5=options (in long format).
4	Current jobs, consisting of the following elements: 1=report channel number, 2=print job number, 3=print job size (in pages), 4=creation date (internal format), 5=creation time (internal format), 6=job status, 7=user name (OS login name), 8=user name (OS login name), 9=MV account name,
5	New Caché values, consisting of the following element: 1=name of Caché global for spooler. Default is ^SPOOL.

You can use the [OCONV](#) function to convert dates and times from internal to display format.

Examples

The following example illustrate the use of the **SPOOLER 2** function:

```
PRINT ON 1 "The quick brown fox"
PRINT SPOOLER(2)
```

returns:

```
STANDARDv1vFredv5948v14213v54958vCLOSEDv1v1vvvFredvFredvvUSERv4v
```

which contains the following elements:

```
1 form queue name=STANDARD
2 print job number=1
3 user name (OS login name)=Fred
4 port number of creator of job=5948
5 creation date (internal format)=14213 (29 NOV 2006)
6 creation time (internal format)=54958 (03:15:58PM)
7 job status=CLOSED
8 options (in legacy format) [none]
9 print job size (in pages)=1
10 number of copies=1
14 user name (same as 3)=Fred
15 user name (same as 3)=Fred
17=MV account name=USER
18=page size (in lines)=4
19=options (in long format).[none]
```

The following example illustrate the use of the **SPOOLER** 5 function:

```
PRINT SPOOLER(5)
```

returns:

```
^SPOOL
```

See Also

- [PRINT](#) statement
- [PRINTER](#) statement
- “Spooling” in Operational Differences between MultiValue and Caché

SQRT

Returns the square root of a number.

```
SQRT ( number )
```

Arguments

<i>number</i>	Any valid numeric expression greater than or equal to 0, specified as a number or a numeric string.
---------------	---

Description

The **SQRT** function returns the square root of *number*. This numeric value can be expressed as either a number or as a string. Leading plus signs and leading and trailing zeros are ignored. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7. Non-numeric strings and null strings are parsed as 0. The square root of 0 is 0.

You cannot return the square root of a negative number. Attempted to do so results in a runtime error.

Examples

The following example uses the **SQRT** function to calculate the square roots of the integers 0 through 16:

```
FOR x = 0 TO 16
PRINT "Square root of ":x:" = ":Sqrt(x)
NEXT
```

The following example uses the **SQRT** function to calculate the square root of pi:

```
pi = 4 * ATAN(1)
PRINT "Square root of pi = ":SQRT(pi)
```

See Also

- [PWR](#) function

SQUOTE

Encloses a value in single quotation marks.

```
SQUOTE(string)
```

Arguments

<i>string</i>	Any expression that resolves to a string or a numeric. <i>string</i> may be a dynamic array .
---------------	---

Description

The **SQUOTE** function returns *string* enclosed in single quotation marks. Using **SQUOTE** increases the length of *string* by 2 characters. If *string* is the null string or an undefined variable, **SQUOTE** returns a string consisting of two single quotation mark characters, a string with a length of 2. This should not be confused with the null string, which has a length of 0.

The **SQUOTE** function converts a numeric to canonical form before enclosing it in quotation marks. **SQUOTE** does not convert a numeric string to canonical form.

The **SQUOTE** function encloses *string* with single quotation marks. The similar **QUOTE** and **DQUOTE** functions enclose *string* with double quotation marks.

Examples

The following example uses the **SQUOTE** function to convert a numeric to a string enclosed in single quotation marks:

```
quoted = SQUOTE(+007.000)
PRINT quoted;           ! Returns '7'
PRINT LEN(quoted);     ! Returns 3
```

See Also

- [QUOTE](#) function
- [DQUOTE](#) function
- [LEN](#) function
- [PRINT](#) statement

SSUB

Subtracts two numeric strings.

```
SSUB(numstr1,numstr2)
```

Arguments

<i>numstr1</i>	The minuend. Any valid numeric or numeric string .
<i>numstr2</i>	The subtrahend. Any valid numeric or numeric string .

Description

The **SSUB** function subtracts *numstr2* from *numstr1*, expressed as either numbers or as strings, and returns the result. Leading plus signs and leading and trailing zeros are ignored. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7. Non-numeric strings and null strings are parsed as 0.

Arithmetic Operations

- To perform arithmetic operations on numeric strings, use the [SADD](#), **SSUB**, [SMUL](#), and [SDIV](#) functions.
- To perform arithmetic operations on floating point numbers, use the [FADD](#), [FSUB](#), [FMUL](#), and [FDIV](#) functions, or use the standard arithmetic operators.
- To perform integer division, use the [DIV](#) function. To perform modulo division, use the [MOD](#) function.

- To perform arithmetic operations on corresponding elements of dynamic arrays, use the [ADDS](#), [SUBS](#), [MULS](#), [DIVS](#), and [MODS](#) functions.
- To perform numeric comparison operations, use the [SCMP](#) function, or use the standard comparison operators.

Examples

The following examples use the **SSUB** function to subtract two numeric strings. All of these examples return 4:

```
PRINT SSUB(7,3)
PRINT SSUB("7","3")
PRINT SSUB("+7.00","003")
PRINT SSUB("7dwarves","3wishes")
```

All of the following examples return 7:

```
PRINT SSUB(7,0)
PRINT SSUB("7","")
PRINT SSUB("7","three")
```

See Also

- [FSUB](#) function
- [SUBS](#) function
- [Operators](#)

STATUS

Returns the status of the most recent operation.

```
STATUS( )
```

Arguments

None. The parentheses are mandatory.

Description

The **STATUS** function returns an integer code indicating the success or failure of many MVBASIC operations. It returns 0 for successful completion, and a non-zero integer to indicate the type of error encountered.

A **STATUS** value is returned for the following operations. Please see the individual command or function for the applicable status code values: BSCAN, [CLOSESEQ](#), DELETE, FILEINFO(), FILELOCK, [FMT\(\)](#), GET, GETX, [ICONV\(\)](#), INPUT @, MATWRITE, [OCONV\(\)](#), OPEN, OPENCHECK, OPENPATH, [OPENSEQ](#), READ, READBLK, READL, [READSEQ](#), READT, READU, READVL, READVU, REWIND, SETLOCALE(), WEOF, WRITEBLK, WRITESEQ, WRITESEQF, WRITET, WRITEU, WRITEV, and WRITEVU.

STR

Repeats a string value.

```
STR(string, repeats)
```

Arguments

<i>string</i>	Any valid string or number.
<i>repeats</i>	A positive integer specifying the number of repeats.

Description

The **STR** function replicates and concatenates a string multiple times. The number of repetitions is specified by the *repeats* argument. The *repeats* argument specifies the number of repeats as a positive integer. If *repeats* is a decimal number, it is truncated to an integer. The *repeats* string is parsed as an integer until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7. If *repeats* is 0, a negative number, or a non-numeric string, **STR** returns a null string.

You can use the **STRS** function to perform the same operation on all of the elements of a dynamic array.

Examples

The following example uses the **STR** function to repeat a string:

```
PRINT STR("test",5)
```

It returns: testtesttesttesttest

See Also

- [LEN](#) function

- [STRS](#) function

STRS

Repeats the string value of each element of a dynamic array.

```
STRS(dynarray, repeats)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
<i>repeats</i>	A positive integer specifying the number of repetitions of the current value of each element.

Description

The **STRS** function replicates each element of a dynamic array the number of times specified by *repeats*. In the returned dynamic array the value of each element of *dynarray* is replicated and concatenated the same number of times. The number of replications is specified by the *repeats* argument.

The *repeats* argument specifies the number of repeats as a positive integer. If *repeats* is a decimal number, it is truncated to an integer. The *repeats* string is parsed as an integer until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7. If *repeats* is 0, a negative number, or a non-numeric string, **STRS** returns a null string for all elements.

You can use the **STR** function to perform the same operation on a single value.

Examples

The following example uses the **STRS** function to triplicate the value of each element of a dynamic array. Note that the third element is a null string:

```
test="A":@VM:"B":@VM:"":@VM:"D":@VM:"E"  
PRINT STRS("test",3)
```

It returns: AAAvBBBvvDDDvEEE

See Also

- [LENS](#) function
- [STR](#) function

- [Dynamic Arrays](#)

SUBR

Returns a value from an external subroutine.

```
SUBR(name[,arg1[,arg2...]])
```

Arguments

<i>name</i>	The name of an existing external subroutine, specified as a quoted string.
<i>arg</i>	<i>Optional</i> — An argument, or comma-separated list of arguments to pass to the subroutine.

Description

The **SUBR** function calls an existing subroutine. It optionally passes the subroutine one or more argument values. **SUBR** returns the value supplied by the subroutine.

You can create a subroutine using the **SUBROUTINE** statement.

SUBR, CALL, and GOSUB

The **SUBR** function is used to call an external subroutine that returns a value. The **CALL** statement is used to call an external subroutine that does not return a value. The **GOSUB** statement is used to call an internal subroutine.

Examples

The following example uses the **SUBR** function call a subroutine that computes the cube of a number:

```
INPUT mynum
x=SUBR(Cube,mynum)
PRINT "the cube of ":mynum:" is ":x
```

See Also

- [CALL](#) statement
- [GOSUB](#) statement
- [SUBROUTINE](#) statement

SUBS

Subtracts the values of corresponding elements in two dynamic arrays.

```
SUBS(dynarray1, dynarray2)
```

Arguments

<i>dynarray1</i>	The minuend. Any valid dynamic array of numeric values. If a dynamic array element contains a non-numeric value, SUBS treats this value as 0 (zero).
<i>dynarray2</i>	The subtrahend. Any valid dynamic array of numeric values. If a dynamic array element contains a non-numeric value, SUBS treats this value as 0 (zero).

Description

The **SUBS** function subtracts the value of each element in *dynarray2* from the corresponding element in *dynarray1*. It then returns a dynamic array containing the results of these subtractions. If an element value is an uninitialized variable, a null string, or a non-numeric value, **SUBS** parses its value as 0 (zero).

Examples

The following example uses the **SUBS** function to subtract the elements of two dynamic arrays:

```
a=11:@VM:22:@VM:33:@VM:44
b=10:@VM:9:@VM:8:@VM:7
PRINT SUBS(a,b)
    ! returns 1v13v25v37
```

See Also

- [ADDS](#) function
- [DIVS](#) function
- [MODS](#) function
- [MULS](#) function
- [Dynamic Arrays](#)

SUBSTRINGS

Returns a substring for each element of a dynamic array.

```
SUBSTRINGS( dynarray , start , length )
```

Arguments

<i>dynarray</i>	A dynamic array of elements from which a dynamic array of substrings is to be extracted.
<i>start</i>	A positive integer specifying the start position (counting from 1) within each element to begin extracting a substring.
<i>length</i>	A positive integer specifying the number of characters to extract from each element, beginning with the <i>start</i> position.

Description

The **SUBSTRINGS** function returns a dynamic array of substrings, each substring element containing the specified *length* number of characters from the corresponding *dynarray* element. **SUBSTRINGS** returns a substring for every element, regardless of the element's level delimiter.

If *start* is 1, substrings are extracted starting with the first character of each element. If *start* is 0, a negative number, the null string, a non-numeric string, or an undefined variable, **SUBSTRINGS** behaves as if *start*=1. If *start* is greater than the character length of an element, the returned dynamic array contains only the level delimiter for that element.

If *length* is greater than the *dynarray* element's length, the full element value is returned. If *length* is 0, a negative number, the null string, a non-numeric string, or an undefined variable the returned dynamic array contains only the level delimiters from the original *dynarray*.

If *start* or *length* is a mixed numeric string, the numeric part is parsed until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7.

You can use the [] string operator to perform a similar substring extract from a string. For further details, refer to the [Operators](#) page of this manual.

Examples

The following example uses the **SUBSTRINGS** function to return a dynamic array containing the first three characters of each element of a dynamic array:

```
cities="New York":@VM:"London":@VM:
"Chicago":@VM:"Boston":@VM:"Los Angeles"
alphalist=SUBSTRINGS(cities,1,3)
PRINT alphalist
! Returns: "NewvLonvChivBosvLos"
```

See Also

- [Dynamic Arrays](#)
- [Strings](#)

SUM

Adds the values of the elements of a single-level dynamic array.

```
SUM(dynarray)
```

Arguments

<i>dynarray</i>	Any valid dynamic array of numeric values.
-----------------	--

Description

The **SUM** function adds the values the elements in a dynamic array and returns the sum. If an element value is an uninitialized variable, a null string, or a non-numeric value, **SUM** parses its value as 0 (zero).

The **SUM** function adds dynamic array values that are on the same dynamic array level. To add all values in a dynamic array, regardless of level, use the **SUMMATION** function. To add the elements of two dynamic arrays, use the **ADDS** function.

Examples

The following example uses the **SUM** function to add the elements of a dynamic array:

```
a=10:@VM:9:@VM:8:@VM:7
PRINT SUM(a); ! returns 34
```

See Also

- [ADDS](#) function
- [SUMMATION](#) function
- [Dynamic Arrays](#)

SUMMATION

Adds the values of the elements of a multi-level dynamic array.

```
SUMMATION(dynarray)
```

Arguments

<i>dynarray</i>	Any valid dynamic array of numeric values.
-----------------	--

Description

The **SUMMATION** function adds the values of all of the elements in a dynamic array and returns the sum. If an element value is an uninitialized variable, a null string, or a non-numeric value, **SUMMATION** parses its value as 0 (zero).

The **SUMMATION** function adds all dynamic array values, regardless of dynamic array levels of the elements. To add only those elements that are on the same dynamic array level, use the **SUM** function. To add the elements of two dynamic arrays, use the **ADDS** function.

Examples

The following example uses the **SUMMATION** function to add the elements of a dynamic array:

```
a=10:@FM:9:@VM:8:@SM:7
PRINT SUMMATION(a);      ! returns 34
```

See Also

- [ADDS](#) function
- [MAXIMUM](#) function
- [MINIMUM](#) function
- [SUM](#) function
- [Dynamic Arrays](#)

TAN

Returns the tangent of an angle.

```
TAN(number)
```

Arguments

<i>number</i>	Any valid numeric expression that expresses an angle in radians.
---------------	--

Description

TAN takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle. The returned value is in radians.

To convert degrees to radians, multiply degrees by $\pi/180$. To convert radians to degrees, multiply radians by $180/\pi$.

Examples

The following example uses the **TAN** function to return the tangent of an angle:

```
Dim MyAngle
MyAngle = 1.3;           ! Define angle in degrees.
Print Tan(MyAngle);     ! Return tangent in radians.
```

The following example uses the **TAN** function to return the cotangent of an angle:

```
Dim MyAngle, MyCotangent
MyAngle = 1.3;           ! Define angle in degrees.
MyCotangent = 1 / Tan(MyAngle); ! Calculate cotangent.
Print MyCotangent;      ! Return in radians.
```

See Also

- [ATAN](#) function
- [COS](#) function
- [SIN](#) function
- [TANH](#) function
- Derived Math Functions
- ObjectScript: \$ZTAN function

TANH

Returns the hyperbolic tangent of an angle.

```
TANH(number)
```

Arguments

<i>number</i>	Any valid numeric expression that expresses an angle in degrees.
---------------	--

Description

TANH takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle. The returned value is in radians.

To convert degrees to radians, multiply degrees by pi/180. To convert radians to degrees, multiply radians by 180/pi.

Examples

The following example uses the **TANH** function to return the hyperbolic tangent of an angle:

```
Dim MyAngle
MyAngle = 1.3;           ! Define angle in degrees.
Print Tan(MyAngle);    ! Return htan in radians.
```

The following example uses the **TANH** function to return the cotangent of an angle:

```
Dim MyAngle, MyCotangent
MyAngle = 1.3;           ! Define angle in degrees.
MyCotangent = 1 / Tan(MyAngle); ! Calculate cotangent.
Print MyCotangent;     ! Return value in radians.
```

See Also

- [ATAN](#) function
- [COS](#) function
- [SIN](#) function
- [TAN](#) function
- [Derived Math Functions](#)

TIME

Returns the current local system time in internal format.

```
TIME ( )
```

Arguments

None. The parentheses are mandatory.

Description

The **TIME** function returns the current time in a format such as the following:

```
29848.349
```

This represents the elapsed number of seconds since midnight, with fractional seconds.

The **TIME** function is not the same as the `@TIME` system variable. **TIME** returns the current time when it is invoked. `@TIME` returns the time at which the current process started, this value does not change for the life of the process. For further details, see the [Variables](#) page of this manual.

Examples

The following example calls the **TIME** function to return the current system time in internal format, then uses the **OCNV** function to convert time from internal format to display format.

```
PRINT TIME ( )
PRINT OCONV ( TIME ( ) , "MTS" )
```

The following example shows the difference between the **TIME** function and the `@TIME` variable:

```
PRINT @TIME , TIME ( )
SLEEP 4
PRINT @TIME , TIME ( )
```

The two **TIME** functions return values that differ by approximately 4 seconds; the two `@TIME` variables return the same value.

See Also

- [TIMEDATE](#) function
- [OCNV](#) function

- Caché ObjectScript: \$HOROLOG special variable
- SQL: NOW function

TIMEDATE

Returns the current local date and time.

```
TIMEDATE ( )
```

Arguments

None. The parentheses are mandatory.

Description

The **TIMEDATE** function returns the current local date and time in the following format:

```
hh:mm:ss dd mmm yyyy
```

Time is represented on a 24-hour clock. Colons are used as the time separator. The date is represented as the number of days, the three-letter abbreviation for the month, and the year. Spaces are used as the date separator.

TIMEDATE does not return fractional seconds; for fractional seconds, use the **TIME** function.

See Also

- [TIME](#) function
- [OCONV](#) function
- Caché ObjectScript: \$HOROLOG special variable
- SQL: NOW function

TRIM

Removes leading and trailing characters from a string.

```
TRIM(string[,char[,code]])
```

Arguments

<i>string</i>	Any expression that resolves to a valid string .
<i>char</i>	<i>Optional</i> — The character to be trimmed, specified as a single-character string . The default is to trim blank spaces.
<i>code</i>	<i>Optional</i> — A code character specifying the type of trimming to perform, specified as a single-character string . The default is to trim leading, trailing, and redundant characters.

Description

The **TRIM** function trims the specified character from both ends of a string (leading and trailing characters). By default, it trims leading and trailing blank spaces, and replaces multiple (redundant) spaces with a single space. It returns the resulting trimmed string. The original input *string* is not changed.

The optional *char* argument is case sensitive. It trims the specified character until it encounters the first instance of another character.

You can use the **TRIMB** function to remove blank spaces from the back end of a string (trailing blanks). You can use the **TRIMF** function to remove blank spaces from the front end of a string (leading blanks).

Trim Codes

By default, **TRIM** removes leading, trailing and redundant space characters. You can perform other types of trim operation by specifying a single-character *code*. The following are the available *code* characters:

A	All occurrences of <i>char</i> removed from <i>string</i> .
B	Both leading and trailing occurrences of <i>char</i> removed.
D	Duplicate blank spaces and leading and trailing blank spaces removed. <i>char</i> must be specified, but its value is ignored.
E	Remove trailing spaces. <i>char</i> must be specified, but its value is ignored.
F	Remove leading spaces. <i>char</i> must be specified, but its value is ignored.
L	Leading occurrences of <i>char</i> removed.
R	Redundant, leading, and trailing occurrences of <i>char</i> removed. This is the default.
T	Trailing occurrences of <i>char</i> removed.

These *code* characters are not case sensitive.

Examples

The following example uses the **TRIM** function to trim leading and trailing spaces:

```
MyVar = TRIM("  Caché  ")
      ! MyVar contains "Caché".
PRINT LEN(MyVar), " [" :MyVar: "]"
```

The following example uses the **TRIM** function to trim leading and trailing lowercase “a”. In this case, leading a's are trimmed until an uppercase A is encountered and trailing a's are trimmed until a blank space is encountered:

```
MyVar = TRIM("aaaaaAnaconda aaaa", 'a')
      ! MyVar contains "Anaconda ".
PRINT LEN(MyVar), " [" :MyVar: "]"
```

See Also

- [TRIMB](#) function
- [TRIMF](#) function
- [LEFT](#) function
- [RIGHT](#) function
- [Strings](#)

TRIMB

Removes trailing blanks from a string.

```
TRIMB(string)
```

Arguments

<i>string</i>	Any expression that resolves to a valid string .
---------------	--

Description

The **TRIMB** function trims blank spaces from the back end of a string (trailing blanks). It returns the resulting trimmed string.

You can use the **TRIM** function to remove blank spaces (or other repetitive characters) from both ends of a string. You can use the **TRIMF** function to remove blank spaces from the front end of a string (leading blanks).

You can use the **TRIMFS** and **TRIMBS** functions to remove leading or trailing blanks from the elements of a dynamic array.

Examples

The following example uses the **TRIMB** function to trim trailing spaces:

```
MyVar = TRIMB("  Caché  ")
      ! MyVar contains "  Caché".
PRINT LEN(MyVar), "[" : MyVar : "]"
```

See Also

- [TRIM](#) function
- [TRIMBS](#) function
- [TRIMF](#) function
- [LEFT](#) function
- [RIGHT](#) function
- [Strings](#)

TRIMBS

Removes trailing blanks from each element of a dynamic array.

```
TRIMBS (dynarray)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
-----------------	---

Description

The **TRIMBS** function trims trailing blank spaces from each element of a dynamic array. It returns the resulting trimmed dynamic array.

TRIMBS does not trim leading blank spaces from dynamic array elements. You can use the **TRIMFS** function to remove leading blank spaces from each element of a dynamic array.

You can use **TRIM** to remove both leading and trailing blanks from a string. You can use **TRIMB** to remove trailing blanks from a string. You can use **TRIMF** to remove leading blanks from a string.

Examples

The following example uses the **TRIMBS** function to trim trailing spaces:

```
RawDyn="North   ":@VM:"South   ":@VM:"East ":@VM:"West..."
PRINT LENS(RawDyn)
TrimDyn = TRIMBS(RawDyn)
PRINT LENS(TrimDyn)
```

See Also

- [LENS](#) function
- [TRIMFS](#) function
- [TRIM](#) function
- [TRIMB](#) function
- [TRIMF](#) function
- [Dynamic Arrays](#)

TRIMF

Removes leading blanks from a string.

```
TRIMF(string)
```

Arguments

<i>string</i>	Any expression that resolves to a valid string .
---------------	--

Description

The **TRIMF** function trims blank spaces from the front end of a string (leading blanks). It returns the resulting trimmed string.

You can use the **TRIM** function to remove blank spaces (or other repetitive characters) from both ends of a string. You can use the **TRIMB** function to remove blank spaces from the back end of a string (trailing blanks).

You can use the **TRIMFS** and **TRIMBS** functions to remove leading or trailing blanks from the elements of a dynamic array.

Examples

The following example uses the **TRIMF** function to trim leading spaces:

```
MyVar = TRIMF("  Caché  ")
      ! MyVar contains " Caché".
PRINT LEN(MyVar), "[" : MyVar : "]"
```

See Also

- [TRIM](#) function
- [TRIMB](#) function
- [TRIMFS](#) function
- [LEFT](#) function
- [RIGHT](#) function
- [Strings](#)

TRIMFS

Removes leading blanks from each element of a dynamic array.

```
TRIMFS (dynarray)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
-----------------	---

Description

The **TRIMFS** function trims leading blank spaces from each element of a dynamic array. It returns the resulting trimmed dynamic array.

TRIMFS does not trim trailing blank spaces from dynamic array elements. You can use the **TRIMBS** function to remove trailing blank spaces from each element of a dynamic array.

You can use **TRIM** to remove both leading and trailing blanks from a string. You can use **TRIMF** to remove leading blanks from a string. You can use **TRIMB** to remove trailing blanks from a string.

Examples

The following example uses the **TRIMFS** function to trim leading spaces:

```
RawDyn="   North":@VM:"   South":@VM:"East":@VM:"   West.."
PRINT LENS(RawDyn)
TrimDyn = TRIMFS(RawDyn)
PRINT LENS(TrimDyn)
```

See Also

- [LENS](#) function
- [TRIMBS](#) function
- [TRIM](#) function
- [TRIMB](#) function
- [TRIMF](#) function
- [Dynamic Arrays](#)

UNASSIGNED

Determines if a variable is unassigned.

```
UNASSIGNED ( var )
```

Arguments

<i>var</i>	Any valid variable name . If <i>var</i> is not a valid variable name, MVBasic issues a syntax error.
------------	--

Description

The **UNASSIGNED** function determines whether a variable is assigned or not assigned. If *var* is not assigned a value, **UNASSIGNED** returns 1. If *var* is assigned a value, **UNASSIGNED** returns 0. An assigned value can be a single value or a dynamic array value. **UNASSIGNED** also returns 0 if *var* is assigned the null string or is assigned an unassigned variable.

The **ASSIGNED** function is the functional opposite of the **UNASSIGNED** function.

Examples

The following example tests the assignment of several variables. **UNASSIGNED** returns 0 (assigned) for variables *a* through *f*. **UNASSIGNED** returns 1 (unassigned) for variable *g*.

```
a=123
b="fred"
c=1:@VM:2:@VM:3
d=" "
e=NULL
f=g
PRINT UNASSIGNED(a)
PRINT UNASSIGNED(b)
PRINT UNASSIGNED(c)
PRINT UNASSIGNED(d)
PRINT UNASSIGNED(e)
PRINT UNASSIGNED(f)
PRINT UNASSIGNED(g)
```

Note that the assignment `f=g` assigns an empty string to `f` because `g` was previously unassigned.

See Also

- [ASSIGNED](#) function

UNICHAR

Returns the character corresponding to the specified character code.

```
UNICHAR ( charcode )
```

Arguments

<i>charcode</i>	A decimal integer that identifies a character. For 8-bit characters, the value in <i>charcode</i> must evaluate to a positive integer in the range 0 to 255. For 16-bit characters, specify integers in the range 256 through 65534.
-----------------	--

Description

The **UNICHAR** function takes a character code and returns the corresponding character. The **UNISEQ** function takes a character and returns the corresponding character code.

Numbers from 0 to 31 are the same as standard, nonprintable ASCII codes. For example, **UNICHAR**(10) returns a linefeed character.

Note: **UNICHAR** and **CHAR** are functionally identical. On Unicode systems both can be used to return 16-bit Unicode characters. On 8-bit systems, these functions return a null string for character codes beyond 255.

The Caché MVBasic **UNICHAR** function returns a single character. The corresponding Caché ObjectScript **\$CHAR** function can return a string of multiple characters by specifying a comma-separated list of ASCII codes. The Caché MVBasic **UNICHARS** function takes a dynamic array of ASCII codes and returns the corresponding single characters as a dynamic array.

Examples

The following example uses the **UNICHAR** function to return the character associated with the specified character code:

```
PRINT UNICHAR(65);      ! Returns A.
PRINT UNICHAR(97);      ! Returns a.
PRINT UNICHAR(37);      ! Returns %.
PRINT UNICHAR(62);      ! Returns >.
```

The following example uses the **UNICHAR** function to return the lowercase letter characters of the Russian alphabet on a Unicode version of Caché. On an 8-bit version of Caché it returns a null string for each letter:

```
letter=1072
FOR x=1 TO 32
  PRINT UNICHAR(letter)
  letter=letter+1
NEXT
```

See Also

- [CHAR](#) function
- [CHARS](#) function
- [SEQ](#) function
- [UNISEQ](#) function
- ObjectScript: [\\$CHAR](#) function

UNICHARS

Returns the character corresponding to the specified character code for each element of a dynamic array.

```
UNICHARS(dynarray)
```

Arguments

<i>dynarray</i>	A valid dynamic array of decimal integers that identify characters . For 8-bit characters, these element values must evaluate to positive integers in the range 0 to 255. For 16-bit characters, these element values must evaluate to positive integers in the range 256 through 65534.
-----------------	--

Description

The **UNICHARS** function takes a dynamic array of character codes and returns the corresponding characters. It returns these values as a dynamic array. The **UNISEQS** function takes a dynamic array of characters and returns the corresponding ASCII codes.

Numbers from 0 to 31 are the same as standard, nonprintable ASCII codes. For example, **UNICHARS**(10) returns a linefeed character.

Note: **UNICHARS** and **CHARS** are functionally identical. On Unicode systems both can be used to return 16-bit Unicode characters. On 8-bit systems, these functions return a null string for character codes greater than 255.

The Caché MVBasic **UNICHARS** function returns a dynamic array of characters. The corresponding Caché ObjectScript **\$CHAR** function returns a string of characters by specifying a comma-separated list of ASCII codes.

Examples

The following example uses the **UNICHARS** function to return a dynamic array of the characters associated with each specified ASCII character code:

```
a=65:@VM:66:@VM:67:@VM:68
PRINT UNICHARS(a); ! returns AvBvCvD
```

The following example uses the **UNICHARS** function to return the first four letters of the Greek alphabet. On a Unicode version of Caché it returns the Greek letters in a dynamic array; on an 8-bit version of Caché it returns a dynamic array with a null string for each letter:

```
b=945:@VM:946:@VM:947:@VM:948
PRINT UNICHARS(b)
```

See Also

- [CHARS](#) function
- [CHAR](#) function
- [UNISEQS](#) function
- [Dynamic Arrays](#)
- ObjectScript: [\\$CHAR](#) function

UNISEQ

Returns the character code corresponding to a specified character.

```
UNISEQ(char)
```

Arguments

<i>char</i>	Any valid character. If <i>char</i> is a string, UNISEQ returns the value of the first character.
-------------	--

Description

The **UNISEQ** function takes a character and returns the corresponding Unicode numeric code. Its inverse, the **CHAR** function takes a numeric code and returns the corresponding character.

The Caché MVBasic **UNISEQ** function returns the numeric value for a single character. The corresponding Caché ObjectScript **\$ASCII** function can take a string of characters and return the numeric value for a specific character by specifying its position in the string.

Note: **UNISEQ** and **SEQ** are functionally identical.

Examples

The following example uses the **UNISEQ** function to return the numeric code associated with the specified character:

```
PRINT UNISEQ('A');      ! Returns 65.
PRINT UNISEQ('a');      ! Returns 97.
PRINT UNISEQ('%');      ! Returns 37.
PRINT UNISEQ('>');      ! Returns 62.
```

The following example uses the **UNISEQ** function to return lowercase letter characters and associated numeric codes of the Russian alphabet. On a Unicode version of Caché it returns the Russian letters; on an 8-bit version of Caché it returns a -1 (indicating a null string) for each letter:

```
letter=1072
FOR x=1 TO 32
  glyph=CHAR(letter)
  PRINT UNISEQ(glyph),glyph
  letter=letter+1
NEXT
```

See Also

- [SEQ](#) function
- [CHAR](#) function
- ObjectScript: [\\$ASCII](#) function

UNISEQS

Returns the character code for the first character of each element in a dynamic array.

```
UNISEQS (dynarray)
```

Arguments

<i>dynarray</i>	Any valid dynamic array .
-----------------	---

Description

The **UNISEQS** function takes a dynamic array and returns the corresponding numeric codes for the first character in each element. It returns these character codes as a dynamic array. If an element consists of a string of more than one character, **UNISEQS** returns the numeric value of the first character of that element. If an element contains an uninitialized variable or a null string, **UNISEQS** returns -1 for that element.

If the first character of a dynamic array element is one of the following dynamic array level delimiters: CHAR(252), CHAR(253), or CHAR(254), **UNISEQS** treats this character as a level delimiter, and returns -1 for the null element(s) established by parsing this character as a level delimiter.

Note: **UNISEQS** and **SEQS** are functionally identical. On Unicode systems both can be used to return character codes for 16-bit Unicode characters. On 8-bit systems, these functions return that character code of the first 8 bits of a 16-bit Unicode character.

The **UNICHARS** function is the inverse of **UNISEQS**. It takes a dynamic array of numeric codes and returns the corresponding characters.

The **UNISEQ** function (or **SEQ** function) takes the first character of a string and returns the corresponding numeric code.

The **UNISEQS** function returns the numeric value for the first character of each element as a dynamic array element. The corresponding Caché ObjectScript **\$ASCII** function can take a string of characters and return the numeric value for a specific character by specifying its position in the string.

Examples

The following example uses the **UNISEQS** function to return the numeric codes associated with each character in a dynamic array:

```
alpha="A":@VM:"B":@VM:"C":@VM:"D"  
PRINT UNISEQS(alpha)  
! returns 65v66v67v68
```

The following example returns the numeric codes associated with four lowercase Russian letters in a dynamic array. On a Unicode system, it returns the Russian character codes. On an 8-bit system, characters beyond 255 are treated as null strings, so **UNISEQS** returns -1 for each element.

```
russian=CHAR(1072):@VM:CHAR(1073):@VM:CHAR(1074):@VM:CHAR(1075)  
PRINT UNISEQS(russian)
```

See Also

- [SEQS](#) function
- [CHARS](#) function
- [UNICHARS](#) function
- [SEQ](#) function
- [Dynamic Arrays](#)
- ObjectScript: `$ASCII` function

UPCASE

Converts alphabetic characters to uppercase.

```
UPCASE(string)
```

Arguments

<i>string</i>	Any valid string or numeric expression.
---------------	---

Description

The **UPCASE** function returns a string of characters with all lowercase letters converted to uppercase. Characters other than lowercase letters are passed through unchanged. If you specify a null string, **UPCASE** returns a null string.

By default, **UPCASE** performs case conversion on ANSI Latin-1 letters. To perform case conversion on letters in other character sets, you must set the appropriate locale.

The **OCNV** function with the “MCU” option is functionally identical to the **UPCASE** function. To convert uppercase to lowercase, use the **DOWNCASE** function.

Examples

The following example uses the **UPCASE** function to convert lowercase letters to uppercase:

```
MyString = "Caché from InterSystems"
PRINT UPCASE(MyString)
! Returns "CACHE FROM INTERSYSTEMS"
```

See Also

- [DOWNCASE](#) function
- [OCNV](#) function

XTD

Converts a number from hexadecimal to decimal.

```
XTD(hexnum)
```

Arguments

<i>hexnum</i>	A positive hexadecimal integer expressed as a string .
---------------	--

Description

The **XTD** function returns a hexadecimal integer converted to decimal. The *hexnum* value must be a positive hexadecimal integer. **XTD** returns the corresponding decimal value. If *hexnum* is zero, a negative number, or a non-numeric string, **XTD** returns 0. If *hexnum* is a mixed numeric string, the numeric part is parsed until a non-numeric character is encountered. Thus “7Dwarves” is parsed as 7D. If *hexnum* is the null string or an undefined variable, a syntax error occurs.

Use **DTX** to convert from decimal to hexadecimal.

Examples

The following examples use the **XTD** function to return a decimal number:

```
PRINT XTD(12);           ! Returns 12
PRINT XTD("1C");        ! Returns 28
PRINT XTD("-1C");       ! Returns 0
PRINT XTD("red");       ! Returns 0
```

See Also

- [DTX](#) function

Caché MultiValue Basic General Concepts

Comments

A comment line indicator.

Description

A comment is text within a program that is not executed. Comments are used for documenting source code. They do not become part of the executable program and do not affect the size or performance of the object code.

A comment can be on a separate program line, or can follow an executable statement on the same line. If a comment follows an executable statement on the same line, the executable statement must end with a semicolon (;).

There are four ways to indicate a comment: the **REM** statement, an asterisk (*), an exclamation mark (!), or a dollar sign asterisk (\$*).

All MVBasic comments indicators are single-line comments. You must begin each line of a comment with a comment indicator.

Examples

The following examples are all valid comments:

```
PRINT TIMEDATE(); ! comment text
! several lines of
! additional comment text
PRINT TIMEDATE(); * comment text
* further comments
PRINT TIMEDATE(); REM comment text
REM this is a comment
PRINT TIMEDATE(); $* comment text
```

See Also

- [REM statement](#)

Dynamic Arrays

A user-defined structure for storing multiple data values.

Description

A dynamic array is a uniquely named entity used to store and retrieve multiple data values. These data values are called “elements”. These elements can be flat (all on the same level), or in a hierarchical structure. A dynamic array is a [string](#). Its elements are marked by special delimiter characters within the string.

A dynamic array is assigned element values by using the equal sign (=), the same as an ordinary string. The naming conventions for dynamic arrays are the same as for [variables](#). Caché MVBasic does not distinguish between dynamic array names and variable names; you cannot assign the same name to a dynamic array and to a single-value variable.

Note: MVBasic also supports standard arrays, using the **DIM** statement. These should not be confused with dynamic arrays, which are a unique feature of MultiValue database systems.

The scope of a dynamic array is the current process.

Dynamic arrays are assigned element values using the following format:

```
val1:@nM:val2:@nM:val3
```

The dynamic array level is specified by the @nM special variable, which resolves to a single character that is used as a level delimiter. This delimiter character is concatenated into the string between two data values, using the colon (:) [string concatenation operator](#). The level is specified by the first letter of this code variable, as shown in the following table. Levels are listed in descending order:

Level code variable	Level abbreviation and name	Character value
@IM	I = Item Mark	CHAR(255)
@FM or @AM	F = Field Mark or Attribute Mark	CHAR(254)
@VM	V = Value Mark	CHAR(253)
@SM or @SVM	S = Subvalue Mark	CHAR(252)
@TM	T = Text Mark	CHAR(251)
	Z	CHAR(250)

Note: Caché MVBasic supports the UniVerse dynamic array levels to CHAR(250). It does not support UniVerse levels below CHAR(250). It does not support UniData dynamic array levels (such as @RM) that are not supported by the UniVerse implementation.

You can use the [RAISE](#) and [LOWER](#) functions to change the level of dynamic array elements.

Dynamic Array Extract and Replace Operators

You use the <> operators to extract an element value from a dynamic array, or to replace an element value in a dynamic array with another value. These operators use the following formats:

```
<f>
<f , v>
<f , v , s>
```

Where *f*, *v*, and *s* are comma-separated integers representing the position of the desired element. For example, <2,3> indicates the 3rd Value Mark value within the 2nd Field Mark field.

To extract an element value from a dynamic array:

```
var=dynarray<f , v , s>
```

Where *dynarray* is a dynamic array, and the angle brackets specify by position the element to extract into the *var* variable.

To replace an element value in a dynamic array:

```
dynarray<f,v,s>=newval
```

Where *dynarray* is a dynamic array, and the angle brackets specify by position the element to replace with the *newval* value.

Labels

A program line identifier.

Description

A label is a unique identifier for a program line. A label must appear in column 1; it cannot be indented. A label can be on a line by itself, or be followed by an MVBASIC command on the same line.

The following are the naming conventions for labels:

- A label can contain letters, numbers, the period (.), dollar sign (\$), and percent sign (%) characters. The first character of the label must be a letter or a number.
- A label is followed by a colon (:). This colon can be omitted if all of the characters of the label are numbers.
- Letters in labels are case-sensitive.
- A label can be of any length, but only the first 64 characters are significant.

Labels are used by the **GOTO** statement. This statement may be abbreviated as the **GO** statement.

MATCH Pattern Matching

A pattern match operator.

```
string MATCH code
string MATCHES code

string MATCH string
string MATCHES string
```

Arguments

<i>string</i>	Any valid string expression .
<i>code</i>	A pattern match code, specified as a quoted string.

Description

The MATCH (or MATCHES) operator has two forms: a pattern match operation (`string MATCH code`) and an equality match operation (`string MATCH string`).

Pattern Match Operator

The MATCH (or MATCHES) operator performs a pattern match test on *string* resulting in a boolean value: 1=*string* matches *code*; 0=*string* does not match *code*. For a match to occur, every character of *string* must exactly match *code*. The exception to this is the null string (""), which matches all character type codes and has a length of 0.

The following are the available *code* values:

Code	Meaning
...	Matches any number of characters of any type. This includes CHAR(250) through CHAR(255), the dynamic array delimiter characters. Also matches the null string ("").
0X	Matches any number of characters of any type. This includes CHAR(250) through CHAR(255), the dynamic array delimiter characters. Also matches the null string ("").
nX	Matches exactly <i>n</i> number of characters of any type.
0A	Matches any number of alphabetic characters. The alphabetic characters include CHAR(250) through CHAR(255), the dynamic array delimiter characters. Also matches the null string ("").
nA	Matches exactly <i>n</i> number of alphabetic characters.
0N	Matches any number of number characters, defined as the numbers 0 through 9. Number characters <i>do not</i> include the plus sign, minus sign, or decimal separator. Also matches the null string ("").
nN	Matches exactly <i>n</i> number of number characters.

code characters are not case-sensitive.

You can specify multiple *code* characters to match complex string patterns. For example a *code* of "0X3A4N" matches any number of characters of any type, followed by three alphabetic characters, followed by four number characters.

Equality Match Operator

The MATCH (or MATCHES) operator performs an equality match test on *string* resulting in a boolean value: 1=*string* matches *string*; 0=*string* does not match *string*. These matches must be identical, and are case sensitive.

The exception to string equality matching is that the characters CHAR(253) and CHAR(254) (or any string containing them) do not return an equality match. Any string match containing one of these characters returns 0. This exception is provided because these characters are dynamic array delimiter characters.

Combining Pattern and Equality Matching

You can use the MATCH (or MATCHES) operator to mix pattern match and equality match operations. An equality match string must be specified as a quoted string, and the entire match code must be a quoted string. Therefore, to combine pattern codes and equality match strings, you must use both double quotes and single quotes. For example, "' ('3N') '3N' - '4N'" is

the pattern code for a telephone number such as (617) 123–4567. You may include single quotes within double quotes (as shown above) or double quotes within single quotes.

See Also

- [Strings](#)
- [Dynamic Arrays](#)

MultiValue Files

A data storage structure.

Description

A MultiValue file is a data storage structure created using Caché global variables. It is a fundamental part of MultiValue database architecture, corresponding to the UniVerse or UniData hashed data or dictionary file.

A MultiValue file is created using the **CREATE.FILE** verb. It is cataloged in the VOC file as a global variable, and can be concurrently accessed by multiple processes. You can use the **FILEINFO** function to determine the pathname of the global variable.

The **OPEN** statement opens a MultiValue file and returns the *filevar* local variable. This *filevar* is used for all subsequent MVBasic operations on this MultiValue file:

- **WRITE** is used to write data to a MultiValue file.
- **READ** is used to read data from a MultiValue file.
- **DELETE** is used to delete a data record from a MultiValue file.
- **CLEARFILE** is used to delete all data records in a MultiValue file.
- **SELECT** is used to read the record identifiers from a MultiValue file into a select list. These record identifiers can then be individually read using the **READNEXT** statement.
- **CLOSE** is used to close a MultiValue file, resetting *filevar* to null.
- The **FILEINFO** function is used to determine the status of *filevar* and other information about a MultiValue file.

Operators

Arithmetic, logical, and string operators.

Description

An operator is a symbol that causes an operation to be performed on the two values to either side of it. There are four types of operators: arithmetic, logical, string, and pattern matching. Pattern matching is described in the [MATCH](#) reference page.

Spaces are permitted (but not required) between operators and their operands.

Arithmetic Operators

The following are the arithmetic operators supported by Caché MVBasic:

=	Numeric equality operator.
+	Addition operator.
+=	Increment (addition assignment) operator.
-	Subtraction operator.
-=	Decrement (subtraction assignment) operator.
*	Multiplication operator.
*=	Multiplication assignment operator.
/	Division operator.
**	Exponentiation operator.
()	Grouping (nesting) operator.

By default, Caché MVBasic order of operations is to perform exponentiation, then division, then multiplication, then subtraction, then addition. You can change this order of operations by using parentheses to nest operations. Note that Caché ObjectScript uses a different order of operations; it uses strict left-to-right evaluation of operators.

Logical Operators

The following are the logical operators supported by Caché MVBasic. Logical operators result in a boolean result, either 1 (true) or 0 (false):

= EQ	Equal to operator.
< LT	Less Than operator.
> GT	Greater Than operator.
<= =< #< LE	Less Than or Equal to operator.
>= => #> GE	Greater Than or Equal to operator.
<> NE	Not equal to operator.
& AND	Logical AND operator.
! OR	Logical OR operator.

String Operators

The following are the string operators supported by Caché MVBasic:

=	String equality operator. String equality is case-sensitive.
: CAT	Concatenation operator. Placed between two expressions, strings, or numeric values to be concatenated. When using the : operator you can include or omit blank spaces. When using the CAT operator you must use a blank space when concatenating a variable.
[]	<p>Substring extract operator. Placed after a string, the brackets enclose positive integers:</p> <p>string[start,length] specifies the start position and length of the substring to be extracted from the start of the string.</p> <p>string[length] specifies the length of the substring to be extracted from the end of the string.</p>
<>	Dynamic array element extract or replacement operator. For further details see the Dynamic Arrays page of this manual.

The following example demonstrates the string equality operator:

```

! Strings are case sensitive:
PRINT "Fred"="Fred"      ! Returns 1 (True)
PRINT "Fred"="fred"     ! Returns 0 (false)
! Number/Numeric strings equality:
PRINT "7"=7            ! Returns 1 (True)
PRINT +007.00="7"     ! Returns 1 (True)
PRINT "+007.00"="7"   ! Returns 1 (True)
! Null string equality:
PRINT ""=""           ! Returns 1 (True)
PRINT ""=NULL        ! Returns 1 (True)
PRINT ""=0            ! Returns 0 (False)
! Unassigned variables equality
! (variables aaa and bbb are unassigned):
PRINT aaa=bbb        ! Returns 1 (True)
PRINT aaa=""         ! Returns 1 (True)
PRINT aaa=NULL       ! Returns 1 (True)

```

The following example demonstrates the string concatenation operator:

```

! String concatenation:
PRINT "fire":"fly"    ! Returns "firefly"
PRINT "fire":":":"fly" ! Returns "firefly"
PRINT "fire":" ":"fly" ! Returns "fire fly"
! Number/Numeric strings concatenation:
PRINT "7":7          ! Returns "77"
PRINT +007.00:"7"   ! Returns "77"
PRINT 7:"+007.00"   ! Returns "7+007.00"
PRINT .0:.0         ! Returns "00"
! Null string concatenation:
PRINT "":"          ! Returns null string
PRINT "":NULL       ! Returns null string
! Unassigned variables concatenation
! (variables aaa and bbb are unassigned):
PRINT aaa:bbb       ! Returns null string

```

The following example demonstrates the substring extract operator:

```
x="The quick brown fox"
! Extract from beginning of string:
PRINT x[5,5] ! Returns "quick"
PRINT x[5,99] ! Returns "quick brown fox"
PRINT x[1,3] ! Returns "The"
PRINT x[0,3] ! Returns "The"
PRINT x["",3] ! Returns "The"
! Extract from end of string:
PRINT x[3] ! Returns "fox"
PRINT x[1] ! Returns "x"
PRINT x[0] ! Returns null string
PRINT x[""] ! Returns null string
```

See Also

- [Strings](#)
- [Pattern Match Operators](#)

Strings

A delimited data literal.

Description

A string is a data literal delimited by an opening and closing delimiter character. A string can contain any character, except the delimiter character itself. For this reason, MVBasic supports three alternative delimiter characters:

- The double quote character (") is most commonly used to delimit a string. It permits the inclusion of single quotes and apostrophes (') within the string, the inclusion of the backslash, and is compatible with other programming languages. For example: "Tom's string of data"
- The single quote character (') can be used to delimit a string. It permits the inclusion of double quotes within the string, the inclusion of the backslash, and is compatible with SQL code. For example: 'His "important" data'
- The backslash character (\) can be used to delimit a string. It permits the inclusion of both double quotes and single quotes within the string. It is not compatible with other programming languages. For example: \Tom's "important" data\

Strings with these three types of delimiters may be freely mixed. For example, it is possible to concatenate strings with different delimiters, as shown in the following example: 'His "important" data':" isn't very important".

A string is a literal, and is not parsed. The exception to this is when a string is being input as a numeric value.

Strings and Numerics

When a string is handled as a numeric value, the string may be a numeric string or a mixed numeric string.

A *numeric string* is a string that contains only numeric characters (the number 0 through 9, the plus and minus sign, and the decimal point delimiter). A numeric string is always accepted as a numeric. Thus "123.4" is identical to 123.4. The distinction between numerics (which do not require a delimiter) and numeric strings is that numerics are always converted to canonical form before being used. This means that all leading signs are resolved and removed (except for a single minus sign), that leading and trailing zeros are removed, and the decimal point is removed if not followed by a fractional value. A numeric string is not automatically converted to canonical form. This may be useful when it is important to preserve plus signs or trailing zeros, for example in dollars and cents amounts.

A *mixed numeric string* is a string that contains both numeric and non-numeric characters, with the first character (or characters) in the string being numeric. For example: "7 dwarves" or "12.5 kilometers". The treatment of mixed numeric strings is specific to each function or command. In many cases, the numeric is parsed until the first non-numeric character is encountered. Thus "12.5 kilometers" is parsed as 12.5. In other cases, the string is treated as a non-numeric string, which in MVBasic is assigned a numeric value of 0. In these cases, "12.5 kilometers" is parsed as 0.

Variables

User-defined variables and system-defined @ variables used for storing data values.

Description

A variable is a unique named entity used to store and retrieve a data value. The following are the available types of variables in Caché MVBasic:

- Local variables, the scope of which is the current process.
- Process-private global variables, the scope of which is the current process.
- Global variables, systemwide in scope.
- System variables, identified by an @ character as the first character.

Naming Conventions

The following are the naming conventions for local variables:

- The first character of a local variable name must be a letter, percent (%), or dollar (\$) character. If the first character is a dollar (\$) character, all of the other characters must be letters. The second and subsequent characters may be letters, numbers, the period (.), dollar (\$), underscore (_), and percent (%) characters. The last character cannot be an underscore (_) character.
- Letters in variable names are case-sensitive.
- Local variable names are limited to 31 characters. You may specify a name longer than 31 characters, but only the first 31 characters are used. Therefore, a local variable name must be unique within its first 31 characters.

A global variable begins with the caret (^) character, indicating that it is a global variable. A global variable follows the Caché ObjectScript naming conventions, not the MultiValue variable naming conventions. For further details, see the Variables chapter of *Using Caché ObjectScript*.

A process-private global variable begins with the ^| | characters (or the ^| "^" | characters), indicating that it is a process-private global. The two syntactic forms are equivalent. A process-private global follows the Caché ObjectScript naming conventions, not the MultiValue variable naming conventions. For further details, see the Variables chapter of *Using Caché ObjectScript*.

Assignment of Values

A variable is assigned a value by using the equal sign (=), as shown in the following examples. Spaces can be included or omitted before or after the equal sign.

```
x="fred"  
y=+1234.5  
z=x:y
```

A variable can be assigned multiple values as a dynamic array. For details on defining a dynamic array, refer to [Dynamic Arrays](#).

@ Variables

Caché MVBasic provides a number of system variables, identified by an @ sign as the first character of their names. These variables are set by MVBasic. They cannot be set by user programs. If no value is set, these variables contain the empty string.

A special case of these @ variables are the dynamic array level delimiter characters. These are single characters, represented by @AM, @FM, @IM, @SM, @SVM, @TM, and @VM.

For further details on these special characters, refer to the [Dynamic Arrays](#) page in this manual.

The following are supported @ variables:

@ACCOUNT	The user login name (User).
@AUTHORIZATION	Current user name.
@COMMAND	The command line that invoked this program. (See Note 2)
@CRTHIGH	Number of lines displayed in the terminal window.
@CRTWIDE	Number of columns displayed in the terminal window.
@DATE	The date when the current process started, in internal format (see Note 1). To convert to display format, use the OCONV function. (See Note 1)
@DAY	The day of the month when the current process started, specified as an integer. (See Note 1)
@FALSE	The 0 character, representing the boolean value.
@FILE.NAME	Same as @FILENAME
@FILENAME	The pathname specified in the most recent invocation of OPENSEQ . If the sequential file exists, @FILENAME contains <i>pathname</i> as a file pathname. If the sequential file does not exist, @FILENAME contains <i>pathname</i> as a directory pathname. No other validation is performed on the pathname. This @FILENAME value is only changed by another invocation of OPENSEQ . It is not changed by operations such as creating a file, closing a file, or deleting a file.
@ID	Current record ID.
@LEVEL	Nested level of execution. Starts at 0.
@LOGNAME	The user login name.
@LPTRHIGH	Number of lines on the current output device, either a printer or a terminal window.
@LPTRWIDE	Number of columns on the current output device, either a printer or a terminal window.
@MONTH	The month of the year when the current process started, specified as an integer. (See Note 1)

@PARASENTECE	The command line that invoked this program. (See Note 2)
@PATH	The full pathname for the current account. For a terminal session running the MV shell, the pathname is: c:\cachesys\mgr\user.
@SELECTED	Number of elements selected from the most recent select list. See the SELECT statement. Defaults to 0.
@SENTENCE	The command line that invoked this program. (See Note 2)
@SYS.BELL	The ASCII bell character (CHAR(7)). Printing this variable rings the bell.
@SYSTEM.RETURN.CODE	Status code for system processes.
@SYSTEM.SET	Status code for system processes.
@TERM.TYPE	The terminal type for the current terminal. For example, vt220.
@TIME	The time when the current process started, in internal format (see Note 1). @TIME rounds to whole seconds. To return the current time in internal format, use the TIME function. The TIME function includes fractional seconds. To convert from internal format to display format, use the OCONV function.
@TRANSACTION	An integer that specifies whether a transaction is active. 0 indicates no active transaction.
@TRUE	The 1 character, representing the boolean value.
@TTY	The terminal device name (Device). For example: TRM : 436.
@UDTNO	UniData Terminal Number. A unique integer assigned to a terminal process. Corresponds to the port number returned by the LISTME MultiValue command. Exiting and re-entering the MV shell does not change this integer value.
@USER.NO	Same as @USERNO.
@USERNO	The port number of the current process. (See Note 3)
@WHO	Name of the current account.

@YEAR	The year when the current process started, specified as two digits. The expansion of two-digit years to four digits is governed by the MultiValue CENTURY.PIVOT verb, described in <i>Operational Differences Between MultiValue and Caché</i> . (See Note 1)
@YEAR4	The year when the current process started, specified as four digits. (See Note 1)

Note 1

This variable is computed when a program is started and does not change during execution.

Note 2

The timing and nature of updates to these variables is very emulation dependent.

Note 3

Port numbers are an optional site configurable value. The default value is the Caché process number.