Caché MultiValue Basic Reference

Version 2018.1
2020-11-13

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Caché MultiValue Basic Reference
Caché Version 2018.1 2020-11-13

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

About This Book

This book provides reference material for various elements of Caché MVBasic: commands, functions, system variables, constants, operators, and symbols.

This book contains the following sections:

- Symbols
- Caché MultiValue Basic Commands
- Caché MultiValue Basic Functions
- Caché MultiValue Basic General Concepts

There is also a detailed Table of Contents.

Other related topics in the Caché documentation set are:

- *MultiValue Basic Quick Reference*
- *Using the MultiValue Features of Caché*
- *Operational Differences between MultiValue and Caché*
- *Caché MultiValue Commands Reference*
- *Caché MultiValue Query Language (CMQL) Reference*
- *The Caché MultiValue Spooler*

For general information, see *Using InterSystems Documentation*.

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 ObjectScript.

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

Symbol	Name and Usage
[space] or [tab]	*White space (Tab (9) or Space (32)):* One or more whitespace characters between keywords, identifiers, and variables.
!	*Exclamation Mark (33):* Single-line comment indicator. Logical OR operator.
"	*Double Quote (34):* Used to enclose string literals. You can use "" to specify an empty string.
#	*Pound (35):* Not Equal To operator (inequality logical operator). For example, `3#4` returns 1 (True). Command line command specifying that the statement following it be executed as an ObjectScript command. See the *Caché MultiValue Commands Reference*.
#<	*Pound, Less than:* Greater than or equal to operator (symbols mean not less than).
#>	*Pound, Greater than:* Less than or equal to operator (symbols mean not greater than).
$	*Dollar sign (36):* Permitted character in variable names.
$*	*Dollar sign, Asterisk:* A single-line comment indicator.
%	*Percent sign (37):* Permitted character in variable names.
&	*Ampersand (38):* Logical AND operator.
'	*Single Quote (39):* Used to enclose string literals. You can use '' to specify an empty string.
( )	*Parentheses (40,41):* Used to enclose a procedure or function parameter list. For example, SYSTEM(15), or OCONV(12345,"D"). Used to enclose a command or program option. For example, `COMPILE.TERM (VT)`. When used with a command option, the closing parenthesis is optional; for example, `COMPILE.TERM (VT`. Used to nest expressions; nesting overrides the default order of operator precedence. Used to specify static array subscripts; a subscripted array has to be dimensioned using the DIM command. In CALL statement, used to specify argument passed by value.
*	*Asterisk (42):* Multiplication operator. Single-line comment indicator.
**	*Double Asterisk:* Exponentiation operator.

Symbol	Name and Usage
*=	*Asterisk, Equal:* Multiplication assignment operator.
+	*Plus sign (43):* Addition operator.
++	*Double Plus sign:* Increment operator.
+=	*Plus, Equal:* Addition assignment (increment) operator.
,	*Comma (44):* Used to separate parameters in a function parameter list. Used to separate subscripts in a static array; a subscripted array has to be dimensioned using the DIM command. At the end of a line of code, a line continuation indicator. In DIM statements, used to separate multiple assignments. In PRINT or CRT statements, inserts a tab between arguments.
–	*Minus sign (45):* Unary arithmetic negative operator. Subtraction operator.
—	*Double Minus sign:* Decrement operator.
–=	*Minus, Equal:* Subtraction assignment (decrement) operator.
–>	*Arrow (minus, greater than):* Object class indicator.
.	*Period (46):* Decimal point character. Permitted character in variable names; cannot be first character. MV Shell command stack command prefix, followed by the letter A, C, D, L, U, or X, or the ? character. See the *Caché MultiValue Commands Reference*.
...	*Three Periods:* MATCH operator pattern match code.
.?	*Period, Question Mark:* MV Shell command stack command to display available commands. See the *Caché MultiValue Commands Reference*.
/	*Slash (47):* Division operator. In COMMON statement, used to enclose a storage area name. For example, `/sharedvars/`.

//	*Double Slash:* As a prefix to a directory name, allows MultiValue to directly reference the directory. For example, to create the Windows file C:/temp/results.txt:
	`OPEN "//C:/temp" TO DSCB`
	`WRITE results ON DSCB,"results.txt"`

/=	*Slash, Equal:* Division assignment operator.
:	*Colon (58):* Label suffix. For example, `LabelOne:`. String concatenation operator. In INPUT statement, a suffix to the variable or length arguments that suppresses a line return. MV Shell prompt character.

Symbol	Name and Usage
:=	*Colon Equals:* String concatenation assignment operator.
;	*Semicolon (59):* MV Shell MVBasic language command prefix. For example, `;print date()`. See the *Caché MultiValue Commands Reference*. MVBasic statement end indicator. Optional, unless the statement is followed on the same line by another MVBasic statement, or by an in-line comment. In INSERT and REPLACE functions, an argument separator.
;!	*Semicolon Exclamation Mark:* In-line comment indicator.
;*	*Semicolon Asterisk:* In-line comment indicator.
;/	*Semicolon Slash:* A command issued from the debug prompt that displays variable values.
<	*Less than (60):* Less than operator.
<=	*Less than, Equal:* Less than or equal to operator.
<>	*Less than, Greater than:* Not Equal To operator (inequality logical operator). For example, `3<>4` returns 1 (True). Used to enclose integers specifying the Field, Value, and Subvalue level of a dynamic array element. For example, `<1,2,2>`.
<< ... >>	*Double less than, double greater than:* an inline prompt, used to interactively request an input value. Inline prompts can be used in MVBasic statements or MultiValue command line commands. Described in the *Caché MultiValue Commands Reference*.
=	*Equal sign (61):* Equality operator. For example, `3=4` returns 0 (False). Assignment operator.
=<	*Equal, Less than:* Less than or equal to operator.
=>	*Equal, Greater than:* Greater than or equal to operator.
>	*Greater than (62):* Greater than operator.
>=	*Greater than, Equal:* Greater than or equal to operator.
@	*At sign (64):* Prefix for system variable names (for example `@RECORD`). Prefix for system variables that specify the characters used for dynamic array level delimiters (for example, `@FM`). The @ 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"`
[ ]	*Square Brackets (91 & 93):* Substring extract operator; brackets enclose integers specifying the substring to extract.
[	*Left Square Bracket (91):* Command line command specifying that the statement following it be executed as an ObjectScript command. See the *Caché MultiValue Commands Reference*. In the MultiValue ED editor, an Escape key is displayed as "[". See the *Caché MultiValue Commands Reference*.

Symbol	Name and Usage
\	*Backslash (92):* 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.
]	*Right Square Bracket (93):* In HEADING or FOOTING, starts a new line.
^	*Caret (94):* as a prefix to a variable name, indicates a Caché global variable. In HEADING or FOOTING, inserts a page number.
_	*Underscore (95):* In INPUT, a suffix to the length argument that makes a line return mandatory.
{}	*Curly Braces (123 & 125):* CALCULATE operation. For example: `totalnums += {num}`.

Caché MultiValue Basic Commands

ABORT, ABORTE, ABORTM

Terminates program execution and returns to MVBasic shell.

```
ABORT [errcode [,val1[,val2]]]
ABORTE [errcode [,val1[,val2]]]
ABORTM [message]
```

Arguments

errcode	*Optional* — A MultiValue error code; commonly (but not always) specified as a positive integer. The error code can be specified as a literal or as a expression that resolves to a literal value. A non-numeric literal value must be specified as a quoted string.
val	*Optional* — A comma-separated list of one or more literal values to insert into the error message corresponding to *errcode*. These insert values can be specified as literals or as expressions that resolves to a literal value. A non-numeric literal value must be specified as a quoted string.
message	*Optional* — An expression that resolves to a literal error message text, specified as a quoted string.

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. For a list of error codes and corresponding error messages, see Error Messages in the *Caché MultiValue Commands Reference*.
- **ABORTM** with an specified argument uses the literal *message* as the error message to display.
- **ABORT** in Caché MVBasic is functionally identical to **ABORTE**. Depending on the emulation setting, **ABORT** in other MultiValue emulations may be functionally identical to either **ABORTE** or **ABORTM**.

An abort operation resets the @LEVEL system variable to 0.

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.

During debugging, **STOP** terminates the debugging session. The debugger treats an **ABORT** as an error condition; the debugger performs a break operation to allow for examination of the condition causing the **ABORT**.

See Also

- BREAK statement
- ERRMSG statement
- STOP statement
- ObjectScript: QUIT command

ASSIGN

Assigns a value to the SYSTEM or STATUS functions.

```
ASSIGN value TO SYSTEM(code)
ASSIGN int TO STATUS()
```

Arguments

value	An expression that evaluates to a value. This may be an integer value or a string value, depending on the SYSTEM() function.
int	An expression that evaluates to an integer value. A non-integer values is truncated to an integer. A non-numeric value evaluates to 0.
code	An integer code specifying which SYSTEM code information to modify. For a list of codes, refer to the SYSTEM function.

Description

The **ASSIGN** statement is used either to assign an integer return value to the STATUS function, or to assign a return value to one of the SYSTEM function options. Assignments apply to the current process.

When assigning a **STATUS** function value, *value* must be a literal, variable, or arithmetic expression that resolves to a positive or negative integer. A fractional number is truncated to its integer portion. A string is truncated at the first non-numeric character. A non-numeric string resolves to the numeric value 0. If you exit and re-enter the MV Shell, the **STATUS** function value is reset to 0.

Most **SYSTEM** functions cannot be assigned a value using this command. **SYSTEM(2)**, **SYSTEM(3)**, and **SYSTEM(7)** can be assigned a value. Only a valid terminal type can be assigned to **SYSTEM(7)**. If you exit and re-enter the MV Shell, these **SYSTEM** function values persist until explicitly reset.

Examples

The following example reduces the terminal's page width setting by 10 characters:

```
pwidth=SYSTEM(2);   ! The old page width
PRINT pwidth
ASSIGN pwidth-10 TO SYSTEM(2)
PRINT SYSTEM(2);    ! The new page width
```

In the following example, the first **ASSIGN** sets the terminal (channel 0) page width to 20. The **PRINTER ON** statement changes channel 0 to the current printer. The second **ASSIGN** sets the printer (channel 0) page width to 40. The **PRINTER OFF** reverts channel 0 to the terminal, which now has a page width of 40:

```
EXECUTE "TERM"
ASSIGN 20 TO SYSTEM(2)
EXECUTE "TERM"
PRINTER ON
ASSIGN 40 TO SYSTEM(2)
PRINTER OFF
EXECUTE "TERM"
```

See Also

- STATUS function
- SYSTEM function

BEGIN TRANSACTION

Begins a transaction.

```
BEGIN TRANSACTION
```

Description

The **BEGIN TRANSACTION** statement initiates a transaction. A transaction is a block of code beginning with **BEGIN TRANSACTION** and ending with **END TRANSACTION**. All statements within the transaction are either applied as a unit by a **COMMIT** statement, or rolled back as a unit by a **ROLLBACK** statement. Following a **COMMIT** or **ROLLBACK**, 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 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 implications for systems that wish to scan files without taking locks.

Example

The following example performs database operations within a transaction. It sets a variable x, which determines whether the transaction should be committed or rolled back.

```
PRINT "Before the transaction"
BEGIN TRANSACTION
 .
 .
 .
IF x=0
  THEN COMMIT
  END
  ELSE ROLLBACK
  PRINT "Transaction rolled back"
  END
PRINT "This should not print"
END TRANSACTION
PRINT "After the transaction"
```

See Also

- END TRANSACTION statement
- COMMIT statement
- ROLLBACK statement

BREAK

Enables or disables keys that pause program execution.

```
BREAK [KEY] { ON | OFF }
BREAK [KEY] flag
```

Arguments

flag	An expression that evaluates to a boolean value. 0=disable break keys. 1 (or any non-zero number)=enable break keys.

Description

The **BREAK** statement is used to enable or disable terminal keys that can pause program execution. It can be executed using the ON or OFF keyword, or by using a boolean *flag* value. These two forms are functionally identical.

When **BREAK** is enabled (ON), the Interrupt, Suspend, and Quit keys will cause program execution to be suspended. When **BREAK** is disabled (OFF) these keys have no effect on program execution. The **BREAK** setting determines how **Ctrl-C** is handled when typed at the INPUT prompt.

The KEY keyword is optional and performs no function; it is provided for code compatibility only.

The MVBasic **BREAK** statement performs the same operation as the various MultiValue command line **BREAK** commands. Issuing any of these statements increments or decrements a counter. Thus multiple BREAK OFF statements (of any type) must be reversed by an equal number of BREAK ON statements.

Emulation

jBASE emulation supports an argumentless **BREAK** statement as a synonym for **EXIT**. Refer to the EXIT statement for details. In jBASE emulation, BREAK statements simply enable or disable (toggle) without maintaining a counter.

See Also

- ABORT statement
- INPUT statement
- STOP statement
- BREAK command in *Caché MultiValue Commands Reference*

BSCAN

Traverses the unique keys in an index, or the item ids in an inode-type file.

```
BSCAN keyvar [,recvar]
[FROM filevar[,startkey]]
[USING indexname] [RESET] [BY seq]
[THEN statements] [ELSE statements]
```

Arguments

keyvar	**BSCAN** assigns to *keyvar* the key or item id returned by the **BSCAN** operation.
recvar	*Optional* — If you specify a *recvar*, **BSCAN** assigns the contents associated with *keyvar* to it. This can be the list of item ids associated with the key returned in *keyvar*, or the contents of the record associated with the item id returned in *keyvar*.
FROM *filevar*	*Optional* — A local variable name assigned to the MultiValue file by the **OPEN** statement. If you do not specify *filevar*, the default file, specified in the system variable @STDFIL, is used.
startkey	*Optional* — An expression that specifies the relative starting position of the scan. *startkey* can be an index key or item id. If the USING *indexname* clause is used, *startkey* is a value in the specified index.
USING *indexname*	*Optional* — The name of a secondary index associated with the file.
BY *seq*	*Optional* — Specifies the direction of the scan. The available *seq* options are “A” (ascending) and “D” (descending). The default is ascending.

Description

The **BSCAN** statement operates in 2 modes:

- **BSCAN** with an *indexname* steps through the unique keys in an index. The keys are returned as *keyvar*. It optionally returns the item id associated with *keyvar* as *recvar*.
- **BSCAN** without an *indexname* steps through the item ids in an inode-type file. The list of item ids is returned as *keyvar*. It optionally returns the contents of each *keyvar* item as *recvar*.

The **BSCAN** statement scans the leaf nodes of either a B-tree file (type 25) or a secondary index. The record ID returned by the scan operation is assigned to *keyvar*. If you specify a *recvar*, **BSCAN** assigns the contents of the *keyvar* record to it.

filevar specifies an open file. If you do not specify *filevar*, the default file is used. (For more information on default files, see the OPEN statement.) If the specified file is neither accessible nor open, **BSCAN** returns nothing and sets STATUS() to 3.

startkey is an expression that evaluates to a record ID of a record in the B-tree file. If the USING clause is used, *startkey* is a value in the specified index. *startkey* specifies the relative starting position of the scan.

startkey need not exactly match an existing record ID or index key. If it does not, the scan finds the next or previous record ID or value, depending on whether the scan is in ascending or descending order. For example, depending on how precisely you want to specify the starting point at or near the record ID or value SMITH, *startkey* can evaluate to SMITH, SMIT, SMI, SM, or S.

If you do not specify *startkey*, on the initial **BSCAN** operation, the scan starts at the beginning (leftmost slot of the leftmost leaf) or end (rightmost slot of the rightmost leaf) of the index or file, depending on the value of the *seq* expression. The

scan then moves in the direction specified in the BY clause. Subsequent **BSCAN** operations with no *startkey* specified will continue from the *keyvar* returned by the previous **BSCAN**.

indexname is an expression that evaluates to the name of a secondary index associated with the file.

RESET resets the internal scan pointer to the first or last key, depending on the BY *seq* clause value. If you do not specify *seq*, the scan is done in ascending order. If you specify *startkey* in the FROM clause, RESET is ignored.

seq is an expression that evaluates to A or D; it specifies the direction of the scan. "A", the default, specifies ascending order. "D" specifies descending order.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If the **BSCAN** statement finds a valid index key, or item id and its associated data, the THEN clause is executed. If the scan does not find a valid index key, or if some other error occurs, the ELSE clause is executed. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

Any file updates executed in a transaction (that is, between a BEGIN TRANSACTION statement and a COMMIT statement) are not accessible to the **BSCAN** statement until after the COMMIT statement has been executed.

Note: Cache supports the **BSCAN** statement for compatibility with legacy MultiValue systems. When retrieving keys from an index, developers should use **OPENINDEX** and **SELECT** with the ATKEY clause, because SELECT ATKEY has a simpler, more intuitive syntax and superior performance.

STATUS Values

The **STATUS** function returns the following values after the **BSCAN** statement is executed:

If NLS is enabled, the **BSCAN** statement retrieves record IDs in the order determined by the active collation locale; otherwise, **BSCAN** uses the default order, which is simple byte ordering that uses the standard binary value for characters; the Collate convention as specified in the NLS.LC.COLLATE file for the current locale is ignored.

0	The scan proceeded beyond the first or last key. *keyvar* and *recvar* are set to empty strings.
1	The scan returned an existing index key, or an index key that matches the key specified by *startkey*.
2	The scan returned an index key that does not match *startkey*. *keyvar* is either the next or the previous record ID in the B-tree, depending on the direction of the scan.
3	*filevar* is not open, or is not an inode-type.
4	*indexname* does not exist.
5	*seq* does not evaluate to A or D.
6	The index specified by *indexname* needs to be built.
10	An internal error was detected.

Examples

The following example demonstrates using **BSCAN** to step through the keys in an index:

```
0001 EXECUTE 'CREATE-FILE DATE-FILE'
0002 OPEN 'DICT','DATE-FILE' TO DICT.DATE.FILE ELSE STOP 201,'DICT DATE-FILE'
0003 WRITE 'D':@AM:1 ON DICT.DATE.FILE,'DAY.OF.WEEK'
0004 EXECUTE 'CREATE-INDEX DATE-FILE DAY.OF.WEEK'
0005 OPEN 'DATE-FILE' TO DATE.FILE ELSE STOP 201,'DATE-FILE'
0006 FOR I=1 TO 21
0007   ID=DATE()+I
0008   WRITE OCONV(ID,'DWA') ON DATE.FILE,ID
0009 NEXT I
0010 LOOP
0011   BSCAN DOW,IDLIST FROM DATE.FILE USING 'DAY.OF.WEEK' ELSE EXIT
0012   CRT 'DOW=':DOW,' DATES=':IDLIST
0013 REPEAT
```

This returns the following output:

```
[421] DICT for file 'DATE-FILE' created. Type = INODE
[418] Default data section for file 'DATE-FILE' created. Type = INODE
[437] Added default record '@ID' to 'DICT DATE-FILE'.
[417] CreateFile Completed.
DOW=FRIDAY      DATES=14642 14649 14656
DOW=MONDAY      DATES=14645 14652 14659
DOW=SATURDAY    DATES=14643 14650 14657
DOW=SUNDAY      DATES=14644 14651 14658
DOW=THURSDAY    DATES=14641 14648 14655
DOW=TUESDAY     DATES=14646 14653 14660
DOW=WEDNESDAY   DATES=14640 14647 14654
```

Notice that on each iteration, **BSCAN** returns the next unique key in the index. The item ids associated with the key are returned as an @AM-delimited list in the optional *recvar* argument. If you want to process each record for the key, you need to code a loop to do so.

The following example demonstrates using **BSCAN** to retrieve the keys for a particular key:

```
0001 OPEN 'DATE-FILE' TO DATE.FILE ELSE STOP 201,'DATE-FILE'
0002 BSCAN DOW FROM DATE.FILE,'SUN' USING 'DAY.OF.WEEK' ELSE NULL
0003 CRT 'DOW=':DOW,' STATUS=':STATUS()
0004 BSCAN DOW FROM DATE.FILE USING 'DAY.OF.WEEK' ELSE NULL
0005 CRT 'DOW=':DOW,' STATUS=':STATUS()
0006 BSCAN DOW FROM DATE.FILE,'Z' USING 'DAY.OF.WEEK' ELSE NULL
0007 CRT 'DOW=':DOW,' STATUS=':STATUS()
```

This returns the following output:

```
DOW=SUNDAY    STATUS=2
DOW=THURSDAY  STATUS=1
DOW=  STATUS=0
```

On line 2 we request key SUN. There is no SUN, so **BSCAN** returns the next key SUNDAY. The **BSCAN** on line 4 doesn't specify a start key, so the next key. THURSDAY, is returned. On line 6, we request key Z. There is no Z and nothing after Z, so **BSCAN** returns status 0

In the following example **BSCAN** is used to scan the item ids of an inode-type file. In this example, we look for item id SEL in VOC. SEL does not exist, so **BSCAN** returns the next id SEARCH:

```
0001 OPEN 'VOC' TO VOC ELSE STOP 201,'VOC'
0002 BSCAN ID,ITEM FROM VOC,'SEL' BY 'D' ELSE NULL
0003 CRT 'ID=':ID,' ITEM=':ITEM
0004
MINE:TRY
ID=SEARCH        ITEM=V SEARCH C  2C
```

See Also

- OPEN statement
- OPENINDEX statement
- SELECT ATKEY statement
- STATUS function

CALL

Transfers control to an external subroutine.

```
CALL routine[(arglist)]
```

Arguments

routine	Name of the external subroutine to call.
arglist	*Optional* — 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 *routine* to specify the external subroutine either directly or indirectly:

- The *routine* argument can specify the exact name under which the subroutine was cataloged.
- The *routine* 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.

  If the *routine* name begins with an asterisk (*), **CALL** first looks it up as a local routine. If not found, **CALL** looks it up as a global routine. If still not found, **CALL** generates an error. Note that **routine* processing is different in UniData emulation, as described below.

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 *arglist*, 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 *arglist* 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 works on only a single value at a time. If you specify a **CALL** with a multivalue argument, Caché MVBasic invokes **CALL** repeatedly, once for each value in the multivalue argument. The called external subroutine can only return single-valued arguments.

CALL, ENTER, SUBR, and GOSUB

The **CALL** statement is used to call an external subroutine with parameter passing and return. If you do not need to pass parameters or return to the calling program, you can use **ENTER** to call an external subroutine.

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

Examples

The following example uses **CALL** to pass arguments by reference:

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

MapSub(name,newname)
  PRINT name        ! Returns "Burma"
  name=newname
  PRINT name        ! Returns "Myanmar"
  RETURN
```

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

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

MapSub(name,newname)
  PRINT name        ! Returns "Burma"
  name=newname
  PRINT name        ! Returns "Myanmar"
  RETURN
```

Emulation

In UniData and UDPICK emulations, a *routine* name with an initial character of * is handled as a global routine name. **CALL** removes the leading * and then looks up the resulting routine name as a global routine. If the runtime environment is not UniData emulation, a normal lookup is done on a *routine* name with a leading * character.

The use of `$OPTIONS UNIDATA` in the MVBasic source file *does not* activate this behavior. The handling of names with leading * is determined by the user setting in the command language at runtime. Therefore, to activate this behavior, the CEMU command must set UniData emulation before running a program that calls a *routine* name with a leading *.

See Also

- ENTER statement
- 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

expression	A value, variable, or expression that evaluates to a boolean value: FALSE = 0, TRUE=1 or any numeric value other than 0.
statement	One or more MVBasic statements to execute if the corresponding *expression* evaluates to TRUE.

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: `CASE 1`. The statements associated with this clause will be executed if all the other **CASE** clauses evaluate to false (0).

You cannot use a **GOTO** statement to transfer execution within a **CASE** statement.

CASE statements can be nested. You can use a **GOTO** statement to transfer execution from a **CASE** clause to a nested a **CASE** statement.

A placeholder **CASE** block, consisting of just the **BEGIN CASE** and **END CASE** statements, is supported.

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. **CASE** expressions are evaluated in the order specified; therefore, an error in an *expression* (for example, a divide-by-zero error: `CASE var/0`) is not detected if a prior *expression* is taken.

statement

One or more statements executed if *expression* evaluates to true. If *expression* does not evaluate to true, *statement* is not parsed.

Examples

The following example takes a user input and executes one of the specified cases based on length of the input string. The final case (CASE 1) is always true. This provides a case that is always taken if all of the previous cases did not evaluate to true:

```
INPUT myword
BEGIN CASE
   CASE 5 > LEN(myword)
     CRT "short"
     CRT "word"
   CASE 5 < LEN(myword)
     CRT "long"
     CRT "word"
   CASE 1
     CRT "five letter"
     CRT "word"
END CASE
CRT "all done"
```

The following example shows nested **CASE** statements. It shows how a **GOTO** can be used to transfer execution to a nested **CASE** statement:

```
INPUT myword
BEGIN CASE
   CASE 5 > LEN(myword)
      CRT "short word"
Atest:
      BEGIN CASE
        CASE 1 = COUNT(myword,"A")
          CRT "contains one A"
        CASE 1 < COUNT(myword,"A")
          CRT "contains more than one A"
        CASE 1
          CRT "contains no A"
      END CASE
   CASE 5 < LEN(myword)
      CRT "long word"
      GOTO Atest:
   CASE 1
      CRT "five letter word"
      GOTO Atest:
END CASE
```

See Also

- IF...THEN...ELSE statement

CATCH

Identifies a block of code to execute when an exception occurs.

```
TRY
   statements
CATCH [exceptionvar]
   statements
END TRY
```

Arguments

exceptionvar	*Optional* — An exception variable. Specified as a local variable, with or without subscripts, that receives a Caché Object reference (oref).

Description

The **CATCH** statement defines an exception handler, one or more statements to execute when an exception occurs in the code following a **TRY** statement. The **CATCH** statement is followed by one or more exception handling code statements. The **CATCH** block must immediately follow its **TRY**, and the paired **TRY** and **CATCH** are terminated by an **END TRY** statement.

The **CATCH** statement has two forms:

- Without an argument
- With an argument

CATCH without an Argument

Argumentless **CATCH** is invoked when an exception occurs in the **TRY** block. This executes the series of statements between **CATCH** and **END TRY**.

CATCH with an Argument

CATCH *exceptionvar* is invoked when an exception occurs in the **TRY** block. This exception passes *exceptionvar* to the **CATCH** block. This exception can either be explicitly invoked by a **THROW** statement, or issued by the system runtime environment in the event of a system exception. The *exceptionvar* Caché Object reference (oref) provides properties that contain information about the exception, such as the Name of the error and the Location where it occurred. The user-written **CATCH** exception handler code can use this information to analyze the exception.

Arguments

exceptionvar

A local variable, used to receive the exception object reference from the **THROW** statement or from the system runtime environment in the event of a system exception. When a system exception occurs, *exceptionvar* receives a reference to an object of type %Exception.SystemException. For further details, refer to the %Exception.AbstractException class in the *InterSystems Class Reference*.

Examples

The following example shows a **CATCH** invoked by a runtime exception. The *myvar* argument receives a system-generated exception object:

```
TRY
  PRINT "about to divide by zero"
  a=7/0
  PRINT "this should not display"
CATCH myvar
  PRINT "this is the exception handler"
  PRINT :myvar->Name,"Error Name"
  PRINT :myvar->Code,"Error Code Number"
  PRINT :myvar->Location,"Error Location"
END TRY
PRINT "this is where the code falls through"
```

See Also

- THROW statement
- TRY statement

CHAIN

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

```
CHAIN command
```

Arguments

command	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. It initially searches the VOC for the *command*; if the *command* is not found in the VOC, it searches the global catalog. For lookup details, refer to CATALOG in the *Caché MultiValue Commands Reference*.

CHAIN does not create a new execution environment. Therefore any select lists that were active when **CHAIN** was invoked are retained as the active select lists of the invoked *command*.

CHAIN cannot pass values to *command*. Because **CHAIN** does not return to the invoking program, it cannot pass a return value from *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 in the invoking program. **EXECUTE** creates a new execution environment; select lists that were active when **EXECUTE** was invoked are not retained by its invoked MultiValue commands. **EXECUTE** can explicitly 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
```

Emulation

In jBASE emulation, **CHAIN** does not pass the default select list (select list 0) to the invoked program.

In UniData and UDPICK emulations, a *command* name with an initial character of * is handled as a global name. **CHAIN** removes the leading * and then looks up the resulting command name in the global catalog in SYS.MV, rather than looking up in the VOC. If the runtime environment is not a UniData emulation, a normal VOC lookup is done on the **command* name.

See Also

- EXECUTE statement
- PERFORM statement
- ObjectScript: XECUTE command

CHANGE

Replaces all instances of a substring in a variable.

```
CHANGE oldstring TO newstring IN variable
```

Arguments

oldstring	The substring to be replaced. An expression that resolves to a valid string or numeric.
newstring	The replacement substring. An expression that resolves to a valid string or numeric. To delete *oldstring*, specify the empty string ("").
variable	An existing variable containing a string value. *variable* may be a dynamic array. *variable* accepts a single dynamic array reference (A<i>), a single substring reference (A[s,l]), or a substring reference nested inside a dynamic array reference (A<i>[s,l]).

Description

The **CHANGE** statement edits the value of *variable* by replacing all instances of *oldstring* with *newstring*. The *oldstring* and *newstring* values may be of different lengths. Matching of strings is case-sensitive. If *oldstring* is not present in the variable, no operation is performed.

The values of *oldstring* and *newstring* 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 string replacement.

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

CHANGE and **SWAP** both perform string substitution, and are functionally identical. **CONVERT** performs character-for-character substitution.

Examples

In following example, **CHANGE** replaces every instance of “?” with “[PLEASE CHECK]” in global variable ^mytest:

```
^mytest="Jones":@VM:"?":@VM:"? Water St.":@VM:"Springfield":@VM:"?"
CHANGE "?" TO "[PLEASE CHECK]" IN ^mytest
PRINT ^mytest
```

See Also

- CONVERT statement
- SWAP statement
- CONVERT function
- CHANGE function
- Strings

CLEAR

Resets variables not assigned to a common storage area.

```
CLEAR
```

Arguments

The **CLEAR** statement takes no arguments.

Description

The **CLEAR** statement clears (sets to 0) all local variables that are not assigned to a common storage area. Variables in a named common storage area or in the unnamed common storage area are unaffected.

Because **CLEAR** sets to 0 both assigned and unassigned variables, it can be usefully invoked at the beginning of a program to prevent problems caused by unassigned variables.

You can use the **COMMON** statement to assign variables to a common storage area. You can use the **CLEAR COMMON** statement to clear (reset to 0) all local variables that are assigned to a named common storage area or to the unnamed common storage area.

See Also

- COMMON statement
- CLEARCOMMON statement

CLEARCOM (CLEARCOMMON)

Resets variables assigned to a common storage area.

```
CLEARCOM [/store/]
CLEARCOMMON [/store/]
CLEAR COM [/store/]
CLEAR COMMON [/store/]
```

Arguments

store	*Optional* — A named common storage area for a group of variables. If specified, this name is enclosed with slashes (/). The default is the unnamed common area.

Description

The **CLEARCOMMON** statement resets all of the variables stored in the common storage area, assigning them the value "0". The **COMMON** statement allows you to assign a list of local variables to a common storage area. These variables do not have to be defined to be listed in a common storage area.

The **COMMON** statement can define a *store* name for a named common storage area. If **COMMON** omits *store*, the named variables are stored in the unnamed common storage area. **CLEARCOMMON** can reset the variables in a named common storage area, or omit *store* and reset the variables in the unnamed common storage area.

CLEARCOM, **CLEARCOMMON**, **CLEAR COM**, and **CLEAR COMMON** are all equivalent syntactical forms for this statement.

You can use the **CLEAR** statement to clear (reset to 0) all local variables that are not assigned to a common storage area.

See Also

- CLEAR statement
- COMMON statement

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 [SETTING var]
   [ON ERROR statements] [LOCKED statements]
```

Arguments

filevar	A file variable name used to refer to a MultiValue file. This *filevar* value is supplied by the **OPEN** statement.
SETTING *var*	*Optional* — When an error occurs, sets the local variable *var* to the operating system's error return code. Successful completion returns 0; error return codes are platform-specific. The SETTING clause is executed before the ON ERROR clause. Provided for jBASE compatibility.

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 a LOCKED clause, which is executed if **CLEARFILE** could not delete all records due to lock contention. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the LOCKED keyword and the first line.

You can optionally specify an ON ERROR clause. If the data deletion fails (for example, the file could not be accessed), the ON ERROR clause is executed. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

See Also

- OPEN statement
- DELETE statement
- STATUS statement
- STATUS function

CLEARINPUT

Clears input data from the type-ahead buffer.

```
CLEARINPUT
```

Arguments

None.

Description

The **CLEARINPUT** statement deletes (clears) any user input data stored in the type-ahead buffer. This affects the **INPUTIF** statement, which receives user input from the type-ahead buffer. **CLEARINPUT** has no effect on the **INPUT** statement, which does not use a type-ahead buffer.

The **CLEARINPUT** and **INPUTCLEAR** statements are functionally identical. **CLEARINPUT** is supported for compatibility with UniData systems.

See Also

- INPUTIF statement
- INPUTCLEAR statement

CLEARSELECT

Resets active select lists.

```
CLEARSELECT [selectlist]
CLEARSELECT ALL
```

Arguments

selectlist	*Optional* — An identifier assigned to an active select list, specified as an integer from 0 through 10 (inclusive), or a named select list variable. If omitted, select list 0 is cleared.

Description

The **CLEARSELECT** statement resets an active select list. It has three syntactical forms:

- **CLEARSELECT selectlist** resets the specified select list.
- **CLEARSELECT** resets select list 0.
- **CLEARSELECT ALL** resets all active numbered select lists. It has no effect on named select lists.

Emulation

By default, **SELECT** uses select list 0 as the default select list for both internal and external use. Reality, D3, R83, POWER95, MVBase, and IN2 systems use two distinct default select lists, one internal and one external. This behavior can be set using **$OPTIONS PICK.SELECT**. When this option is set, the default external select list is 0, and the default internal select list is 10.

See Also

- SELECT statement

CLOSE

Closes a MultiValue file.

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

Arguments

filevar	A file variable name used to refer to a MultiValue file. This *filevar* is supplied by the **OPEN** statement.

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. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

Alternatively, you can use the **STATUS** function to determine the status of the file close operation, as follows: 0=success; -1=file does not exist or has already been closed.

See Also

- OPEN statement
- STATUS function

CLOSESEQ

Closes a file opened for sequential access.

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

Arguments

filevar	A file variable name used to refer to the file in Caché MVBasic. This *filevar* 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. This may occur if the file is already closed. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

See Also

- OPENSEQ statement
- STATUS function

COMMIT

Commits all changes made during the current transaction.

```
COMMIT [TRANSACTION | 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 **COMMIT** must be specified between the **BEGIN TRANSACTION** and **END TRANSACTION** statements. Following a **COMMIT**, program execution skips to the line of code following the **END TRANSACTION** statement.

The TRANSACTION or WORK keywords are optional and provides no functionality. They are 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. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

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.

Locks and Transactions

File locks and record locks that were taken out during a transaction are released at the end of a transaction. If there are nested transactions, the release of locks taken out during the inner transactions is delayed until the completion of the outermost transaction. This release of locks is part of a successful **COMMIT** or **ROLLBACK** operation. Locks are described in the LOCK statement.

Example

The following example performs database operations within a transaction. It sets a variable x, which determines whether the transaction should be committed or rolled back.

```
PRINT "Before the transaction"
BEGIN TRANSACTION
 .
 .
 .
IF x=0
  THEN COMMIT
    THEN PRINT "Commit successful"
    ELSE PRINT "Commit failed"
  END
  ELSE ROLLBACK
  END
PRINT "This should not print"
END TRANSACTION
PRINT "Transaction resolved"
```

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

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

Description

The **COMMON** statement allows you to specify list of local variables that are placed in a common storage area 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. A variable placed in a common storage area may contain a literal value or an object reference.

You can use *store* to specify a named common storage area, or omit this argument and store the listed variables in the unnamed common storage area. A *store* name can be of any length, but it suggested that it be unique within its first 27 characters.

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.

Specifying an array in a **COMMON** statement dimensions that array; it cannot be subsequently dimensioned using a **DIM** statement. Attempting to do so results in a compile error.

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.

You can use the **CLEARCOMMON** statement to reset all of the variables in held in a common storage area.

Emulation

The **COMMON** initialization of array variables for Caché MVBasic is UNASSIGNED, for both named and unnamed common storage areas. Other supported MultiValue emulations provide differing initialization for array variables in named and unnamed common storage areas. Scalar variables are always initialized as UNASSIGNED in all emulations.

Examples

The following example initializes an array variable in the unnamed common storage area, then tests whether the variable is assigned. In native Caché MVBasic the result will always be unassigned; other MultiValue emulations return other results.

```
COMMON c(3)
IF ASSIGNED(c(3)) THEN PRINT c(3)
ELSE PRINT "Unassigned for unnamed storage"
```

The following example initializes an array variable in a named common storage area, then tests whether the variable is assigned. In native Caché MVBasic the result will always be unassigned; other MultiValue emulations return other results.

```
COMMON /ABC/ y(2)
IF ASSIGNED(y(2)) THEN PRINT y(2)
ELSE PRINT "Unassigned  for named storage"
```

See Also

- CALL statement
- CLEARCOMMON statement
- DIM statement
- SUBROUTINE statement
- ASSIGNED function
- UNASSIGNED function

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.

Example

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"
NEXT
PRINT "all done"
```

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

charsout	One or more characters to be replaced. Any expression that resolves to a valid string or numeric.
charsin	The character or characters to be inserted in place of the corresponding characters in *charsout*. Any expression that resolves to a valid string or numeric.
string	The string in which character substitutions are made. An expression that resolves to a valid string. *string* may be a dynamic array. *string* accepts a single dynamic array reference (A<i>), a single substring reference (A[s,l]), or a substring reference nested inside a dynamic array reference (A<i>[s,l]).

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

CREATE

Creates a file for sequential access.

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

Arguments

filevar	A file variable name used to refer to the file in Caché MVBasic. This *filevar* 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 *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

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 terminal screen.

```
CRT [text]
CRT text [format]
```

Arguments

text	*Optional* — Any MVBasic expression that resolves to a quoted string or a numeric. You can specify a single expression or a series of expressions separated by either commas (,) or colons (:). A comma inserts a tab spacing between the two strings. A colon concatenates the two strings. If *text* is omitted, a blank line is returned.
format	*Optional* — A code specifying how to handle *text*, specified as a quoted string. This *format* is applied to the *text* that immediately precedes it. Whitespace characters may be inserted between *text* and *format*.

Description

CRT displays one or more *text* items on the terminal screen. This *text* can consist of any number of text strings separated by commas or colons. Any *text* may be followed by an optional *format*. This *format* applies only to the *text* string that immediately precedes it.

CRT does not send its output to an open PRINTER channel, which allows **CRT** to be executed without using PRINTER OFF and PRINTER ON.

A *text* can consist of a single string or numeric expression, or a series of expressions alternating with separator characters. If no *text* is specified, **CRT** returns a blank line.

The following separators are supported:

- A comma (,) used as a separator character inserts a predefined tab between to items. By default, tabs are set at ten column intervals. You can specify a comma before the first expression to indent that expression. You cannot specify a comma after the last expression; this results in a syntax error. You can specify a series of commas to specify multiple tabs; an odd number of commas increments the number of tabs. Thus, one or two commas (`exp,exp` or `exp,,exp`) equals one tab, three or four commas (`exp,,,exp` or `exp,,,,exp`) equals two tabs, and so forth.
- A colon (:) used as a separator character concatenates two items. Specifying a colon before the first expression has no effect. Specifying a colon after the last expression enables concatenation of the results of two commands. 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.

The **DISPLAY** and **CRT** commands are identical. The **PRINT** command is similar to **CRT**, but provides additional functionality.

Formatting

The optional *format* argument specifies how to handle *text*. **CRT** supports three types of *format* arguments:

- @ function formatting
- implicit formatting, using FMT function codes
- implicit conversion, using OCONV function codes

You can use an @ function with positive arguments to specify the column position and/or line position at which to print. For example, `CRT @(15):"Over here!"` prints the literal string starting at column 16. You can also use the @ function with negative arguments to change screen display modes. For example, `CRT @(-1):"Over here!"` clears the screen, then prints the literal string at line 1, column 1.

You can use the optional *format* argument to specify display width, justification, fill characters, and zero filling or rounding for decimal digits. This is known as “implicit formatting” because it is equivalent to inserting a **FMT** function as one of the **CRT** arguments. For further details on the available *format* codes, refer to the FMT function.

You can disable implicit formatting by specifying `$OPTIONS NO.IMPLICIT.FMT`. Specifying this option prevents the evaluation of the *format* argument in CRT, PRINT, or DISPLAY. It has no effect on the explicit use of the **FMT** function.

Implicit conversion performs many of the OCONV function conversions by specifying the conversion code as the *format* argument. For example, both of the following perform date conversion from internal to display format:

```
CRT 14100 "D";          ! "08 AUG 2006"
CRT OCONV(14100,"D");   ! "08 AUG 2006"
```

For further details on the available *format* conversion codes, refer to the OCONV function.

Examples

The following examples illustrate the use of the **CRT** command:

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

returns:

```
hello    world!
```

See Also

- DISPLAY statement
- PRINT statement
- @ function
- FMT function
- SPACE function

DATA

Provides user input data.

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

Arguments

exp	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 and issues a prompt. From this point you can issue debug commands, including returning to the execution of the interrupted program.

By default, a command issued at the debug prompt is an ObjectScript command. To issue a Caché MVBasic statement at the debug prompt, you must prefix a semicolon to the command. This is shown in the following terminal example:

```
USER:;myvar="ABC"
USER:;DEBUG

<BREAK>+1^MVBASIC1048.mvi
Source Id: File: Line:0
USER 7d1>WRITE "my variable=",myvar
my variable=ABC
USER 7d1>;CRT "my variable",myvar
my variable     ABC
USER 7d1>
```

The ;/ Statement

At the debug prompt, you can use the ;/ statement to display the contents of a variable. The variable may be subscripted. The syntax is as follows:

```
;/varname
```

The ;/ statement returns *varname=value*. It can be used to display the value of a local variable, an array dimensioned with DIM, a variable defined using EQUATE, or a variable defined in a COMMON statement.

The ;/ statement can return **COMMON** variables that are defined in different accounts (namespaces).

See Also

- ; (semicolon) command in the *MultiValue Commands Reference*

DEFFUN

Declares a user-defined function

```
DEFFUN name [(args)] [CALLING routine]
```

Arguments

name	The name of an existing user-defined function. This cannot be the name of any existing system-provided (built-in) function. Name validation is performed on *name*.
args	*Optional* — An argument, or comma-separated list of arguments for the function. Arguments can be subscripted. If one or more arguments are specified, the enclosing parentheses are mandatory.
CALLING *routine*	*Optional* — Used to map a identifier to a valid *name*. Either an identifier, or a quoted string literal that begins with the asterisk (*) character. No name validation is performed on *routine*.

Description

The **DEFFUN** statement allows you to declare an existing user-defined function, placing it in the function lookup table. This operation only declares the function's name and argument list. Prior to invoking **DEFFUN**, the function must have been defined, using the **FUNCTION** statement.

The CALLING clause is commonly used to map an invalid function name (*routine*) to a valid user-defined function name (*name*). User-defined function names cannot begin with a punctuation character (except %); built-in function names often begin with a punctuation character. You can use a CALLING clause to map one to the other. In the optional CALLING clause, the *routine* name can be a quoted string literal function name beginning with an asterisk (*), as follows:

```
DEFFUN foo(x,y,z) CALLING "*foo"
```

In this example, **DEFFUN** allows calls to appear in expressions using the ordinary identifier *foo*, while the name **foo* is passed to the runtime execution. The leading asterisk specifies how to look up this function name. In Caché MultiValue and most emulations, the asterisk is both part of the function name and an indicator specifying how to look up this function. In UniData emulation, the asterisk is removed from the function name and serves only as a lookup indicator. For further details, refer to the CALL statement.

Examples

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

```
DEFFUN cuberoot(mynum,precision)
CRT cuberoot(mynum,precision)
```

See Also

- CALL statement
- FUNCTION statement

DEL

Deletes an element from a dynamic array.

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

Arguments

dynarray	Any valid dynamic array.
f	An integer specifying the Field (attribute) level of the dynamic array on which to perform the deletion. Fields/Attributes are counted from 1.
v	*Optional* — An integer specifying the Value level of the dynamic array on which to perform the deletion. Values are counted from 1 within a Field.
s	*Optional* — 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 YorkýLondonýChicagoýBostonýLos Angeles"
DEL cities <1,2>
PRINT cities
    ! Returns: "New YorkýChicagoýBostonýLos Angeles"
```

See Also

- COUNTS function
- DELETE function
- EXTRACT function
- Dynamic Arrays

DELETE, DELETEU

Deletes a record from a MultiValue file.

```
DELETE filevar,recID
   [SETTING var] [LOCKED statements] [ON ERROR statements] [THEN statements] [ELSE
statements]

DELETEU filevar,recID
   [SETTING var] [LOCKED statements] [ON ERROR statements] [THEN statements] [ELSE
statements]
```

Arguments

filevar	A local variable used as the file identifier of an open MultiValue file. This variable is set by the **OPEN** statement.
recID	The record ID of the record to be deleted.
SETTING *var*	*Optional* — When an error occurs, sets the local variable *var* to the operating system's error return code. Successful completion returns 0; error return codes are platform-specific. The SETTING clause is executed before the ON ERROR clause. Provided for jBASE compatibility.

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.

You can optionally specify a LOCKED clause, which is executed if **DELETE** or **DELETEU** could not delete the specified record due to lock contention. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the LOCKED keyword and the first line.

You can optionally specify an ON ERROR clause. If record delete fails, the ON ERROR clause is executed. This may occur if the *filevar* file has already been closed. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If the record delete is successful, the THEN clause is executed. If record delete is attempted but fails, the ELSE clause is executed. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

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
- STATUS function

DELETELIST

Deletes a saved select list.

```
DELETELIST listname
```

Arguments

listname	A name assigned to a saved select list.

Description

The **DELETELIST** statement deletes a saved select list. The select list was saved using **WRITELIST**.

The *listname* select list is saved in the &SAVEDLISTS& file. Caché stores this file using the ^SAVEDLISTS global.

See Also

- WRITELIST statement

DELETESEQ

Deletes a sequential file.

```
DELETESEQ filename [SETTING setvar] [LOCKED statements]
[ON ERROR statements] [THEN statements] [ELSE statements]
```

Arguments

filename	The file to be deleted. A fully-qualified Windows or UNIX® file pathname, specified as a quoted string. For two-part versions of this argument, see the Emulation section below.
SETTING *setvar*	A variable used to hold the system return code. Because this comes from the underlying operating system, values are platform-dependent. However, all supported platforms return 0 for successful completion. The SETTING clause is executed before the ON ERROR, THEN, or ELSE clause.

Description

The **DELETESEQ** statement is used to delete a sequential access file.

The *filename* must be a fully-qualified pathname. The directories specified in *filename* must exist for a file delete to be successful. File names are not case-sensitive.

You can optionally specify a LOCKED clause, which is executed if **DELETESEQ** could not delete the specified sequential access file due to lock contention. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the LOCKED keyword and the first line.

You can optionally specify an ON ERROR clause, which is executed if the file is located but could not be deleted. If no ON ERROR clause is present, the ELSE clause is taken for this type of error condition. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If the file delete is successful, the THEN clause is executed. If file delete fails (for example, the file does not exist), the ELSE clause is executed. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

You can use the **STATUS** function to determine the status of the sequential file delete operation, as follows: 0=success; 1=file does not exist; 2=path does not exist; 3=access denied; 4=the file is a directory; 5=the file is locked by another MV process; 6=the file is in use; -1=unexpected error; -2=delete failed for platform-dependent reason, see *setvar* for further explanation.

File Locking

Issuing **OPENSEQ** gives a process exclusive access to the specified file. An **OPENSEQ** locks the file against a **DELETESEQ** issued by any other process. This lock persists until the process that opened the file releases the lock, by issuing a **CLOSE**, a **CLOSESEQ**, or a **RELEASE** statement.

Emulation

For jBASE emulation, the *filename* argument can be specified with a two-part *path*,*filename* syntax. When executed, the two parts are concatenated together, with a delimiter added to the end of *path*, when necessary. For example, `DELETESEQ 'c:\temp\','mytest.txt'` or `DELETESEQ 'c:\temp','mytest.txt'`.

For other emulation modes, the *filename* argument can be specified with a two-part *file*,*itemID* syntax. The *file* part is a dir-type file defined in the VOC master dictionary, and the *itemID* part is an operating system file within that directory.

See Also

- CREATE statement
- OPENSEQ statement
- READSEQ statement
- WRITESEQ statement
- FLUSH statement
- NOBUF statement
- CLOSESEQ statement
- RELEASE statement
- STATUS statement
- FILEINFO function
- STATUS function
- @FILENAME system variable

DIM (DIMENSION)

Dimensions an array of variables.

```
DIM array([rows[,columns]])[,array2([rows[,columns]])[,...]
DIMENSION array([rows[,columns]])[,array2([rows[,columns]])[,...]
```

Arguments

array	Name of an array. Follows standard variable naming conventions. Can be a single array or a comma-separated list of arrays.
rows	*Optional* — A positive, non-zero integer specifying the number of array elements to dimension for a one-dimensional (vector) array, or the number of rows to dimension for a two-dimensional array. Maximum value is 65535. A value less than 1 or greater than 65535 results in an <ARRAY DIMENSION> error.
columns	*Optional* — For two-dimensional (matrix) arrays, a positive, non-zero integer specifying the number of columns per row. Can only be used in conjunction with the *rows* argument. Maximum value is 65535. A value less than 1 or greater than 65535 results in an <ARRAY DIMENSION> error.

Description

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

The **DIM** statement can be used in two ways: explicitly, to dimension a one-dimensional or two-dimensional array, or implicitly to dimension a multidimensional array.

- Most MultiValue systems require you to explicitly dimension the *rows* and *columns* of a static array. These values specify the maximum number of elements that can be defined for that array. An explicitly dimensioned array is limited to two subscripts. It can be either one-dimensional, representing a vector array, or two-dimensional, representing the rows and columns of a matrix array. A one-dimensional array can be dimensioned either as a vector array: `DIM arrayname(n)` or a matrix array with a column dimension of 1: `DIM arrayname(n,1)`.
- Caché MVBasic also allows you to dimension arrays of an arbitrary number of dimensions. This allows MVBasic to support the multidimensional arrays used in Caché. You specify a multidimensional array using a DIM statement with empty parentheses: `DIM arrayname()`. This declares *arrayname* as a dimensioned array, but the number of dimensions and number of elements in each dimension may be expanded dynamically at runtime.

If a subroutine or function uses a static array (for example, `DIM myarray(2)`), the static array must be dimensioned within the subroutine or function. However, if a subroutine or function uses an array of unspecified dimensions (for example, `DIM myarray()`), you may specify the **DIM** either within or outside the subroutine or function.

The subscripts of a dimensioned array can be specified using named variables, as well as numeric indices. Variables whose names begin with a % are known as public arrays and their values are preserved across SUBROUTINE calls in a similar manner to COMMON arrays. Variables whose names 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.

To clear data from an implicitly dimensioned array, use $kill. This clears any values that have been assigned.

Note: When executing a DIM statement from the MVBasic command shell, you must assign and use the array elements within the same command line. For example:

```
USER:;DIM x(),y() ;x(1)="fred" ;y(2)="betty" ;CRT x(1),y(2)
```

Attempting to reference a dimensioned array in a subsequent command line results in a MVBasic syntax error.

You cannot **DIM** the same array twice in a **DIM** statement. You cannot **DIM** an array that has already been declared using the **COMMON** statement. Attempting to do so results in a compile error.

You can use the **EXISTS** function or the **$DATA** function to determine if a variable or array node has been defined.

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

Using Dimensioned Arrays

You can use the INMAT function to return the defined dimensions of a static array.

Emulation

IN2, INFORMATION, PIOpen, Prime, UniData, and UniVerse respond to an undimensioned array element by issuing a runtime <UNDEFINED> error. Other emulations respond to an undimensioned array element by issuing a compile-time syntax error.

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)

! Dimensions a two-dimensional array using local variables
DIM MyMatrix(myrows,mycols)

! 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

- COMMON statement
- MAT statement
- MATBUILD statement
- MATPARSE statement
- MATREAD statement
- MATWRITE statement
- $DATA function
- EXISTS function

- INMAT function
- Variables

DISPLAY

Displays on the terminal screen.

```
DISPLAY [text]
DISPLAY  text [format]
```

Arguments

text	*Optional* — Any MVBasic expression that resolves to a quoted string or a numeric. You can specify a single expression or a series of expressions separated by either commas (,) or colons (:). A comma inserts a tab spacing between the two strings. A colon concatenates the two strings. If *text* is omitted, a blank line is returned.
format	*Optional* — A code specifying how to handle *text*, specified as a quoted string. This *format* is applied to the *text* that immediately precedes it. Whitespace characters may be inserted between *text* and *format*.

Description

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

See Also

- CRT statement
- PRINT statement
- FMT function

ECHO

Suppresses user input display on the screen.

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

Arguments

expression	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.

The **ECHO** statement suppresses screen display of user input. The **HUSH** statement suppresses All screen display.

Examples

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

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

See Also

- HUSH statement
- CRT statement
- PRINT statement
- INPUT statement

END

Terminates a block of code or a program.

```
END
```

Arguments

None.

Description

The **END** statement has three uses:

- As a clause terminator
- As a statement terminator
- As a routine terminator

Clause Terminator

When used as a clause terminator, the **END** keyword terminates execution of a block of code.

END is 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.

END is used as part of a multiline LOCKED clause, ON ERROR clause, THEN clause, or ELSE clause, where it terminates execution of the block of code.

Block code clauses have specific line break requirements:

- Each block code statement must appear on its own line.
- The LOCKED, ON ERROR, THEN, or ELSE keyword cannot precede a block code statement on the same line.
- The END keyword can appear on its own line, or can appear at the end of the final block code statement line. The code line END ELSE (concluding a multiline THEN clause and beginning an ELSE clause) is also valid.

The following are valid syntactic forms:

```
command args
THEN
statement1
statement2
END
ELSE
statement1
statement2
END

command args THEN
statement1
statement2 END
ELSE
statement1
statement2 END

command args THEN
statement1
statement2
END ELSE
statement1
statement2
END
```

Statement Terminator

The **END** keyword is used with another keyword in a few statements to indicate the end of the code encompassed by that statement. These uses are:

- BEGIN CASE ... CASE ... END CASE
- BEGIN TRANSACTION ... END TRANSACTION
- TRY ... CATCH ... END TRY

Routine Terminator

When used outside of a block structure clause **END** terminates routine or program execution. Commands following an **END** statement are not executed. If additional lines of code appear after the **END** statement, Caché (and all emulation modes), by default, generates an error: “Unexpected line outside of program”. You can set $OPTIONS IGNORE.EXTRA.LINES to ignore lines that appear after the **END** statement, rather than issuing an error message.

See Also

- GOTO statement
- $OPTIONS statement
- IF...THEN statement
- RETURN statement

END TRANSACTION

Specifies where to continue execution after a transaction.

```
END [TRANSACTION | WORK]
```

Description

The **END TRANSACTION** statement specifies the end of a transaction. This is where to continue program execution following 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.

The TRANSACTION or WORK keywords are optional and provides no functionality. They are provided solely for compatibility with other MultiValue vendor products.

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.

Example

The following example performs database operations within a transaction. It sets a variable x, which determines whether the transaction should be committed or rolled back.

```
PRINT "Before the transaction"
BEGIN TRANSACTION
 .
 .
 .
IF x=0
  THEN COMMIT
  END
  ELSE ROLLBACK
  PRINT "Transaction rolled back"
  END
PRINT "This should not print"
END TRANSACTION
PRINT "After the transaction"
```

See Also

- BEGIN TRANSACTION statement
- COMMIT statement
- ROLLBACK statement

ENTER

Transfers control to an external subroutine.

```
ENTER name
```

Arguments

name	Name of the external subroutine to call.

Description

The **ENTER** statement can be used to call an external subroutine. The external subroutine must have been compiled and cataloged. No parameters can be passed using **ENTER**; use **CALL** if you need to pass parameters to a subroutine. When **ENTER** is used to call an external subroutine, the **RETURN** statement within the external subroutine does not return control to calling program; use **CALL** if you need to return following a subroutine call.

ENTER calls the external subroutine without increasing the stack level. This can be useful when issuing a large number of calls without returning. Because **ENTER** is not increasing the stack level, a <FRAMESTACK> error cannot occur.

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.

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

ENTER, CALL, GOSUB, and SUBR

The **ENTER** statement is used to call an external subroutine with no parameter passing or return, and without increasing the stack level. The **CALL** statement is used to call an external subroutine with parameter passing and returning. **CALL** increases the stack level.

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 **ENTER** to call an external subroutine:

```
Main
  x="Burma"
  PRINT x           ! Returns "Burma"
  ENTER ErrorSub
  PRINT x           ! Does not execute

ErrorSub
  PRINT "An error occurred"
  QUIT
```

See Also

- CALL statement
- COMMON statement
- RETURN statement
- SUBROUTINE statement

- END statement
- DIM statement
- GOSUB statement
- SUBR function

EQUATE

Replaces a symbol with a value at compile time.

```
EQUATE symbol TO expression [, ...]
EQU symbol TO expression [, ...]

EQUATE symbol LITERALLY str [, ...]
EQU symbol LIT str [, ...]
```

Arguments

symbol	The placeholder symbol to be replaced, specified as one or more characters. The first character must be a letter or percent sign (%). Subsequent characters may be letters, numbers, percent sign (%), underscore (_), or dollar sign ($). The final character may not be an underscore.
expression	The value used to replace all instances of *symbol* at compile time. Any valid Caché MVBasic expression.
str	The string used to replace all instances of *symbol* at compile time. Specified as a quoted string.

Description

EQUATE replaces every instance of *symbol* in the program with the specified expression or variable. **EQUATE** performs this substitution at compile time. Therefore, the value replaced is not affected by program execution. **EQUATE** can be used to replace executable statements in the program. Variables perform substitutions during program execution and cannot be used to modify the program's executable code.

You can specify multiple `symbol TO expression` and `symbol LITERALLY str` clauses in any combination as a comma-separated list. You can insert line breaks as needed following a comma separator.

EQUATE treats a sequence of words separated by `->` as a single entity. For example:

```
EQUATE vin TO car(7)
AutoCheck->vin = vin
```

returns `AutoCheck->vin = car(7)`.

The **EQUATE** keyword can be abbreviated as **EQU**. The **LITERALLY** keyword can be abbreviated as **LIT**.

Examples

The following example replaces at compile time every instance of the symbol `addlength` with the expression BYTELEN(x)+20 or LEN(x)+10, depending on the setting of the Unicode variable:

```
IF Unicode=1
  THEN EQUATE addlength TO BYTELEN(x)+20
  ELSE EQUATE addlength TO LEN(x)+10
```

The following example replaces at compile time every instance of the symbol `letters` with the contents of the variable *alpha*:

```
EQUATE letters LITERALLY "alpha"
BEGIN CASE
  CASE lang=English
     alpha="abcdefghijklmnopqrstuvwxyz"
  CASE lang=Greek
     alpha="           μ           "
END CASE
```

See Also

- Variables

ERRMSG

Displays the specified error message.

```
ERRMSG errcode [,val1[,val2]]
```

Arguments

errcode	An expression that resolves to a MultiValue error code; commonly (but not always) specified as a positive integer. The error code can be specified as a literal or as a expression that resolves to a literal value. A non-numeric literal value must be specified as a quoted string.
val	*Optional* — A comma-separated list of one or argument values inserted into the error message text. These argument values can be specified as literals or as expressions that resolve to literals. A non-numeric literal value must be specified as a quoted string.

Description

ERRMSG displays the error message text corresponding to the *errcode* error code. Error messages are defined in the ERRMSG file. An error message text commonly includes the error code (in square brackets) as part of the message.

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

If you specify one or more *val* arguments, **ERRMSG** displays the *errcode* error message text with these *val* arguments inserted in the message. If the *errcode* error message does not take an inserted value, the *val* argument is ignored. If the *errcode* 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^
```

For a list of error codes and corresponding error messages, see Error Messages in the *Caché MultiValue Commands Reference*.

See Also

- ABORTE statement
- STOPE statement

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 | NULL} |  OUTPUT oref]
                    [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

command	One or more MultiValue commands, each command specified as a quoted string. A string can be quoted using single quotes ('cmd arg'), double quotes ("cmd arg"), or backslashes (\cmd arg\). To specify multiple commands, separate the commands with a Field Mark ("cmd1 arg":@FM:"cmd2 arg").
var	A variable used to hold a value.
dynarray	A dynamic array.
oref	An object reference. The corresponding class must have (at minimum) a **WriteLine()** method (which inserts a newline at the end of the write operation) and a **Write()** method (which does not insert a newline).

Description

The **EXECUTE** command executes the specified Caché MultiValue command(s), then returns execution to the next MVBasic statement following the **EXECUTE**. It initially searches the VOC for the *command*; if the *command* is not found in the VOC, it searches the global catalog. For lookup details, refer to CATALOG in the *Caché MultiValue Commands Reference*.

The first syntactical form supports the following optional clauses:

- The CAPTURING clause diverts all terminal output from the MultiValue command to the supplied *dynarray* variable. This output is stored as a dynamic array, with lines separated by Field Marks. If *command* executes successfully, the resulting terminal output is captured; if *command* fails, the error message is captured. CAPTURING NULL discards all terminal output, with the following exceptions: Output from the OUT statement is displayed. Output from non-MultiValue commands or shell commands cannot be captured, and is therefore displayed. If *command* includes the **HUSH ON** command, output is not stored in *dynarray* based on that command, and terminal display is disabled upon return from the **EXECUTE** command.

- The OUTPUT clause diverts all terminal output from the MultiValue command to the supplied oref. (One use of this object is to invoke a class from which you can execute write methods to write to a sequential file.) This is especially useful when handling extremely large command outputs (>3.6 Mbytes). The following example:

  ```
  oref = "%Stream.FileCharacter"->%New()
  EXECUTE "LIST BIGFILE ID-SUPP A1" OUTPUT oref
  ```

  directs the output to the standard Caché %Stream class using the standard **Write()** and **WriteLine()** methods. An object derived from a user-written class can be used if it has both a **Write()** and **WriteLine()** method, as shown in the example below. A **WriteLine()** method ends by forcing a new line; a **Write()** method does not force a new line.

  If an error occurs, such as specifying a *filevar* that is not an existing sequential file, **EXECUTE** fails without displaying an error message. It is the programmer's responsibility to check the @SYSTEM.RETURN.CODE for -1, indicating an error. If using a Unicode version of Caché, you must change the file translation in the locale. The OUTPUT clause and the CAPTURING clause are mutually exclusive.

- 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 *var* variable must be simple variable name. It cannot include a system variable, an **EQUATE**, a dynamic array reference, or a substring reference.
- The PASSLIST clause supplies the specified *dynarray* to the executed command as the current default external select list.
- The STATUS clause *var* variable contains the execution status of the last executed list or select command. If the command completed successfully, *var* contains the number of items listed, selected, or otherwise processed. If the command failed, *var* contains -1. If the specified command name is not a valid command, *var* contains -1. Commands that do not list or select items do not set *var*; *var* is set to 0 regardless of whether the command succeeded or failed. *var* can be a simple variable, or a single dynamic array reference (A<i>), a single substring reference (A[s,l]), or a substring reference nested inside a dynamic array reference (A<i>[s,l]).

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.

Emulation

In Reality emulation, **EXECUTE** executes all stacked data, regardless of list status. In Caché and all other emulations, only the **EXECUTE** *command* argument is executed. For all emulations (except Reality) **EXECUTE** either clears or maintains stacked data on the DATA queue, depending on the STACK.GLOBAL option.

EXECUTE supports the STACK.GLOBAL option, which can be set using **$OPTIONS**. When STACK.GLOBAL is on, **EXECUTE** does not clear unused items from the DATA queue upon completion. By default, STACK.GLOBAL is on for Caché, and for UniVerse, UniData, PICK, Prime, PIOpen, and IN2 emulations. STACK.GLOBAL is off for all other emulations.

In jBASE emulation, the RTNLIST clause returns the select list expanded to a dynamic array. All other emulations return the select list name. The PASSLIST clause requires that the select list be designated as external. This external select list bit setting is only used for jBASE emulation.

EXECUTE supports the RETURNING.CODE option, which can be set using **$OPTIONS**.

In UniData and UDPICK emulations, a *command* name with an initial character of * is handled as a global name. **EXECUTE** removes the leading * and then looks up the resulting command name in the global catalog in SYS.MV, rather than looking up in the VOC. If the runtime environment is not a UniData emulation, a normal VOC lookup is done on the **command* name.

Invoking Other Command Shells

You can use the **$XECUTE** command to issue an ObjectScript command from within Caché MVBasic.

You can use the **PCPERFORM** command to issue an operating system command from within Caché MVBasic.

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()
```

This following example directs output to a sequential file using the OUTPUT clause. It uses the standard Caché-supplied %Stream.FileCharacter class, which uses an operating system file as temporary storage. The location of the file is determined by the class itself, although a method allows you to override the default location. This example uses the default location. Once the object is closed, the temporary file is deleted. In this example, the size of the output from the **LIST** command is limited only by the maximum size of the file:

```
* Execute the LIST command to an object.
*
oref = "%Stream.FileCharacter"->%New()
EXECUTE "LIST BIGFILE ID-SUPP A1 SAMPLE 100" OUTPUT oref
*
* Read back that object, one line at a time.
*
oref->Rewind()
lineno = 1
LOOP WHILE oref->AtEnd = 0 DO
line = oref->ReadLine()
PRINT lineno "R%4":": " : OCONV(line,"MCP")
lineno = lineno + 1
REPEAT
```

The following example uses an OUTPUT clause with a user-defined MVExecute.Output class:

```
0001: oref = "MVExecute.Output"->%New()
0002: EXECUTE "LIST BIGFILE ID-SUPP A1 SAMPLE 100" OUTPUT oref
0003: *
0004: * Read back that object, one line at a time.
0005: *
0006: LineCount = oref->LineCount
0007: PRINT "Line count = ":LineCount
0008: FOR I = 1 TO LineCount
0009: PRINT I "R%4" : ": " : oref->ReadLine(I)
0010: NEXT I
```

The following is the definition of this user-defined MVExecute.Output class. It contains the required **Write()** and **WriteLine()** methods, and a **ReadLine()** method:

```
Class MVExecute.Output Extends %Persistent
{

Property LineCount As %Integer;

Property Lines As array Of %String;

Method Write(line As %String) As %Integer [ Language = mvbasic ]
{
    IF NOT(@ME->LineCount) THEN
        @ME->LineCount = 1
        dummy = @ME->Lines->SetAt("",1)
    END
    LineNew = @ME->Lines->GetAt(@ME->LineCount) : line
    dummy = @ME->Lines->SetAt(LineNew , @ME->LineCount)
    RETURN 0
}

Method WriteLine(line As %String) As %Integer [ Language = mvbasic ]
{
    IF NOT(@ME->LineCount) THEN
        @ME->LineCount = 1
        dummy = @ME->Lines->SetAt("",1)
    END
    LineNew = @ME->Lines->GetAt(@ME->LineCount) : line
    dummy = @ME->Lines->SetAt(LineNew , @ME->LineCount)
    @ME->LineCount = @ME->LineCount + 1
    RETURN 0
}

Method ReadLine(LineNumber As %Integer) As %String [ Language = mvbasic ]
{
    IF LineNumber LE 0 OR LineNumber GT @ME->LineCount THEN RETURN "??"
    RETURN @ME->Lines->GetAt(LineNumber)
}

}
```

See Also

- CHAIN statement
- OUT statement
- PERFORM statement
- STATUS function
- 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.

Emulation

In jBASE emulation mode, **EXIT** has both an argumentless and an argumented form.

- **EXIT** without an argument is used to exit a loop, as described above. The keyword **BREAK** without an argument can also be used for this purpose.
- **EXIT** with an argument is used to exit a program and return the argument value. The argument is commonly an integer code value.

See Also

- FOR...NEXT statement
- LOOP...REPEAT statement
- GOTO statement
- CONTINUE statement

FILELOCK

Locks a MultiValue file.

```
FILELOCK [filevar] [,locktype] [ON ERROR statements] [LOCKED statements]
```

Arguments

filevar	*Optional* — A file variable name used to refer to a MultiValue file. This *filevar* is supplied by the **OPEN** statement. If not specified, the default file is locked.
locktype	*Optional* — The type of lock requested, specified by the keyword SHARED or EXCLUSIVE. If not specified, the default is EXCLUSIVE.

Description

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

You can optionally specify a LOCKED clause. This clause is executed if *filevar* refers to a file that has already been locked by another user. The clause is executed if *locktype* conflicts with an existing lock. The LOCKED clause is optional, but strongly recommended; if no LOCKED clause is specified, program execution waits indefinitely for the conflicting lock to be released. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the LOCKED keyword and the first line.

If a file is locked by another user, the STATUS function returns the process ID (pid) of the user holding the lock.

You can optionally specify an ON ERROR clause. If file lock fails, the ON ERROR clause is executed. This may occur if *filevar* does not refer to a currently open file. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

You can release a file lock by issuing a **FILEUNLOCK**, issuing a **RELEASE** with no record ID, or by closing the file.

File and Record Locking

A **FILELOCK** is equivalent to taking a **RECORDLOCK** on all records in the file. For **FILELOCK** to exclusively lock a file, not only must no other user have a conflicting **FILELOCK**, but no other user may have a RECORDLOCKU or RECORDLOCKL for any record of the file. You can check the status of file locks and record locks using the RECORD-LOCKED function.

Lock Promotion

If you have a shared lock on a file, then request an exclusive lock on the same file, MVBasic attempts to get the exclusive lock. If it is successful, your shared lock is promoted to an exclusive lock. The result is that you hold one exclusive lock, not two locks.

See Also

- OPEN statement
- FILEUNLOCK statement
- RELEASE statement

- RECORDLOCKED function
- STATUS function

FILEUNLOCK

Unlocks a MultiValue file.

```
FILEUNLOCK [filevar] [ON ERROR statements]
```

Arguments

filevar	*Optional* — A file variable name used to refer to a MultiValue file. This *filevar* is supplied by the **OPEN** statement. If not specified, the default file is unlocked.

Description

The **FILEUNLOCK** statement is used to unlock a MultiValue file, undoing the lock established by **FILELOCK**. **FILEUNLOCK** only releases locks established by **FILELOCK**. It does not release record locks; record locks can be released using the RELEASE statement. You can check the status of file locks and record locks using the RECORDLOCKED function.

You can unlock a MultiValue file by issuing a **FILEUNLOCK**, by issuing a RELEASE with no record ID, or by closing the file.

FILEUNLOCK takes the file identifier *filevar*, defined by the **OPEN** statement.

You can optionally specify an ON ERROR clause. If file unlock fails, the ON ERROR clause is executed. This may occur if *filevar* does not refer to a currently open file. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

See Also

- CLOSE statement
- FILELOCK statement
- RELEASE statement
- STATUS statement
- RECORDLOCKED function
- STATUS function

FIND

Finds an element of a dynamic array by exact value.

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

Arguments

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

The *f*, *v*, and *s* arguments accept a single dynamic array reference (A<i>), a single substring reference (A[s,l]), or a substring reference nested inside a dynamic array reference (A<i>[s,l]).

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 *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

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

substring	A string to match against each element in *dynarray*.
dynarray	The target dynamic array in which *substring* is located.
occurrence	*Optional* — An integer that specifies which occurrence of substring to return *dynarray*. The default is 1.
fm *vm* *sm*	Variables that receive an integer specifying the Field Mark (*fm*) Value Mark (*vm*) and Subvalue Mark (*sm*) where *substring* is located in *dynarray*. For further information on these level delimiters, see the Dynamic Arrays page of this manual.

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.

The *fm*, *vm*, and *sm* arguments accept a single dynamic array reference (A<i>), a single substring reference (A[s,l]), or a substring reference nested inside a dynamic array reference (A<i>[s,l]).

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 *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

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

FLUSH

Flushes (immediately applies) writes to a sequential I/O file.

```
FLUSH filevar [THEN statements] [ELSE statements]
```

Arguments

filevar	A file variable name used to refer to a MultiValue sequential I/O file. This *filevar* is supplied by the **OPENSEQ** statement.

Description

The **FLUSH** statement flushes the I/O buffer for a MultiValue sequential file. That is, it immediately performs any pending file I/O **WRITESEQ** operations. It takes the file identifier *filevar*, defined by the **OPENSEQ** statement.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If the file buffer flush is successful (the specified file exists), the THEN clause is executed. If the buffer flush fails (the specified file does not exist), the ELSE clause is executed. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

Caché MVBasic also provides a **NOBUF** statement, which disables I/O buffering, causing all subsequent I/O operations to be immediately issued to the sequential file.

See Also

- OPENSEQ statement
- WRITESEQ statement
- NOBUF statement

FOOTING

Prints a footer at the bottom of each output page.

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

Arguments

channel	*Optional* — An integer that specifies a logical print channel. The default is 0.
footer	The footer to print on output pages, specified as a string enclosed in double quotation marks. This footer can consist of any combination of literal text and code characters. Code character letters are enclosed in single quote characters, and are not case-sensitive.

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 **FOOTING** operation can be reversed using PRINTER RESET, which resets the footing (and heading) to null.

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' \	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'. This code specifies the page number position and alignment; the **PAGE** statement defines the actual page number value.
'S'	Include current page number, left-aligned. This code specifies the page number position and alignment; the **PAGE** statement defines the actual page number value.
'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. The **PAGE** statement advances printing to the next page and prints any defined heading or footing on that 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
- PAGE statement
- PRINTER statement
- PRINTER RESET statement

FORMLIST

Selects field ids into a numbered select list.

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

Description

The **FORMLIST** statement is functionally identical to the **SELECT** statement.

See Also

- SELECT 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 [var]
```

Arguments

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

var	A numeric variable used as a loop counter. *var* must be a local variable. It can be a % variable. It can be a subscripted array. *var* cannot be an @ variable, a global variable, or an object property. It cannot be an element of a user-defined type.
start	Initial value of counter.
end	Final value of counter.
STEP *increment*	*Optional* — The STEP clause sets the amount the counter is changed each time through the loop. *increment* can be a positive or negative integer. If a STEP clause is not specified, *increment* defaults to 1. If *increment* is 0, **FOR...NEXT** loops infinitely.
WHILE *expression* UNTIL *expression*	*Optional* — 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.
statements	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.

FOR and GOTO

Caché MVBasic permits you to exit or enter a **FOR** loop using a **GOTO** statement. This implementation of **GOTO** follows MultiValue standards, and is less restrictive than the ObjectScript standard for **GOTO** statements.

FOR.INCR.BEF

Caché MVBasic supports FOR.INCR.BEF as the Caché default. This option increments the **FOR** loop counter before performing bounds checking. To perform bounds checking before incrementing the loop, specify `$OPTIONS -FOR.INCR.BEF` to turn off this option.

See Also

- CONTINUE statement
- EXIT statement
- GOTO statement
- LOOP...REPEAT statement
- IF...THEN statement

FUNCTION

Defines an external function.

```
FUNCTION name[(arglist)]
    [statements]
    RETURN(returnval)
```

Arguments

name	Name of the **FUNCTION**; follows standard variable naming conventions.
arglist	*Optional* — List of variables specifying arguments that are passed to the **FUNCTION** procedure when it is called. Multiple arguments are separated by commas. The *arglist* is enclosed with parentheses.
statements	A group of statements to be executed within the body of the **FUNCTION** procedure.
returnval	Return value of the **FUNCTION**. If no return value is specified, **FUNCTION** returns the empty string.

Description

The **FUNCTION** statement defines an external function that returns a value to the invoking procedure. This **FUNCTION** procedure is visible to all other procedures in your script. The values of local variables in a **FUNCTION** are not preserved between calls to the procedure.

The **FUNCTION** statement is very similar to SUBROUTINE, except that **FUNCTION** returns a value. Like a **SUBROUTINE** 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 **SUBROUTINE** procedure, you can use a **FUNCTION** procedure on the right side of an expression in the same way you use any intrinsic function.

There cannot be a label on the **FUNCTION** statement line. The **FUNCTION** statement must be the first line in the external function, with the following exceptions: comment lines, **$OPTIONS** statements, **$COPYRIGHT** statements, and **DIM** statements that do not dimension a static array. For example, `DIM Var()` and `DIM abc` are permitted, but `DIM Var(2)` is not.

There can only be one **FUNCTION** statement in an external function (no nested functions). You can't define a **FUNCTION** procedure inside another **FUNCTION** or inside a **SUBROUTINE** procedure.

Before invoking a function, it must be locally defined using the DEFFUN statement.

Examples

The following two examples show the definition of a function and the invocation of that function:

```
FUNCTION IsGreaterThan(lower, upper)
IF lower < upper
THEN RETURN(1)
ELSE RETURN(0)

DEFFUN IsGreaterThan(x,y)
CRT IsGreaterThan(x,y)
```

See Also

- DEFFUN statement
- DIM statement

- RETURN statement
- SUBROUTINE statement

GET(ARG.)

Retrieves the next command line argument.

```
GET(ARG.[,n]) variable [THEN statements] [ELSE statements]
```

Arguments

n	*Optional* — An integer specifying which command line argument to retrieve. The default is the first unread argument (the next argument).
variable	A local variable used to hold the value of the command line argument retrieved.

Description

The **GET(ARG.)** statement retrieves a command line argument, copying its value into *variable*. Each time you invoke **GET(ARG.)** it updates a command line pointer. Therefore, repeated invocation of **GET(ARG.)** without the *n* argument results in the sequential retrieval of each command line argument in left-to-right order.

The keyword `ARG.` (note the period at end of this keyword) and the surrounding parentheses are mandatory.

You can use the optional *n* value to retrieve a command line argument by its integer position in the command line argument list. Command line arguments are counted from 1. If *n*=0, **GET(ARG.)** retrieves the next command line argument.

GET(ARG.) considers all values following the program name to be command line arguments. Command line arguments are separated by blank spaces; a blank space within a quoted string is not treated as a command line argument separator.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If the command line argument retrieval is successful, the THEN clause is executed. If there are no command line arguments, no more command line arguments, or if you specify a value of *n* that does not correspond to a command line argument, or a negative value for *n*, **GET(ARG.)** executes the ELSE clause. If no ELSE clause in provided, **GET(ARG.)** returns the empty string to *variable*. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

The **GET(ARG.)** statement both moves the command line argument pointer and retrieves the argument value. The **SEEK(ARG.)** statement just moves the command line argument pointer. The **EOF(ARG.)** function returns whether or not the command line argument pointer is past the end of the list of command line arguments.

See Also

- SEEK(ARG.) statement
- EOF(ARG.) function

GETLIST

Retrieves a saved select list.

```
GETLIST listname [TO listnum] [SETTING variable] [THEN statements] [ELSE statements]
```

Arguments

listname	A record ID assigned to a saved select list.
TO *listnum*	*Optional* — A numbered select list, specified as an integer from 0 through 10. If omitted, select list 0 is used.
SETTING *variable*	*Optional* — An integer count returned, which contains the number of elements in the specified select list.

Description

The **GETLIST** statement retrieves a saved select list, making it available to the **READNEXT** statement. You specify the *listname* specifying the storage location of the select list, and the *listnum* of a numbered active select list into which to copy it. The select list was saved using **WRITELIST**.

The *listname* select list is saved in the &SAVEDLISTS& file. Caché stores this file using the ^SAVEDLISTS global.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If the saved select list retrieval is successful, the THEN clause is executed. If saved select list retrieval fails (*listname* does not exist), the ELSE clause is executed. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

See Also

- READNEXT statement
- WRITELIST statement

GOSUB

Transfers program execution to a label, with return option.

```
GOSUB label
```

Arguments

label	Any valid label. The *label* 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.

Under the following circumstances control does not revert to the line following the **GOSUB** statement: The internal subroutine invokes an ENTER statement, or the internal subroutine terminates with an END statement.

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.

Emulation

jBASE emulation uses the ONGO.RANGE option setting for handling out-of-range *label* values.

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
G label
```

Arguments

label	Any valid label. The *label* 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. Numeric labels do not use a colon suffix. Non-numeric labels end with a colon character; this colon is option when specifying the *label* argument.

G is an abbreviation for 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.

GOTO can be used to exit from a **FOR...NEXT** or **LOOP...REPEAT** loop. You can also 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.

GOTO can be used to enter the middle of a **FOR...NEXT** or **LOOP...REPEAT** loop. This use of **GOTO** is generally not recommended, and is not supported in other Caché languages, such as ObjectScript.

Emulation

jBASE emulation uses the ONGO.RANGE option setting for handling out-of-range *label* values.

Examples

The following examples illustrate the use of the **GOTO** statement with numeric and non-numeric labels:

Numeric label:

```
  IF TIME()=0 THEN
    GOTO 20
  ELSE
    PRINT Time()
  END IF
  END
20
  PRINT "It's midnight, time is reset to 0"
  END
```

Non-numeric label:

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

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

channel	*Optional* — An integer that specifies a logical print channel. The default is 0.
header	The header to print on output pages, specified as a string enclosed in double quotation marks. This header can consist of any combination of literal text and code characters. Code character letters are enclosed in single quote characters, and are not case-sensitive.

Description

The **HEADING** statement prints a header at the top of each page of output text. This output text can be displayed on a terminal (by default) or directed to a printer (by specifying PRINTER ON).

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 **HEADING** operation can be reversed using PRINTER RESET, which resets the heading (and footing) to null.

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 spaces. You can specify a larger or smaller alignment by appending an integer to 'P'. For example, 'P2'. 'PP' prints the page number twice, both right aligned 4 spaces. This code specifies the page number position and alignment; the **PAGE** statement defines the actual page number value.
'S'	Include current page number, left-aligned. This code specifies the page number position and alignment; the **PAGE** statement defines the actual page number value.
'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.

To clear an existing heading, specify **HEADING CHAR(255)**. If you specify more than one **HEADING** statement in a program, MVBasic issues a form feed before executing the second (and all subsequent) **HEADING** statement(s).

The **HEADING** statement places text at the top of each page. The **FOOTING** statement places text at the bottom of each page. The **PAGE** statement advances printing to the next page and prints any defined heading or footing on that page.

Emulation

IN2, jBASE, MVBase, PICK, Reality, R83, POWER95, Ultimate: in these emulation modes, the **HEADING** statement is immediately applied when issued.

D3: The initial **HEADING** statement is immediately applied when issued. Subsequent **HEADING** statements are applied following either the end of a page or the issuing of a **PAGE** statement. This does not apply if **PRINTER ON** is immediately followed by a **PAGE** statement, or if a **PRINT** statement is followed by **HEADING**.

jBASE: 'PP' includes the page number right aligned 4 spaces.

Reality: 'P' includes the page number with no alignment; 'PP' includes the page number right aligned 4 spaces.

In Caché MVBasic, the **HEADING** is applied to only the current output device. For example, if you specify **HEADING** for the terminal page header, then specify **PRINTER ON**, you must specify **HEADING** again for the printer page header. In D3, MVBase, and Reality, if you specify **HEADING** for the terminal page header, then specify **PRINTER ON**, the terminal header is inherited by printer channel 0.

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
- PAGE statement
- PRINTER statement
- PRINTER RESET statement

HUSH

Suppresses all screen display.

```
HUSH [ON | OFF | flag] [SETTING var]
```

Arguments

flag	*Optional* — An expression that evaluates to a boolean value. 0=disable hushing. 1 (or any non-zero number)=enable hushing. The same boolean values can be supplied using the ON or OFF keyword.
SETTING *var*	*Optional* — A variable that **HUSH** sets to the hush state (0 or 1) prior to invoking the command. This clause is useful for restoring the prior **HUSH** state setting.

Description

The **HUSH** statement is used to enable or disable all output display to the screen. It can be executed using the ON or OFF keyword, or by using a boolean *flag* value. **HUSH** with no arguments toggles the current hush state.

When **HUSH** is ON, all screen output is disabled, both user input and program output, including display of the programmer prompt. This distinguishes it from the **ECHO** statement, which only disables the display of user input.

The **HUSH** statement *does not* disable display of output from the **OUT** statement.

See Also

- ECHO statement
- OUT 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

condition	An expression that evaluates to **True** or **False**. For further details on boolean logical operators, refer to the Operators page of this manual.
statements	One or more statements executed if *condition* is **True**.
elsestatements	One or more statements executed if no previous *condition* 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. If *condition*=1 and the THEN clause is omitted, or *condition*=0 and the ELSE clause is omitted, **IF** returns the empty string. 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.

The *condition* expression can be a compound expression, using = (equal to), # (not equal to), and other the comparison operators. You can use literals, variables, and dynamic arrays as *condition* expression elements. Multiple test expressions can be associated by AND and OR logical operators.

See Also

- CASE statement
- Operators

IN

Reads a single character of user input.

```
IN variable [FOR timeout [THEN statements] [ELSE statements]]
```

Arguments

variable	A variable used to hold the user input character.
FOR *timeout*	*Optional* — An expression that resolves to an integer specifying the number of tenths of a second to wait for input before timing out. A *timeout* value of 0 is permitted. The FOR clause requires either a THEN clause or an ELSE clause, or both.

Description

The **IN** statement pauses program execution for user input, then reads a single character of user input into *variable*. The character is stored in *variable* as an ASCII code value. It is therefore necessary to use the **CHAR** function to display the character.

You specify the *timeout* value in tenths of a second; however, Caché only handles timeout in whole seconds. Caché rounds this *timeout* value to an integer number of whole seconds. Any *timeout* value less than 10 is rounded up to one second.

If no FOR clause is specified, the **IN** statement pauses execution indefinitely until receiving user input. The FOR clause, which is used with the THEN and ELSE clauses, provides for timeout of this pause for user input.

If you have specified a FOR clause, you can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If the user input occurs within the FOR timeout, the THEN clause is executed. If the user input does not occur within the FOR timeout, the ELSE clause is executed. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

By default, the input character is not echoed, regardless of the setting of ECHO. However, echoing is emulation-dependent. For example, in D3 emulation the input character is always echoed, regardless of the setting of ECHO. The user value is echoed to the terminal; it is never echoed to the printer.

If you specify **Ctrl-C** as the user input to **IN**, the process enters the ObjectScript (COS) debugger. It displays the instruction "Type G to continue or Q to exit." You can disable this use of **Ctrl-C** by specifying the **BREAK OFF** statement before issuing the **IN** statement.

You can also use the **KEYIN** function to receive a single character of user input, or the **INPUT** statement to receive one or more characters of user input. You can use the `<<...>>` inline prompt to prompt for a user input value to insert in a MVBasic statement or a MultiValue command line command. The `<<...>>` inline prompt is described in the *Caché MultiValue Commands Reference*.

See Also

- INPUT statement
- KEYIN function

INPUT

Receives user input.

```
INPUT [@(col[,row])] variable [,length [_]] [:] [format] [FOR n | WAITING n]
            [THEN statements] [ELSE statements]

INPUT variable,-1
```

Arguments

@(*col*,*row*)	*Optional* — 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 *variable* at the prompt. A *col* value of 0 or 1 displays the prompt at column 1. If *row* is omitted, it defaults to *row*=1, the top of the Terminal window; *row*=23 is the bottom of the Terminal window.
variable	A variable used to receive the user input. This variable does not need to be previously defined. If *length* is not specified, you can follow *variable* with a colon (:) character to suppress the line return. This character is further described below.
length	*Optional* — An integer specifying the maximum length of the input data. By default, the input data is accepted when the number of characters specified in *length* are input. If less than the number of characters specified in *length* are input, the input data is accepted when the user presses the Enter key. If *length* is omitted, or *length*=0, data of any length can be specified. The data is accepted by pressing the Enter key. The *length* integer can be followed by the underscore (_) character, and/or the colon (:) character (in any order). These special-purpose characters are described below. If *length* is -1, *variable* is assigned a boolean value indicating whether or not data was input. This option does not prompt the user for data.
:	Suppresses line return.
_	Requires Enter key to accept input data, regardless of the length of the input data.
format	*Optional* — A format mask string used to validate the input data. *format* can be specified with or without *length*. If *length* is specified, *format* can be preceded by a comma delimiter or just a blank space. For further details on format mask strings, refer to the FMT function.
FOR *n* WAITING *n*	The FOR *n* and WAITING *n* clauses are functionally identical ways to specify a timeout value. *n* is an integer specifying tenths of a second to wait before timing out. Caché rounds *n* to the nearest whole second interval.

Description

The **INPUT** statement has two forms:

- **INPUT** with *length* specified as a positive integer, or with *length* unspecified. This syntax receives input data. It can be used in interactive programs to receive input data from the user, or to receive input data non-interactively from the **DATA** statement.
- **INPUT** with *length* specified as -1. This syntax tests for the presence of input data and returns a boolean value.

Receiving Input Data

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 is echoed character-by-character at the input prompt.

- If *length* omitted, the user must press the Enter key to accept the input data.
- If the input data is less than the number of characters specified in *length* (or *length*=0), the user must press the Enter key to accept the input data.
- If the input data is equal to the number of characters specified in *length* the input data is accepted without pressing the Enter key. However, if the underscore (_) character is specified after the *length* argument, *length* specifies the maximum number of characters that can be input, but accepting the input data requires pressing the Enter key, regardless of the number of input characters.

INPUT can also receive data from the **DATA** statement, as described below. If data is present in a **DATA** statement, the ? prompt and user input are suppressed, and input is taken from **DATA**.

By default, when **INPUT** accepts data input it performs a line return. You can suppress this line return by following either the *variable* or the *length* argument with a colon character (:). You can append a colon to *variable* if *length* is not specified; otherwise, append the colon to *length*. You can include or omit a space between *variable* or *length* and the colon.

If *length*=0, user input continues until the Enter key is pressed.

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*. (If *variable* is undefined, the prompt displays an empty string as the previous value.) To accept the previous value, press the Enter 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 Enter key. This @(*col*,*row*) clause suppresses the line return following data input. For further details, refer to the @ function.

By default, the input characters and the @ clause previous value are echoed, regardless of the setting of ECHO. However, **INPUT** echoing is emulation-dependent. These values are echoed to the terminal; they are never echoed to the printer.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If any data is input, the THEN clause is executed. If no data is input (the Enter key is pressed), the ELSE clause is executed. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

You can also use the **KEYIN** function or the **IN** statement to receive a single character of user input. You can use the <<...>> inline prompt to prompt for a user input value to insert in a MVBasic statement or a MultiValue command line command. The <<...>> inline prompt is described in the *Caché MultiValue Commands Reference*.

Timeout

Specifying a timeout for a user input prompt is optional, but highly recommended. You can specify a FOR *n* clause or a WAITING *n* clause to establish how long **INPUT** should wait for completion of user input data before timing out. These two clauses are functionally identical. Input completion is determined by either the Enter key or *length*.

The *n* value is an integer, specifying timeout in tenths of a second. However, Caché timeout is executed in whole seconds. For *n* values less than 10, Caché times out at 1 second. For *n* values greater than 10, Caché rounds to the closest whole second interval. Therefore an *n* value of 3 specifies three-tenths of a second, but actually times out at one second; an *n* value of 13 specifies thirteen-tenths of a second, but is rounded down to 10, so actually times out at one second; an *n* value of 16 is rounded up to 20, so actually times out at two seconds.

When timeout occurs, Caché MVBasic executes the ELSE clause (if present). If no ELSE clause is specified, Caché MVBasic executes the next statement.

INPUT and INPUTIF

INPUT does not support type-ahead — the user's ability to type input data before the prompt is displayed. The **INPUTIF** statement does support type-ahead. **INPUT** and **INPUTIF** are otherwise identical.

Testing for the Presence of Input Data

If *length*=-1, **INPUT** does not prompt the user for data. It checks the input buffer for the presence of data and places a boolean value in *variable*: 1 if data was present in the input buffer; 0 if no data was present in the input buffer. An empty string ("") is considered data. **INPUT** with *length*=-1 tests for the presence of input data, but does not remove data from the input buffer or advance a buffer pointer.

You can use the **DATA** statement to place data in the input buffer. You can use the **CLEARDATA** statement to remove all data from the input buffer. This is shown in the following example:

```
INPUT var,-1 THEN PRINT "Boolean=",var    ;! prints 0
DATA "abc"
INPUT var,-1 THEN PRINT "Boolean=",var    ;! prints 1
CLEARDATA
INPUT var,-1 THEN PRINT "Boolean=",var    ;! prints 0
```

Because **INPUT** with *length*=-1 does not prompt for data, the underscore (_) and colon (:) special-purpose characters have no effect. The @(*col*,*row*), *format*, and FOR *n* or WAITING *n* clauses also have no effect.

To test for the presence of user-input data, use the **SLEEP** statement to allow time for the user to type (or not type) data to the input buffer before **INPUT** checks the input buffer for the presence of data. This is shown in the following example:

```
SLEEP 5
  ;! suspends execution for 5 seconds, allowing the user to type data
INPUT var,-1 THEN PRINT "Boolean=",var
  ;! prints 1 if user input data during sleep interval
  ;! or prints 0 if the user did not input data during sleep interval
  ;! The user-input data (if any) will appear at the MV command prompt
  ;! after the execution of this statement.
```

Non-text Input Values

Null String

To input a null string, you must first designate a character to represent the null string using the INPUTNULL statement. You then specify that designated character to **INPUT** to specify the null string. This **INPUTNULL** character designation only applies to the **INPUT** statement. In all other contexts this character is a literal.

Space and Tab

The user can input space characters and tab characters. In *variable* space and tab are distinct characters. Both space and tab are *length*=1, and both can be removed using a single backspace. However, when echoing input to the terminal, both space and tab are echoed as a space character.

Ctrl-C

If the user types **Ctrl-C** at the prompt, **INPUT** behavior depends on the BREAK setting.

- If **BREAK** is disabled (OFF), any input data that the user has typed into **INPUT** up to that point is deleted. The user can then type a new input value at the prompt and press Enter.
- If **BREAK** is enabled (ON), the process checks the login mode. If in Programmer mode, the process enters the ObjectScript debugger. If in Application mode, it does not enter the debugger. For further details refer to the ObjectScript BREAK command in the *Caché ObjectScript Reference*.

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** item value, but the entire **DATA** item is consumed.

The *length* argument suffix characters (colon or underscore) 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 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.

You can configure **INPUT** to accept only stacked **DATA** input values. You can configure this behavior using the class method **%SYSTEM.MV.InputDataOnly()**. Setting **InputDataOnly()** to 0 (the default) causes **INPUT** to accept both stacked **DATA** and user-input data values; once all stacked **DATA** values are exhausted, the next **INPUT** statement prompts the user for input data. Setting **InputDataOnly()** to 1 causes **INPUT** to accept only stacked **DATA** values; once all stacked **DATA** values are exhausted, the next **INPUT** statement issues an **ABORT**. You can determine the status of the **InputDataOnly()** flag by displaying the ObjectScript $MVV(218) special variable.

Examples

The following example displays the input prompt and pauses ten seconds for user input:

```
PRINT "Input the person's last name"
INPUT namevar,16 FOR 100
IF namevar=""
   PRINT "No name input"
ELSE
   PRINT "Last name (max 16 chars) ":namevar
```

The following example positions the input prompt using the @(col,row) clause, then takes an input of any length to variable *namevar*. If you press the Enter key or timeout without supplying any user input, *namevar* retains the default value "ANONYMOUS".

```
namevar="ANONYMOUS"
INPUT @(1,23) namevar,16 FOR 100
```

The following example takes input data from the **DATA** statement. At each iteration **INPUT** takes the next **DATA** value. Note that in this program **INPUT** takes a maximum of 5 characters, regardless of the length of each **DATA** value; each iteration advances to the next **DATA** value. This program does not pause for user input. However, if the **FOR** loop iterated one more time, the fifth **INPUT** would prompt the user:

```
DATA "Adams","Bean","Clarkenwell","Davis"
FOR i=1 TO 4
  INPUT namevar,5
  PRINT "Last name (max 16 chars) ":namevar
NEXT
```

Emulation

Several aspects of **INPUT** echoing display are emulation-dependent:

For all emulations, except PIOpen, regardless of **ECHO** setting, with INPUT @, the cursor is initially positioned, the prompt displayed, the original value of the data is displayed, and the cursor is positioned on the first character of the original value for user input.

For PIOpen, the cursor is positioned at the location specified by INPUT@, not on the prior position (as in other emulations), and the original value of the data is not displayed.

With **ECHO OFF** set, the original value of the input variable is not displayed in all emulations except UniVerse and Cache.

With **ECHO OFF** set, when you type user input it is displayed character-by-character in Cache, UniVerse, INFORMATION, PIOpen, PICK, and IN2 emulations; typing is not echoed in all other emulations.

With **ECHO OFF** set, when input has been satisfied (by pressing Enter or by entering the number of characters specified on INPUT @) the new value of the variable is redisplayed for most emulations. On PIOpen and UniData no redisplay occurs. On jBASE, if **ECHO ON** the cursor is positioned and the new value is displayed; if **ECHO OFF** the cursor is positioned, and blank spaces the length of the new value are displayed. With **ECHO OFF** set, D3 and Reality replace the length of the original value with blank spaces and then display the new value. With **ECHO OFF** set, UniVerse replaces the original value with blank spaces when the user types the first character; UniVerse redisplays the new value after you press the Enter key.

See Also

- IN statement
- INPUTIF statement
- INPUTNULL statement
- DATA statement
- PROMPT statement
- CLEARDATA statement
- BREAK statement
- KEYIN function
- STATUS function

INPUTCLEAR

Clears input data from the type-ahead buffer.

```
INPUTCLEAR
```

Arguments

None.

Description

The **INPUTCLEAR** statement immediately deletes (clears) any user input data stored in the type-ahead buffer. It does not wait for the next **READ** statement. This affects the **INPUTIF** statement, which receives user input from the type-ahead buffer. **INPUTCLEAR** has no effect on the **INPUT** statement, which does not use a type-ahead buffer.

The **INPUTCLEAR** and **CLEARINPUT** statements are functionally identical.

See Also

- INPUTIF statement
- CLEARINPUT statement

INPUTCTRL

Filters control characters from input.

```
INPUTCTRL [ON | OFF | flag]
```

Arguments

flag	A boolean value. 0: no filtering of control characters (the default). 1 (or any non-zero number): control characters are filtered out of the input stream. The same boolean values can be supplied using the OFF and ON keywords.

Description

The **INPUTCTRL** statement is used to filter out control characters from the characters accepted by the **INPUT** command. It can be executed using the ON or OFF keyword, or by using a boolean *flag* value. The default is OFF, meaning control characters are accepted by **INPUT**.

When **INPUTCTRL** is on, control character sequences that perform operations are neither executed nor recorded as part of the input string (for example, Ctrl-c). Control characters that code for special characters are recorded as part of the input string (for example, Ctrl-r or Ctrl-w).

The **INPUTCTRL** is only applied to the current EXECUTE level.

You can use the CONTROL.CHARS command line command to set or display the current process-wide default for control character filtering.

See Also

- IN statement
- INPUT statement
- INPUTIF statement

INPUTERR

Writes a message to the user terminal.

```
INPUTERR [message [, ...] ]
```

Arguments

message	*Optional* — A string literal to write to the terminal screen. Can be an expression or variable that resolves to a literal value. If omitted, a blank line is written.

Description

The **INPUTERR** statement performs several operations affecting the user terminal.

- It advances the terminal cursor to the last line of the current page. For further information on page lines refer to the SYSTEM function and the ASSIGN statement
- It writes the optional *message* to the terminal screen at the new cursor location. If you specify multiple comma-separated *message* arguments, they are displayed with tab spacing, similar to the **PRINT** or **CRT** command. You can also concatenate multiple *message* arguments, using the colon (:) concatenation operator.
- It deletes (clears) any user input data stored in the type-ahead buffer. This affects the **INPUTIF** statement, which receives user input from the type-ahead buffer.

The *message* is cleared by the next **INPUT @(col,row)** statement.

Emulation

Caché and UniVerse clear user input data stored in the type-ahead buffer. All other emulations do not perform this action.

See Also

- INPUT statement
- INPUTIF statement

INPUTIF

Receives data from input buffer.

```
INPUTIF [@(col[,row])] variable [,length [_]] [:] [format]
          {THEN statements | ELSE statements}

INPUTIF variable,-1  {THEN statements | ELSE statements}
```

Description

The **INPUTIF** statement is used to receive data from the input buffer. While it can be used for interactive user input, this usage is not recommended.

INPUT and **INPUTIF** are similar, with the following differences:

- **INPUTIF** does not display a prompt when awaiting user input. **INPUT** displays a prompt.
- **INPUTIF** does not support timeout clause syntax. For this reason, it should not be used for interactive user input in most circumstances. **INPUT** supports timeout.
- **INPUTIF** requires either a THEN clause, an ELSE clause, or both. For **INPUT** the THEN clause and ELSE clause are optional.
- **INPUTIF** takes the THEN clause when the Enter key is pressed without typing user input data. **INPUT** takes the ELSE clause if *length* is not -1 and the Enter key is pressed without typing user input data.

For further details on **INPUTIF**, refer to the INPUT statement.

See Also

- INPUT statement

INPUTNULL

Specifies a null character for INPUT.

```
INPUTNULL char
```

Arguments

char	An expression that resolves to a single character.

Description

The **INPUTNULL** statement designates a character to represent the null string. If you specify this character to an **INPUT** statement, it is saved to the variable as a null string (a string of length 0). This character remains as the designated null string character for the current process until you reset it by specifying `INPUTNULL ""`.

INPUTNULL only affects the **INPUT** statement. It has no effect on the **IN** statement or the **KEYIN** function.

Example

The following example designates the ^ character to represent the null string for **INPUT**:

```
INPUTNULL "^"
INPUT @(1,23) inval
?^
PRINT "Value =":inval
Value =
PRINT LEN(inval)
0
```

See Also

- INPUT statement

INS

Inserts data in a dynamic array.

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

Arguments

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

To insert a value at the beginning of a *dynarray* set *f* to 1 or 0. To insert a value at the end of a *dynarray* set *f* to -1. 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.

Both the **INS** command and the **INSERT** function insert a value into a dynamic array. The **INS** command changes the value of the supplied *dynarray*. The **INSERT** function returns a dynamic array containing the insert; it does not change the value of the supplied *dynarray*.

Examples

The following example uses the **INS** command 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 YorkýProvidenceýLondonýChicagoýBostonýLos Angeles"
```

See Also

- INSERT function
- COUNTS function
- DELETE function
- EXTRACT function
- Dynamic Arrays

$KILL

Deletes variables.

```
$KILL variable[, ...]
```

Arguments

variable	The variable(s) to be deleted by the **$KILL** command. *variable* can be a single variable name or a comma-separated list of variable names.

Description

The **$KILL** statement deletes the specified variable or comma-separated list of variables. The variables can be local variables, process-private variables, or globals. They do not have to be actual defined variables, but they must be valid variable names. You cannot kill a special variable, even if its value is user-specified. Attempting to do so generates a <SYNTAX> error.

The **ASSIGNED** function returns 0 if a variable is unassigned or has been deleted.

Using **$KILL** to delete variables frees up local variable storage space. To determine or set the maximum local variable storage space (in kilobytes), use the ObjectScript $ZSTORAGE special variable. To determine the currently available local variable storage space (in bytes), use the $STORAGE special variable.

Examples

In the following example, **$KILL** deletes local variables *a*, *b*, and *d*. The **PRINT** returns 3 and 5.

```
a=1
b=2
c=3
d=4
e=5
$KILL a,b,d
PRINT a,b,c,d,e
```

In the following example, **$KILL** deletes the process-private global `^||ppglob` and all of its subscripts. No other variables are affected.

```
^||ppglob(1)="fruit"
^||ppglob(1,1)="apples"
^||ppglob(1,2)="oranges"
$KILL ^||ppglob
PRINT ^||ppglob(1),^||ppglob(1,1)
```

Notes

$KILL and Objects

Object variables (OREFs) automatically maintain a reference count — the number of items currently referring to an object. Whenever you set a variable or object property to refer to an object, Caché increments the object's reference count. When you **$KILL** a variable, Caché decrements the corresponding object reference count. When this reference count goes to 0, the object is automatically destroyed; that is, Caché removes it from memory. The object reference count is also decremented when a variable is set to a new value, or when the variable goes out of scope.

In the case of a persistent object, call the %Save() method before removing the object from memory if you wish to preserve changes to the object. The %Delete() method deletes the stored version of a Caché object; it does not remove the in-memory version of that object.

Using $KILL with Arrays

You can use **$KILL** to delete an entire array or a selected node within an array. The specified array can be a local variable, a process-private global, or a global variable. For further details on global variables with subscripted nodes, see Global Structure in *Using Caché Globals*.

To delete a global array and all of its subordinate nodes, simply supply the global name to **$KILL**.

To delete an array node, supply the appropriate subscript. For example, the following **$KILL** command deletes the node at subscript 1,2. This example uses the **ASSIGNED** function to return a boolean value indicating whether the variable has been deleted:

```
^fruitbasket(1)="fruit"
^fruitbasket(1,1)="apples"
^fruitbasket(1,2)="oranges"
^fruitbasket(1,2,1)="navel"
^fruitbasket(1,2,2)="mandarin"
PRINT ^fruitbasket(1)," contains ",^fruitbasket(1,1),
         " and ",^fruitbasket(1,2)
PRINT ^fruitbasket(1,2)," contains ",^fruitbasket(1,2,1),
         " and ",^fruitbasket(1,2,2)
$KILL ^fruitbasket(1,2)
PRINT "1st level node: ",ASSIGNED(^fruitbasket(1))
PRINT "2nd level node: ",ASSIGNED(^fruitbasket(1,1))
PRINT "Deleted 2nd level node: ",ASSIGNED(^fruitbasket(1,2))
PRINT "3rd level node under deleted 2nd: ",ASSIGNED(^fruitbasket(1,2,1))
```

When you delete an array node, you automatically delete all nodes subordinate to that node and any immediately preceding node that contains only a pointer to the deleted node. If a deleted node is the only node in its array, the array itself is deleted along with the node.

See Also

- Variables
- ASSIGNED function
- UNASSIGNED function

LET

Assigns a value to a variable.

```
LET var=expression
```

Arguments

var	Any valid variable name.
expression	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.

LET permits value assignment to all valid variable names, including variable names that are keywords. For clarity and compatibility, use of keywords as variable names is discouraged.

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]]>] [,start]  [BY format] SETTING variable
[THEN statements] [ELSE statements]

LOCATE(data,dynarray[,f[,v];variable[;format]) [THEN statements] [ELSE statements]
```

Arguments

data	The element value to search for in *dynarray*. 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.
IN *dynarray*	A valid dynamic array.
f	*Optional* — An integer that denotes the Field level of the dynamic array to search for the element *data*. Fields are counted from 1. The surrounding angle brackets are required.
v	*Optional* — An integer that denotes the Value level of the dynamic array to search for the element *data*. Values are counted from 1.
s	*Optional — Supported by Some Emulations Only* — An integer that denotes the Subvalue level of the dynamic array to search for the element *data*. Subvalues are counted from 1.
start	*Optional* — An integer specifying the starting location to begin searching the level specified in *f*, *v*, and *s*. This argument is not supported by all emulations.
BY *format*	*Optional* — specifies the collation sequence. Specify *format* as a quoted string with one of the following values: “AL” (ascending, left justified); “AR” (ascending, right justified); “DL” (descending, left justified); “DR” (descending, right justified).
SETTING *variable*	A local variable that **LOCATE** sets to an integer specifying either where *data* is located or where *data* 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*. Caché MVBasic supports both syntactical forms, as shown above.

In Caché MVBasic you can set the *f*, *v* variables to integers to specify which data item(s) of the dynamic array to search. If you search with just the *dynarray* array name, you are searching for an Field within the dynamic array. If you search with *dynarray*<*f*> then you are searching within Field *f* of *dynarray* for a Value. If you search with *dynarray*<*f*,*v*> you are searching for a Subvalue within the Value *dynarray*<*f*,*v*>. For example, setting *f*=2 searches the second dynamic array field for the *data* value. Caché MVBasic **LOCATE** does not support *s* (Subvalue level); this is only supported by the INFORMATION, PIOpen, and UniData emulations, which use a different search logic, as described below.

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. **LOCATE** behavior when *dynarray* is the null string ("") is described below.

The *f*, *v*, and *s* arguments accept a single dynamic array reference (A<i>), a single substring reference (A[s,l]), or a substring reference nested inside a dynamic array reference (A<i>[s,l]).

The optional BY clause specifies the collation (ascending or descending) and the justification (left or right) used to locate a value. Left justification is commonly used for strings, and right justification is used for numbers. Positive and negative numbers are sorted in numeric sequence, regardless of the justification. However, a mixed numeric value (for example `-24degrees`) sorts in string collation sequence, rather than numeric sequence.

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 *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

Null Value Arguments

LOCATE behavior when *data* and/or *dynarray* has an empty string value is as follows:

- *data*="", *dynarray*=value: Sets *variable* to an integer 1 larger than the last element of *dynarray*. The ELSE clause is taken. If *start* is specified with a value greater than 1, *variable=start*.
- *data*=value, *dynarray*="": Sets *variable*=1. The ELSE clause is taken. This is because searching for a nonempty string treats an empty dynamic array component as containing zero subcomponents, so zero components are searched and the **LOCATE** stops before searching the subcomponent at position 1. If *start*=0 or *start*=1, *variable*=1; if *start* is greater than 1, *variable=start*.
- *data*="", *dynarray*="": Sets *variable*=1. The THEN clause is taken. This is because searching for the empty string in the subcomponent contained in an empty component of a dynamic array considers that component as containing one empty subcomponent at position 1 which matches the searched-for empty string. If *start* is specified with a value other than 1, the ELSE clause is taken. If *start*=0 or *start*=1, *variable*=1; if *start* is greater than 1, *variable=start*.

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.

Emulation

In INFORMATION, PIOpen, UDPICK, and UniData, *f*, *v*, and *s* arguments mean to search at that level, rather than to search within that level. The *f*, *v*, and *s* argument values are start positions, rather than array subscripts. The search begins at the lowest level specified and only that level is searched. For this reason, these emulations require the *f* argument, and only these emulations support the *s* argument. $OPTIONS INFO.LOCATE supports this emulation feature. If *dynarray* is the null string (""), the SETTING *variable* is the integer value of the lowest specified level (*f*, *v*, or *s*). These emulations do not support the *start* argument.

In UniData, if *f* is less than or equal to 0, it is treated as 1. If *v*, or *s* (or both) are less than or equal to 0, they are ignored. If *data*="" and *dynarray*="" the THEN clause is always taken, regardless of the value of *start*.

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 "found 2" found in field 1 at position 2
   ELSE PRINT "not found",a
LOCATE "London" IN cities<2> SETTING a
   THEN PRINT "found",a
   ELSE PRINT "not found",a
   ! returns "not found 1", append to field 2 at level 1
LOCATE "London" IN cities<1,3> SETTING a
   THEN PRINT "found",a;
   ELSE PRINT "not found",a
   ! returns 2 not found, append to field 2 at level 2
```

The following example uses the second syntactical form of **LOCATE**. It is otherwise identical to the previous example:

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

See Also

- FIND statement
- FINDSTR statement
- EXTRACT function
- Dynamic Arrays
- Strings
- Variables

LOCK

Obtains a logical process lock.

```
LOCK name [THEN statements] [ELSE statements]
```

Arguments

name	A number or a string, or an expression that evaluates to a number or a string specifying a lock name. Commonly, an integer from 0 through 64.

Description

The **LOCK** statement sets a named process lock, preventing other processes from obtaining a lock with the same name.

Process locks are not incremental: A process can set the same lock multiple times with **LOCK**. A single UNLOCK releases the lock

Commonly, *name* evaluates to an integer in the range 0 through 64. However, in Caché any number or string may be specified as a logical lock name. The lock name may not be empty, so **LOCK ""** sets **LOCK 0**.

You can specify optional THEN and ELSE clauses. If you obtain lock *name*, the THEN clause is executed. If you already have lock *name*, the THEN clause is also executed. If you could not obtain lock *name* because it is held by another resource, the ELSE clause is executed. If you could not obtain lock name because it is held by another resource, 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.

You can determine which locks are held using the LIST.LOCKS command line command. You can unlock one or all locks using the CLEAR.LOCKS command line command. These commands are described in the *Caché MultiValue Commands Reference*.

Example

The following example uses the **LOCK** statement to obtain a logical lock named 17.

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

See Also

- UNLOCK statement
- LIST.LOCKS command line command
- CLEAR.LOCKS command line command

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

condition	*Optional* — Numeric or string expression that evaluates to True or False. Loop repeats either WHILE *condition* is True, or UNTIL *condition* is True. If this clause is omitted, an infinite loop occurs.
statements	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; if specified it must be on the same line as the *condition* clause.

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

LOOP...REPEAT statements can be nested.

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

MAT

Assigns values to all the elements in a dimensioned array.

```
MAT array = value
MAT array1 = MAT array2
```

Arguments

array	Name of an existing array. This array must have been dimensioned using the **DIM** statement.
value	The value to assign to all of the elements of the array. An expression that resolves to a value.

Description

The **MAT** statement assigns values to all of the elements of a specified array. This array may be one-dimensional or two-dimensional. **MAT** has two forms:

- `MAT array=value` assigns the same value to every element of the array.
- `MAT array1=MAT array2` assigns the values of the elements in *array2* to the corresponding elements in *array1*. Both *array1* and *array2* must already have been dimensioned using the **DIM** statement. The arrays may be differently dimensioned. If there are more elements in *array2* than *array1*, the excess *array2* elements are ignored. If there are more elements in *array1* than *array2*, the excess *array1* elements are not assigned a value. All uninitialized variables are treated as zero-length strings ("").

Note: This statement cannot be executed from the MVBasic command shell. Attempting to do so results in a MVBasic syntax error.

Emulation

D3 supports `array1 = MAT array2` as functionally equivalent to `MAT array1 = MAT array2`.

Examples

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

```
! Dimension a one-dimensional array with 6 elements.
DIM MyVector1(6)
! Dimension a one-dimensional array with 10 elements.
DIM MyVector2(10)
! Assign the value "pending" to all elements of MyVector2
MAT MyVector2="pending"
! Assign the values of elements of one array to another array
MAT MyVector1=MAT MyVector2
! Results are a that MyVector1 contains 6 elements all assigned
! the value "pending"
```

See Also

- DIM statement
- MATBUILD statement
- MATPARSE statement
- MATREAD statement
- MATWRITE statement

- Variables

MATBUILD

Builds a dynamic array from a dimensioned array.

```
MATBUILD dynarray FROM array [,start [,end]] [USING delimiter]
```

Arguments

dynarray	A dynamic array, each element of which receives the value of the corresponding dimensioned array element.
array	Name of an existing dimensioned array. This array must have been dimensioned using the **DIM** statement.
start	*Optional* — An integer that specifies the first element to be transcribed. The default is 1.
end	*Optional* — An integer that specifies the last element to be transcribed. You must specify a *start* value to specify an *end* value. The default is the last element in *array*.
USING *delimiter*	*Optional* — The dynamic array delimiter character to be used to separate elements, specified as a variable (for example @VM) or a quoted string. The default is a field mark (@FM). If a string of more than one characters is specified, only the initial character is used. The empty string ("") is a valid value here; it's use would create a string of concatenated elements, not a dynamic array.

Description

The **MATBUILD** statement assigns the values of the elements of a specified dimensioned array to a dynamic array. You can create a dynamic array containing all of the element values of the dimensioned array, or you can limit the dynamic array to those elements of the dimensioned array between *start* and *end*.

Note: This statement cannot be executed from the MVBasic command shell. Attempting to do so results in a MVBasic syntax error.

By default, **MATBUILD** assigns empty strings to unassigned nodes. If the highest subscripts of the dimensioned array are unassigned or have empty string values, the dynamic array is truncated at the last assigned data value. This behavior can be configured using the **%SYSTEM.Process.MVUndefined()** method.

MATBUILD is the functional opposite of **MATPARSE**.

Emulation

D3, jBASE, MVBase, R83, POWER95, Reality, and Ultimate set **$OPTIONS MATBUILD.UNASSIGNED.ERROR**. This causes these emulations to not support unassigned dimensioned array nodes. If **MATBUILD** encounters an unassigned node, it issues an <UNDEFINED> error. This behavior can be configured using the **%SYSTEM.Process.MVUndefined()** method.

UniData **MATBUILD** does not truncate the highest subscripts of a dimensioned array if they are unassigned or have empty string values.

Examples

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

```
! Dimension a one-dimensional array with 6 elements.
DIM MyVector1(6)
! Assign the value "pending" to all elements of MyVector1
MAT MyVector1="pending"
! Assign the values of elements of a dimensioned array
! to a dynamic array
MATBUILD mydynarray FROM MyVector1 USING "^"
```

The results are the *mydynarray* dynamic array string assigned the value "`pending^pending^pending^pending^pending^pending`". Here the "^" character is used as the dynamic array delimiter, rather than the default field mark character.

See Also

- DIM statement
- MAT statement
- MATPARSE statement
- MATREAD statement
- MATWRITE statement
- Variables

MATPARSE

Builds a dimensioned array from a dynamic array.

```
MATPARSE array [,start [,end]] FROM dynarray [USING delimiter]
```

Arguments

array	Name of an existing dimensioned array. This array must have been dimensioned using the **DIM** statement.
start	*Optional* — An integer that specifies the first dimensioned *array* element to receive a value. The default is 1.
end	*Optional* — An integer that specifies the last dimensioned *array* element to receive a value. You must specify a *start* value to specify an *end* value. The default is the last element in *array*.
dynarray	An existing dynamic array, each element of which is transcribed to the corresponding dimensioned array element.
USING *delimiter*	*Optional* — Specifies the dynamic array delimiter character used to define separate elements. The default is a field mark (@FM).

Description

The **MATPARSE** statement assigns the values of the elements of a dynamic array to a dimensioned array. You can create a dimensioned array containing all of the element values of the dynamic array, or you can limit the transcription of dynamic array elements to those only dimensioned array elements between *start* and *end*.

Note: This statement cannot be executed from the MVBasic command shell. Attempting to do so results in a MVBasic syntax error.

MATPARSE is the functional opposite of **MATBUILD**.

Examples

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

```
   ! Dimension a static array
DIM MyArray(2,5)
   ! Create a dynamic array with 5 elements
MyDyn="Fred":@FM:"Barney":@FM:"Wilma":@FM:"Betty":@FM:"Pebbles"
   ! Assign dynamic array elements to the dimensioned static array
MATPARSE MyArray FROM MyDyn
   ! Display the number of elements parsed
CRT INMAT()
   ! Display static array element values
CRT MyArray(1,2); ! returns "Barney"
CRT MyArray(1,3); ! returns "Wilma"
```

The following example uses a *start* value of 2. It is otherwise identical to the previous example:

```
   ! Dimension a static array
DIM MyArray(3,5)
   ! Create a dynamic array with 5 elements
MyDyn="Fred":@FM:"Barney":@FM:"Wilma":@FM:"Betty":@FM:"Pebbles"
   ! Assign dynamic array elements to the dimensioned static array
MATPARSE MyArray,2 FROM MyDyn
   ! Display the number of elements parsed
CRT INMAT()
   ! Display static array element values
CRT MyArray(1,2); ! returns "Fred"
CRT MyArray(1,3); ! returns "Barney"
```

The following example uses Value Marks (@VM) as the dynamic array delimiters. It is otherwise identical to the first example:

```
   ! Dimension a static array
DIM MyArray(2,5)
   ! Create a dynamic array with 5 elements
MyDyn="Fred":@VM:"Barney":@VM:"Wilma":@VM:"Betty":@VM:"Pebbles"
   ! Assign dynamic array elements to the dimensioned static array
MATPARSE MyArray FROM MyDyn USING @VM
   ! Display the number of elements parsed
CRT INMAT()
   ! Display static array element values
CRT MyArray(1,2); ! returns "Barney"
CRT MyArray(1,3); ! returns "Wilma"
```

See Also

- DIM statement
- MAT statement
- MATBUILD statement
- MATREAD statement
- MATWRITE statement
- INMAT function
- Dynamic Arrays

MATREAD, MATREADL, MATREADU

Reads data from a MultiValue file to a dimensioned array.

```
MATREAD array FROM filevar,recID
   [SETTING var] [ON ERROR statements] [[THEN statements] [ELSE statements]]

MATREADL array FROM filevar,recID
   [ON ERROR statements] [LOCKED statements] [[THEN statements] [ELSE statements]]

MATREADU array FROM filevar,recID
   [SETTING var] [ON ERROR statements] [LOCKED statements] [[THEN statements] [ELSE
 statements]]
```

Arguments

array	Name of an existing dimensioned array that receives the file data. This array must have been dimensioned using the **DIM** statement.
filevar	A local variable used as the file identifier of an open MultiValue file. This variable is set by the **OPEN** statement.
recID	The record ID of the record to be read, specified as either a number or an alphanumeric string of up to 31 characters. Letters in a *recID* are case-sensitive. For naming conventions, refer to MATWRITE.
SETTING *var*	*Optional* — When an error occurs, sets the local variable *var* to the operating system's error return code. Successful completion returns 0; error return codes are platform-specific. The SETTING clause is executed before the ON ERROR, THEN, or ELSE clause. Provided for jBASE compatibility.

Description

The **MATREAD**, **MATREADL**, and **MATREADU** statements read the specified record into a dimensioned array.

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

You must use the DIM statement to dimension *array* before issuing any of these statements. If a record read by **MATREAD** has more attributes than specified by **DIM**, the handling of these extra attributes is controlled by the STATIC.DIM option:

- With **$OPTIONS STATIC.DIM** (the default in Reality, PICK, Ultimate, POWER95, MVBase, IN2, and R83 emulations) dimensioned arrays are created starting from element #1. When there is a **MATREAD** of more attributes than the array has dimensions, the extra attributes are appended to the last element, and INMAT returns 0 to indicate the overflow. On legacy platforms, there is no array element 0, and the array usually cannot be re-dimensioned, but Caché does have an element 0 and allows re-dimensioning.
- With **$OPTIONS -STATIC.DIM** (the default in Cache, UniVerse, UniData, INFORMATION, PIOpen, UDPICK, D3, and jBASE emulations) dimensioned arrays are created starting from element 0. When there is a **MATREAD** with more attributes than dimensions, the extra attributes are put into element 0.

A read operation must be able to acquire at least a shared lock on the desired resource. The **MATREADL** statement acquires a shared lock before performing the read. The **MATREADU** statement acquires an update (exclusive) lock before performing the read. An optional LOCKED clause is provided that is executed if the desired lock could not be acquired. A **MATREAD** pauses execution until it can acquire a shared lock on the specified record.

You can optionally specify a LOCKED clause. This clause is executed if **MATREADL** or **MATREADU** could not perform a read due to lock contention. The LOCKED clause is optional, but strongly recommended; if no LOCKED clause is

specified, program execution waits indefinitely for the conflicting lock to be released. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the LOCKED keyword and the first line.

You can optionally specify an ON ERROR clause, which is executed if *array* is not a MultiValue dimensioned array. If no ON ERROR clause is present, the ELSE clause is taken, or an <ARRAY DIMENSION> error is issued. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. **MATREAD** executes the THEN clause if the read was successful. The THEN clause is executed even when all remaining field identifiers are the null string. **MATREAD** executes the ELSE clause if the read operation fails. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

MATREAD, **MATREADL**, and **MATREADU** all read the specified MultiValue file record value into *array*. If *recID* refers to a non-existent record, the read operation fails.

Note: This statement cannot be executed from the MVBasic command shell. Attempting to do so results in a MVBasic syntax error.

The various **MATREAD** statements read from a MultiValue file into a dimensioned array. The various **READ** statements read from a MultiValue file into a dynamic array.

Examples

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

```
DIM myarray(6)
OPEN "TEST.FILE" TO myfile
MATREAD myarray FROM myfile,1
PRINT "the number of records read:",INMAT()
PRINT "the record value:",myarray(1)
```

See Also

- DIM statement
- MAT statement
- MATBUILD statement
- MATPARSE statement
- MATWRITE statement
- OPEN statement
- READ statement
- INMAT function
- Variables

MATWRITE, MATWRITEU

Writes data from a dimensioned array to a MultiValue file record.

```
MATWRITE array {ON | TO} filevar,recID
   [SETTING var] [ON ERROR statements] [LOCKED statements] [THEN statements] [ELSE
statements]

MATWRITEU array {ON | TO} filevar,recID
   [SETTING var] [ON ERROR statements] [LOCKED statements] [THEN statements] [ELSE
statements]
```

Arguments

array	Name of an existing dimensioned array that supplies the record data written to the MultiValue file. This array must have been dimensioned using the **DIM** statement.
filevar	A local variable used as the file identifier of an open MultiValue file. This variable is set by the **OPEN** statement. You can specify either ON or TO as the keyword.
recID	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 *recID* are case-sensitive. Additional naming conventions are described below.
SETTING *var*	*Optional* — When an error occurs, sets the local variable *var* to the operating system's error return code. Successful completion returns 0; error return codes are platform-specific. The SETTING clause is executed before the ON ERROR clause. Provided for jBASE compatibility.

Description

The **MATWRITE** statements are used to write data from a dimensioned array to a record in a MultiValue file.

- MATWRITE writes a record, then releases the update (exclusive) record lock
- MATWRITEU writes a record, retaining the update (exclusive) record lock

You can optionally specify a LOCKED clause. This clause is executed if the write command could not acquire an exclusive record lock due to lock contention. The LOCKED clause is optional, but strongly recommended; if no LOCKED clause is specified, program execution waits indefinitely for the conflicting lock to be released. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the LOCKED keyword and the first line.

You can optionally specify an ON ERROR clause, which is executed if *array* is not a MultiValue dimensioned array. If no ON ERROR clause is present, an <ARRAY DIMENSION> error is issued. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If the record write is successful, the THEN clause is executed. If record write is attempted but fails, the ELSE clause is executed. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

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).

Note: This statement cannot be executed from the MVBasic command shell. Attempting to do so results in a MVBasic syntax error.

Record Naming Conventions

The following are naming conventions for a valid MultiValue *recID*:

- 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.

Empty Nodes

By default, **MATWRITE** assigns empty strings to unassigned nodes. If the highest subscripts of the dimensioned array are unassigned or have empty string values, the resulting record is truncated at the last assigned data value. This behavior can be configured using the **%SYSTEM.Process.MVUndefined()** method.

Record Locks

RECORDLOCKU performs an update (exclusive) lock on a record. This update record lock is automatically released when you write data to the record using **MATWRITE**. The **MATWRITEU** command does not release the update record lock. You can check the status of an update record lock using the RECORDLOCKED function. You can explicitly release an update record lock using the RELEASE command.

MATWRITE and WRITE

The **MATWRITE** and **MATWRITEU** statements write from a dimensioned array to a MultiValue file record. The various **WRITE** statements write from a dynamic array (or an ordinary string) to a MultiValue file record.

Emulation

D3, jBASE, MVBase, R83, POWER95, Reality, and Ultimate set **$OPTIONS MATBUILD.UNASSIGNED.ERROR**. This causes these emulations to not support unassigned dimensioned array nodes. Because **MATWRITE** uses **MATBUILD** to construct the output string, if **MATWRITE** encounters an unassigned node, it issues an <UNDEFINED> error. This behavior can be configured using the **%SYSTEM.Process.MVUndefined()** method.

UniData **MATWRITE** truncates the highest subscripts of a dimensioned array if they are unassigned or have empty string values. (UniData **MATBUILD** does not truncate in these circumstances.)

By default, Caché and the D3, jBASE, PIOpen, Prime, UniData, and UniVerse emulations do not set **$OPTIONS STATIC.DIM**; all other emulations set **$OPTIONS STATIC.DIM**. When set, STATIC.DIM re-dimensions an array at runtime when there are more attributes than the number of dimensioned array elements. Thus excess attributes are appended to the end of the array. When STATIC.DIM is not set, excess attributes are placed in array element 0.

Examples

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

```
DIM myarray(6)
OPEN "TEST.FILE" TO mytest
   IF STATUS()=0
   THEN
     MATWRITE myarray TO mytest,1
     ON ERROR PRINT "MATWRITE error occurred"
     CLOSE mytest
   END
   ELSE
     PRINT "File open failed"
   END
```

See Also

- DIM statement
- MAT statement
- MATBUILD statement
- MATPARSE statement
- MATREAD statement
- OPEN statement
- STATUS statement
- WRITE statement
- Variables

$MERGE

Merge two arrays.

```
$MERGE destination=source
```

Arguments

destination	A local variable, process-private global, or global to be merged. If specified as a class property, the *source* variable must be a multidimensional (subscripted) variable.
source	A local variable, process-private global, or global to be merged. If specified as a class property, the *source* variable must be a multidimensional (subscripted) variable.

Description

The **$MERGE** statement is used to merge two arrays. **$MERGE** *destination=source* copies *source* into *destination* and all descendants of *source* into descendants of *destination*. It does not modify *source*, or kill any nodes in *destination*.

Note: **$MERGE** operates only on non-MultiValue arrays. It cannot be used with a MultiValue static dimensioned array that has been declared using **DIM**.

$MERGE simplifies the copying of a subtree (multiple subscripts) of a variable to another variable. Either variable can be a subscripted local variable, process-private global, or global. A subtree is all variables that are descendants of a specified variable.

If *destination* is undefined, **$MERGE** defines it and sets it to *source*. If *source* is undefined, **$MERGE** completes successfully, but does not change *destination*. When the *destination* and *source* are the same variable, no merge occurs. **$MERGE** issues an error if the *source* and *destination* have a parent-child relationship.

You can specify multiple, comma-separated *destination=source* pairs. They are evaluated in left-to-right order.

The **$MERGE** command can take longer than most other Caché MVBasic commands to execute. As a result, it is more prone to interruption. The effect of interruption is implementation-specific. Under Caché, an interruption may cause an unpredictable subset of the source to have been copied to the destination subtree.

See Also

- Global Structure chapter in *Using Caché Globals*

NAP

Suspends processing for a specified number of milliseconds.

```
NAP [millisecs]
```

Arguments

millisecs	*Optional* — 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

NOBUF

Turns off buffering for sequential file I/O.

```
NOBUF filevar [THEN statements] [ELSE statements]
```

Arguments

filevar	A file variable name used to refer to a MultiValue sequential I/O file. This *filevar* is supplied by the **OPENSEQ** statement.

Description

By default, sequential file I/O is performed using I/O buffering. This buffer is automatically assigned as part of the **OPENSEQ** operation. I/O buffering significantly improves overall performance, but means that write operations are not immediately applied to the sequential file. The **NOBUF** statement disables the I/O buffer for an open MultiValue sequential file. That is, all I/O operations are immediately executed on the sequential file.

NOBUF takes the file identifier *filevar*, defined by the **OPENSEQ** statement. Thus, **NOBUF** can only be issued after a sequential file has been opened with I/O buffering.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If the file buffer is successfully disabled (the specified *filevar* exists), the THEN clause is executed. If the buffer disable fails (usually because the specified *filevar* does not exist), the ELSE clause is executed. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

Caché MVBasic also provides a **FLUSH** statement, which immediately writes the current contents of the I/O buffer to the sequential file.

See Also

- OPENSEQ statement
- WRITESEQ statement
- FLUSH statement

NULL

Performs no operation, used in a clause.

```
NULL
```

Arguments

The **NULL** statement takes no arguments.

Description

The **NULL** statement performs no operation. It is used to indicate in an optional clause that no operation is to be performed when that clause is executed. It can be used within a **THEN** clause, an **ELSE** clause, or an **ON ERROR** clause. It is most commonly used in a **THEN** clause. For example:

```
WRITE mydata TO filevar,recID
   THEN NULL
   ELSE GOTO write_error
PRINT "All done"
```

NULL transfers control to the statement immediately following the THEN...ELSE construction.

See Also

- END statement
- IF...THEN statement

ON

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

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

Arguments

integer	A positive non-zero integer that corresponds to the list of *label*s.
label	Any valid label. The *label* 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

OPEN

Opens a MultiValue file.

```
OPEN [SECTION,] mvfile [TO filevar]
 [SETTING var] [ON ERROR statements]
 [THEN statements] [ELSE statements]
```

Arguments

SECTION	*Optional* — An expression evaluating to “DICT”, “DATA”, or “”. Unless SECTION is “DICT”, the section opened by OPEN is determined by the *mvfile* argument.
DATA DICT	*Optional* — A keyword specifying whether to access the MultiValue data file or the dictionary file. The default is to access the data file. Note the required comma following this keyword.
mvfile	An expression evaluating to a filename defined in the VOC, or an *mv filename* path. See the Description below.
filevar	*Optional* — A local variable name assigned to the MultiValue file. If omitted, the file is opened into the special variable @STDFIL.
SETTING *var*	*Optional* — When an error occurs, sets the local variable *var* to the operating system's error return code. Successful completion returns 0; error return codes are platform-specific. The SETTING clause is executed before the ON ERROR, THEN, or ELSE clause. Provided for jBASE compatibility.

Description

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

You can specify *mvfile* in any of the following ways:

```
"filename"
"filename,"
"filename,datasection"
"account,,"
"account,filename,"
"account,filename,datasection"
```

The following four **OPEN** statements all do the same thing:

```
OPEN “DATA”,”filename”
OPEN “DATA filename”
OPEN “filename”
OPEN “filename,filename”
```

The following two **OPEN** statements both do the same thing:

```
OPEN “DICT”,”filename”
OPEN “DICT filename”
```

Note the trailing comma(s) in several of these formats. `"filename"` and `"filename,"` are functionally identical. If you specify `"account,,"` the VOC for the specified *account* is opened. If *mvfile* is an empty string, **OPEN** executes its ELSE clause.

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 an ON ERROR clause, which is executed if an argument is invalid. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

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. Commonly, a STOP is issued as part of an ELSE clause. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

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.

The following example uses **OPEN** to open the VOC file:

```
OPEN 'VOC' TO MyVoc
IF 0=STATUS() THEN PRINT 'Opened file' ELSE STOP 201,'VOC'
CLOSE MyVoc
IF 0=STATUS() THEN PRINT 'Closed file' ELSE PRINT 'Close error'
```

Directory Direct Reference

You can use the double slash (//) prefix to directly reference a directory pathname from the **OPEN** command. For example:

```
OPEN "//C:/temp" TO DSCB
WRITE results ON DSCB,"results.txt"
```

This program opens the Windows directory C:/temp and creates the file C:/temp/results.txt.

The directory pathname must be preceded by //, and must not already exist in the VOC.

The DICT and DATA keywords are not meaningful in this context. Either may be specified or omitted without affecting the directory reference.

Emulation

You may specify the keyword SYSTEM before the *account* portion of *mvfile*. For example: `"SYSTEM,account,filename"`. In Caché MVBasic this keyword is a no-op, it is provided for compatibility with other MultiValue implementations.

In D3 emulation mode, you can specify MDS, rather than SYSTEM, as this no-op keyword prefix.

See Also

- READ statement
- WRITE statement
- CLOSE statement
- OPENSEQ statement
- STATUS statement
- OPENPATH statement
- FILEINFO function
- STATUS function

OPENINDEX

Opens an index.

```
OPENINDEX filename,indexname TO ivar [THEN statements] [ELSE statements]
```

Arguments

filename	The name of a MultiValue file defined in the VOC, or the name VOC. *filename* must be specified as a quoted string. If there are multiple defined data sections (data files), you can specify *filename* as `"filename,datasection"`.
indexname	The name of a defined index, specified as a quoted string.
ivar	The name of an index variable, a dynamic array.

Description

The **OPENINDEX** statement is used to open an existing index and create an *ivar* (index variable) for use by the **SELECT**, **SELECT ATKEY**, or **SELECTINDEX** statement.

The *ivar* argument accepts a single dynamic array reference (A<i>), a single substring reference (A[s,l]), or a substring reference nested inside a dynamic array reference (A<i>[s,l]).

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If the index is opened, the THEN clause is executed. If the index cannot be opened, the ELSE clause is executed. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

Examples

The following example used **OPENINDEX** to open an index to VOC on the attribute F1. The **SELECT** selects this index to a select list. The **READNEXT KEY** reads an item from the select list:

```
OPENINDEX 'VOC','F1' TO IdxFp ELSE ABORT
SELECT IdxFp TO IdxList
READNEXT KEY Idx,Id FROM IdxList
```

See Also

- SELECT statement
- SELECT ATKEY statement
- SELECTINDEX statement
- READNEXT statement
- READNEXT KEY statement

OPENPATH

Opens a directory.

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

Arguments

pathname	*Optional* — A fully-qualified pathname of a directory, specified as a quoted string. For example: “C:\foo\”
TO *filevariable*	*Optional* — A structure that contains the file type and the file name. For details, see below.
SETTING *var*	*Optional* — When an error occurs, sets the local variable *var* to the operating system's error return code. Successful completion returns 0; error return codes are platform-specific. The SETTING clause is executed before the ON ERROR, THEN, or ELSE clause. Provided for jBASE compatibility.

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.

The *filevariable* argument accepts a single dynamic array reference (A<i>), a single substring reference (A[s,l]), or a substring reference nested inside a dynamic array reference (A<i>[s,l]).

You can optionally specify an ON ERROR clause, which is executed when the directory is located but could not be opened. The ELSE clause is executed when the directory could not be located. If no ON ERROR clause is specified, the ELSE clause is executed for both types of failed access. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If the directory open succeeds, the THEN clause is executed. If the directory open fails, the ELSE clause is executed. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

See Also

- OPEN statement
- OPENSEQ statement
- READ statement
- WRITE statement
- STATUS function

OPENSEQ

Opens a file for sequential access.

```
OPENSEQ filename TO filevar [LOCKED statements]
[ON ERROR statements] [THEN statements] [ELSE statements]
```

Arguments

filename	The file to be opened. A fully-qualified Windows or UNIX® file pathname, specified as a quoted string. For two-part versions of this argument, see the Emulation section below.
TO *filevar*	A file variable name used to refer to the file in Caché MVBasic. *filevar* can be a simple variable, or can be a single dynamic array reference (A<i>), a single substring reference (A[s,l]), or a substring reference nested inside a dynamic array reference (A<i>[s,l]).

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*.

You can optionally specify a LOCKED clause, which is executed if **OPENSEQ** could not open the specified file due to lock contention. The LOCKED clause is optional, but strongly recommended; if no LOCKED clause is specified, program execution waits indefinitely for the conflicting lock to be released. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the LOCKED keyword and the first line.

You can optionally specify an ON ERROR clause, which is executed if the file could not be opened. If no ON ERROR clause is present, the ELSE clause is taken for this type of error condition. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

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. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

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. You then supply this *filevar* to **CREATE** to create the new file.

The *filename* must be a fully-qualified pathname. The directories specified in *filename* must exist for a file create to be successful. Pathnames are not case-sensitive; however, case is preserved when you specify a *filename* to create a sequential file.

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.

File Locking

Issuing **OPENSEQ** gives the process exclusive access to the specified file. An **OPENSEQ** locks the file against an **OPENSEQ** issued by any other process. This lock persists until the process that opened the file releases the lock, by issuing a **CLOSE**, a **CLOSESEQ**, or a **RELEASE** statement.

Issuing an **OPENSEQ** for a non-existent file also performs an exclusive file lock, so that your process can issue a **CREATE** to create this file. A **CLOSE** or **CLOSESEQ** releases this file lock, whether or not the file has been successfully created.

If an **OPENSEQ** without a LOCKED clause attempts to open a file already opened by another process, the **OPENSEQ** waits until the first process closes (or releases) the desired file. If an **OPENSEQ** with a LOCKED clause attempts to open a file already opened by another process, the **OPENSEQ** concludes by executing the LOCKED clause *statements*. The ELSE clause is not invoked because of lock contention.

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 *filename* specified in **OPENSEQ** for that *filevar* (*key*=2). The @FILENAME system variable also contains the *filename* 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.

Sequential File I/O Buffering

By default, sequential file I/O is performed using I/O buffering. This buffer is automatically assigned as part of the **OPENSEQ** operation. I/O buffering significantly improves overall performance, but means that write operations are not immediately applied to the sequential file.

Caché MVBasic provides two statements that override I/O buffering. The **FLUSH** statement immediately writes the current contents of the I/O buffer to the sequential file. The **NOBUF** statement disables the I/O buffer for the duration of the sequential file open. That is, all subsequent I/O write operations are immediately executed on the sequential file.

Emulation

For jBASE emulation, the *filename* argument can be specified with a two-part *path*,*filename* syntax. When executed, the two parts are concatenated together, with a delimiter added to the end of *path*, when necessary. For example, `OPENSEQ 'c:\temp\','mytest.txt' TO FD` or `OPENSEQ 'c:\temp','mytest.txt' TO FD`.

For other emulation modes, the *filename* argument can be specified with a two-part *file*,*itemID* syntax. The *file* part is a dir-type file defined in the VOC master dictionary, and the *itemID* part is an operating system file within that directory.

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:

```
filename='c:\myfiles\test1'
OPENSEQ filename TO mytest ELSE STOP 201,filename
   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
```

See Also

- CREATE statement
- DELETESEQ statement
- READSEQ statement
- WRITESEQ statement
- FLUSH statement
- NOBUF statement
- CLOSESEQ statement
- RELEASE statement
- STATUS statement
- FILEINFO function
- STATUS function
- @FILENAME system variable

$OPTIONS

Sets configuration options for MultiValue implementations.

```
$OPTIONS option
```

Arguments

option	The name of a single option, or the names of multiple options separated by spaces. Option names are not case-sensitive.

Description

The **$OPTIONS** statement provides emulation/compatibility options for the various “flavors” of MultiValue database systems for the current MVBasic program. There are two basic types of options:

- Emulation options. You specify the desired emulation, which sets multiple default values appropriate for that MultiValue “flavor”.
- Flag options that set a single specific default value, usually by turning a behavior on or off.

An **$OPTIONS** statement may specify multiple *option* values, separating the values with a blank space. If you specify an emulation option, it must be the first *option* value specified.

Emulation Options

Each option sets the appropriate configuration values for that MultiValue implementation. The following *option* database system values are supported:

Cache
D3
IN2
INFORMATION
jBASE
MVBase
PICK
PIOpen
Prime
R83
POWER95
Reality
UDPICK
Ultimate
UniData
UniVerse

$OPTIONS sets the emulation for the duration of the current MVBasic program. Emulation is specific to the current account. An **$OPTIONS** statement can only specify one emulation option.

Both “Prime” and “INFORMATION” *option* values set an emulation of “INFORMATION.” An *option* value of “Default” sets an emulation of “CACHE”. You can determine the current emulation using the SYSTEM(1001) and SYSTEM(1051) functions.

You can set the systemwide MultiValue emulation for the current account using the CEMU command line command, as described in the *Caché MultiValue Commands Reference*.

D3 emulation, by default, provides variable names that are not case-sensitive. Use of such variables is not advised when interacting with Caché CSP variables, ZEN, and other InterSystems software, all of which uses case-sensitive variables. To make D3 emulation use case-sensitive variables, specify the flag option `$OPTIONS -NO.CASE`. Refer to Chapter 13 Other Compatibility Issues in *Operational Differences between MultiValue and Caché* for further details.

Flag Options

Caché MVBasic supports many flag *option* values that affect the default behavior of individual statements or functions. These are provided to support porting or emulation of specific functional differences between the various MultiValue implementations.

To turn on (activate) a flag *option* value, specify the *option* name (`$OPTIONS CASE`). To turn off a flag *option* value, prefix the *option* name with a minus sign (`$OPTIONS -CASE`). You can specify multiple flag options, separated by blanks. The emulation option, if present, must be specified as the first option.

The specific flag option values are listed in the $OPTIONS section of *Operational Differences between MultiValue and Caché*. They are described individually in the reference page for the statement or function for which they modify default behavior.

Command Line Emulation Mode

From the MultiValue Shell, the emulation mode is specified for the current account (namespace) using the **CEMU** command line command. The initialization value is Cache. However, once **CEMU** sets an emulation for an account, that emulation is persistent across processes and Caché restart. This is the emulation mode used for the compilation and execution of an MVBasic statement from the command line.

Use the **$OPTIONS** statement to temporarily override the emulation setting established by **CEMU**. To specify an *option* setting other than the ones set by the **CEMU** emulation mode, it is necessary to specify the **$OPTIONS** statement and the MVBasic statement that it affects on the same command line.

For example, the default value for SYSTEM(33) is the contents of the command stack. To return the UniData SYSTEM(33) value (the system platform name), it is necessary to specify **$OPTIONS UniData** on the same command line. This is shown in the following Windows example:

```
USER:CEMU
Emulation for account 'USER' is 'CACHE'
USER:;PRINT SYSTEM(33)
;PRINT SYSTEM(33)þCEMU
USER:;$OPTIONS UniData ;PRINT SYSTEM(33)
Windows NT
USER:;PRINT SYSTEM(33)
;PRINT SYSTEM(33)þ;$OPTIONS Unidata ;PRINT SYSTEM(33)þ;PRINT SYSTEM(33)þCEMU
```

The following example sets a custom emulation. It begins in Cache' emulation (SYSTEM(1001)=0). **$OPTIONS** sets the emulation as PICK (SYSTEM(1001)=5) and also turn off the CASE option (-CASE). This makes local variable names not case-sensitive. The result is a PICK emulation without case sensitivity, which is a behavior otherwise only found in D3 emulation.

```
USER:CEMU
Emulation for account 'USER' is 'CACHE'
USER:;PRINT SYSTEM(1001)
0
USER:;$OPTIONS PICK -CASE ;x=123 ;PRINT SYSTEM(1001) ;PRINT x ;PRINT X
5
123
123
USER:;PRINT SYSTEM(1001)
0
```

See Also

- SYSTEM function
- $OPTIONS section in *Operational Differences between MultiValue and Caché*
- CEMU command line command in the *Caché MultiValue Commands Reference*

OUT

Displays the character(s) specified by the corresponding numeric code(s).

```
OUT int[,int2[,...]
```

Arguments

int	An integer code that corresponds to a character. You can specify a single character code or a comma-separated list of character codes.

Description

The **OUT** statement displays the specified characters on the terminal screen. Valid *int* codes include the ASCII character codes and the Unicode character codes. Codes are specified as base-10 integers. You can specify a single character, or a comma-separated list of characters. Characters specified in a list are concatenated into an output string.

The **OUT** statement display is *not* suppressed by the **HUSH** statement or the **EXECUTE** statement's CAPTURING clause.

Examples

The following example displays the character string “ABCD”:

```
OUT 65,66,67,68
```

The following example displays a character string of the first four lowercase letters of the Greek alphabet:

```
OUT 945,946,947,948
```

The following example displays the Euro currency symbol:

```
OUT 8364
```

The following example rings the bell on the terminal:

```
OUT 7
```

See Also

- EXECUTE
- HUSH

PAGE

Advances printing to the next output page.

```
PAGE [ON channel] [pagenum]
```

Arguments

ON *channel*	*Optional* — 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.
pagenum	*Optional* — An integer specifying the page number to print on the next page. Page numbering must be defined in the header or footer for this option to take effect.

Description

The **PAGE** statement advances the output device (printer or terminal) to a new page. If a header and/or a footer are defined, **PAGE** prints these on the new page. If the header and/or footer defines a page number field, **PAGE** uses the *pagenum* field to specify the page number to print on the new page.

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 current user terminal).

For **PAGE ON channel** (with *channel* < 0) to affect a print job, the **PRINTER ON** statement must have been specified. Otherwise, no operation is performed.

Before calling **PAGE**, you can use the **HEADING** and **FOOTING** statements to define the text to be printed at the top and bottom of each page. After calling **PAGE**, you can use the **PRINT** statement to specify the text to be printed on the new page.

Page Length and Number Settings

You can determine the current page length by calling `SYSTEM(3)`. You can determine the current page number by calling `SYSTEM(5)`. You can change these values by calling the **ASSIGN** statement.

See Also

- ASSIGN statement
- FOOTING statement
- HEADING statement
- PRINT statement
- PRINTER statement
- SYSTEM function

PCPERFORM

Issues an operating system command and returns to MVBasic.

```
PCPERFORM cmdstr [CAPTURING {var | NULL}]
```

Arguments

cmdstr	An operating system command to be issued to the operating system shell (command prompt). Specified as a variable or a quoted string. This string cannot exceed 248 characters in length.
CAPTURING *var*	*Optional* — a variable used to receive the operating system's response. You can specify a local variable name or the NULL keyword.

Description

The **PCPERFORM** statement is used to issue an operating system shell command from within Caché MVBasic. If you specify a CAPTURING *var* clause, the response from the operating system is returned as the value of *var*. CAPTURING NULL discards the response from the operating system.

Invoking Other Shells

You can use the **EXECUTE**, **PERFORM**, and **CHAIN** commands to issue MultiValue commands from within Caché MVBasic.

You can use the **$XECUTE** command to issue an ObjectScript command from within Caché MVBasic.

Invoking Operating System Commands from the MV Shell

You can use the DOS or SH MultiValue command line commands to issue an operating system shell command from the MultiValue Shell. For further details refer to the *Caché MultiValue Commands Reference*.

See Also

- CHAIN statement
- EXECUTE statement
- PERFORM statement
- $XECUTE statement

PERFORM

Executes a MultiValue command from a program and returns.

```
PERFORM command
```

Arguments

command	One or more MultiValue commands, each command specified as a quoted string. A string can be quoted using single quotes ('cmd arg'), double quotes ("cmd arg"), or backslashes (\cmd arg\). To specify multiple commands, separate the commands with a Field Mark ("cmd1 arg":@FM:"cmd2 arg").

Description

The **PERFORM** command executes the specified Caché MultiValue command(s), then resumes execution of the MVBasic program. It initially searches the VOC for the *command*; if the *command* is not found in the VOC, it searches the global catalog. For lookup details, refer to CATALOG in the *Caché MultiValue Commands Reference*.

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.

Emulation

In Reality and D3 emulations, the **PERFORM** command is functionally identical to the **EXECUTE** command.

In UniData and UDPICK emulations, a *command* name with an initial character of * is handled as a global name. **PERFORM** removes the leading * and then looks up the resulting command name in the global catalog in SYS.MV, rather than looking up in the VOC. If the runtime environment is not a UniData emulation, a normal VOC lookup is done on the **command* name.

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
- ObjectScript: XECUTE command

PRECISION

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

```
PRECISION int
```

Arguments

int	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 text[:]
PRINT [ON channel] text [format][:]

PRINT ON CHANNEL #channel
```

Arguments

ON *channel*	*Optional* — 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.
text	*Optional* — Any MVBasic expression that resolves to a quoted string or a numeric. You can specify a single expression or a series of expressions separated by either commas (,) or colons (:). A comma inserts a tab spacing between the two strings. A colon concatenates the two strings. If *text* is omitted, a blank line is returned.
format	*Optional* — A code specifying how to handle *text*, specified as a quoted string. This *format* is applied to the *text* that immediately precedes it. Whitespace characters may be inserted between *text* and *format*.

Description

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

To print to a printer, the PRINTER ON command must have been issued. The *channel* specifies a printer print channel as a positive integer. If *channel* is 0 or -1, the *text* is printed to the terminal screen (or to a CAPTURING clause), regardless of whether **PRINTER ON** has been set.

A *text* can consist of a single string or numeric expression, or a series of expressions alternating with separator characters. Any *text* may be followed by an optional *format*. This *format* applies only to the *text* string that immediately precedes it.

The following separators are supported:

- A comma (,) used as a separator character inserts a predefined tab between to items. By default, tabs are set at ten column intervals. You can specify a comma before the first expression to indent that expression. You cannot specify a comma after the last expression; this results in a syntax error. You can specify a series of commas to specify multiple tabs; an odd number of commas increments the number of tabs. Thus, one or two commas (`exp,exp` or `exp,,exp`) equals one tab, three or four commas (`exp,,,exp` or `exp,,,,exp`) equals two tabs, and so forth.
- A colon (:) used as a separator character concatenates two items. Specifying a colon before the first expression has no effect. Specifying a colon after the last expression enables concatenation of the results of two commands. 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.

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

Formatting

The optional *format* argument specifies how to handle *text*. **PRINT** supports three types of *format* arguments:

- @ function formatting
- implicit formatting, using FMT function codes

- implicit conversion, using OCONV function codes

You can use an @ function with positive arguments to specify the column position and/or line position at which to print. For example, `CRT @(15):"Over here!"` prints the literal string starting at column 16. You can also use the @ function with negative arguments to change screen display modes. For example, `CRT @(-1):"Over here!"` clears the screen, then prints the literal string at line 1, column 1.

To advance to the next page, and to print defined headings and footings, use the **PAGE** statement.

You can use the optional *format* argument to specify display width, justification, fill characters, and zero filling or rounding for decimal digits. This is known as "implicit formatting" because it is equivalent to inserting a **FMT** function as one of the **PRINT** arguments. For further details on the available *format* codes, refer to the FMT function.

You can disable implicit formatting by specifying `$OPTIONS NO.IMPLICIT.FMT`. Specifying this option prevents the evaluation of the *format* argument in CRT, PRINT, or DISPLAY. It has no effect on the explicit use of the **FMT** function.

Implicit conversion performs many of the OCONV function conversions by specifying the conversion code as the *format* argument. For example, both of the following perform date conversion from internal to display format:

```
PRINT 14100 "D";          ! "08 AUG 2006"
PRINT OCONV(14100,"D");   ! "08 AUG 2006"
```

For further details on the available *format* conversion codes, refer to the OCONV function.

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!
```

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

returns:

```
helloworld!
```

Emulation

For Caché and most emulations, if *channel* is a positive integer, **PRINT** output always goes to a spooler print job, regardless of the use of **PRINTER ON** or **PRINTER OFF**. D3, jBASE, and Reality emulations send the **PRINT** output to the screen if the application has not executed a **PRINTER ON** statement.

See Also

- CRT statement
- DISPLAY statement
- ECHO statement
- PAGE statement
- PRINTER ON statement
- @ function
- FMT function
- SPACE function
- SPOOLER function

PRINTER

Specifies whether to direct output to the printer.

```
PRINTER ON
PRINTER OFF
PRINTER CLOSE [ON nnn]
```

Arguments

ON *nnn*	*Optional* — The print channel as assigned by the **PRINT** statement. Specified as an integer value in the range 0 through 255. If omitted, the default is print channel 0.

Description

PRINTER ON directs output to the printer. After setting this option, **PRINT** statements direct their output to the print buffer, with the exception of **PRINT ON 0** or **PRINT ON -1** which always output to the terminal screen. **PRINTER ON** has no effect on **CRT** statements, which always output to the terminal screen.

PRINTER CLOSE spools the print buffer to the printer and closes the print channel. The ON *nnn* clause allows you to specify which print channel. If this clause is omitted, Caché MVBasic closes print channel 0; this behavior is emulation-dependent. An implicit **PRINTER CLOSE** is issued when the program terminates.

PRINTER OFF directs subsequent output to the screen (the default output device).

Note: PRINTER RESET is listed in this manual as a separate command, not an option of the **PRINTER** command.

Emulation

PRINTER CLOSE with no argument closes only print channel 0 in Caché and in UniVerse emulation (and some other emulations). In PICK and Reality (and some other emulations) **PRINTER CLOSE** with no argument closes all print channels. This behavior is governed by the SP-CONDUCT bit mask 4096.

See Also

- PRINT statement
- SPOOLER function

PRINTER RESET

Resets terminal or default printer channel characteristics.

```
PRINTER RESET
```

Description

PRINTER RESET resets the header, footer, and line number characteristics. It resets these characteristics for terminal output if output is directed to the terminal. It resets these characteristics for printer output if output is directed to the default print channel. Where output is directed is specified using the PRINTER command.

PRINTER RESET resets the following:

- The page header to null, removing any header set by the HEADING command.
- The page footer to null, removing any footer set by the FOOTING command.
- The current line number, resetting the value of SYSTEM(4).

See Also

- HEADING statement
- FOOTING statement
- SPOOLER function

PROCREAD

When called by a procedure, reads the input buffer contents.

```
PROCREAD data [THEN statements] [ELSE statements]
```

Arguments

data	Name of a variable used to receive PROC data from the input buffer.

Description

The **PROCREAD** statement reads the results of a PROC from the primary input buffer into the *data* variable. The MVBasic program must have been called by a procedure for **PROCREAD** to execute successfully.

The *data* variable must be simple variable name. It cannot include a system variable, an **EQUATE**, a dynamic array reference, or a substring reference.

When reading from a PQ PROC, **PROCREAD** converts the @AM buffer delimiter to a blank space. **PROCREAD** converts empty buffer entries to the backslash (\) character. **PROCWRITE** reverses these character conversions. For this reason, including a \ literal in the input buffer should be avoided. When reading from a PQN PROC, **PROCREAD** does not perform these character conversions.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If the input buffer read is successful, the THEN clause is executed, even when the contents of the input buffer is the empty string. If input buffer read operation fails, the ELSE clause is executed. If the program containing **PROCREAD** was not called by a procedure, the read operation fails and the ELSE clause is executed. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

Emulation

D3 and several other emulations support **$OPTIONS READ.RETAIN**. This option causes the *data* variable to retain its original value if **PROCREAD** fails.

See Also

- PROCWRITE statement
- Caché MultiValue PROC Reference

PROCWRITE

When called by a procedure, writes to the input buffer.

```
PROCWRITE data
```

Arguments

data	Name of a variable used to contain data to be written to the input buffer.

Description

The **PROCWRITE** statement writes the contents of *data* to a PROC using the primary input buffer. The MVBasic program must have been called by a procedure for **PROCWRITE** to execute successfully.

Following the write operation, **PROCWRITE** resets the primary input buffer pointer so that it points to the beginning of the data in the buffer.

For a PQ PROC, **PROCWRITE** converts each blank space in *data* to an @AM delimiter, reversing the **PROCREAD** operation. **PROCWRITE** converts each elements that consists of a backslash (\) to an empty element, reversing the **PROCREAD** operation. Because of this backslash conversion, a backslash literal transferred by **PROCREAD** will be transferred back by **PROCWRITE** as an empty element. For this reason, backslashes in the input buffer should be avoided. This **PROCWRITE** conversion only applies to backslashes that represent empty elements (a backslash delimited by blank spaces). If a backslash is appended to other characters, **PROCWRITE** treats it as a quote delimiter.

When writing to a PQN PROC, **PROCWRITE** does not perform these character conversions.

Emulation

In Ultimate emulation, **PROCWRITE** converts all attribute marks (@AM) to blank spaces.

See Also

- PROCREAD statement
- Caché MultiValue PROC Reference

PROG (PROGRAM)

Specifies the program name.

```
PROG name
PROGRAM name
```

Arguments

name	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

string	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

number	*Optional* — Any valid numeric expression.

Description

The **RANDOMIZE** statement uses *number* to initialize the **RND** function's random-number generator, giving it a seed value. By specifying the same **RANDOMIZE** *number* seed, you can use **RND** to repeatedly generate the same “random” number.

To restore true randomness, issue a **RANDOMIZE** statement without the *number* argument. 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 uses the system internal clock as a seed the first time it is called, and thereafter uses the last generated random number as the next seed value.

Examples

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

```
RANDOMIZE 10;  ! Seeds random-number generator
PRINT RND(7)
  ! Generates a random value between 1 and 6
PRINT RND(7)
  ! Generates the same "random" value as above
RANDOMIZE;     ! Restores randomness
PRINT RND(7)
  ! Generates a random value between 1 and 6
```

See Also

- RND function

READ, READL, READU, READV, READVL, READVU

Reads data from a MultiValue file.

```
READ dynarray FROM filevar,recID
   [SETTING var] [ON ERROR statements] [[THEN statements] [ELSE statements]]

READL dynarray FROM filevar,recID
   [ON ERROR statements] [LOCKED statements] [[THEN statements] [ELSE statements]]

READU dynarray FROM filevar,recID
    [SETTING var] [ON ERROR statements] [LOCKED statements] [[THEN statements] [ELSE
 statements]]

READV dynarray FROM filevar,recID,fieldno
   [SETTING var] [ON ERROR statements] [[THEN statements] [ELSE statements]]

READVL dynarray FROM filevar,recID,fieldno
   [ON ERROR statements] [LOCKED statements] [[THEN statements] [ELSE statements]]

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

Arguments

dynarray	A dynamic array used to receive the field values from the file. This argument may be a local variable or an object reference.
filevar	A local variable used as the file identifier of an open MultiValue file. This variable is set by the **OPEN** statement.
recID	The record ID of the record to be read, specified as either a number or an alphanumeric string of up to 31 characters. Letters in a *recID* are case-sensitive. For naming conventions, refer to WRITE.
fieldno	The field number of the field to be read, specified as an integer. Used with **READV** and **READVU**. If 0, returns the *recID*.
SETTING *var*	*Optional* — When an error occurs, sets the local variable *var* to the operating system's error return code. Successful completion returns 0; error return codes are platform-specific. The SETTING clause is executed before the ON ERROR, THEN, or ELSE clause. Provided for jBASE compatibility.

Description

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

The *dynarray* argument accepts a single dynamic array reference (A<i>), a single substring reference (A[s,l]), or a substring reference nested inside a dynamic array reference (A<i>[s,l]).

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

A read operation must be able to acquire at least a shared lock on the desired resource. The **READL** and **READVL** statements acquire a shared lock before performing the read. The **READU** and **READVU** statements acquire an update lock before performing the read.

You can optionally specify a LOCKED clause for **READL**, **READU**, **READVL**, and **READVU**. This clause is executed if the statement could not acquire the desired resource due to lock contention. The LOCKED clause is optional, but strongly recommended; if no LOCKED clause is specified, program execution waits indefinitely for the conflicting lock to be released. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the LOCKED keyword and the first line.

You can optionally specify an ON ERROR clause, which is executed if an argument is invalid. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If the read is successful, the THEN clause is executed. The THEN clause is executed even when all remaining field identifiers are the null string. If read cannot read the specified record, the ELSE clause is executed. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

Reading a Record

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

Reading a Field

READV, **READVL**, and **READVU** all read the specified field value from the specified MultiValue file record into *dynarray*. They do this by locating field delimiters in the record string. If *fieldno* is 0, these statements returns the specified *recID* to *dynarray*. If *fieldno* refers to a non-existent field (does not correspond to a field delimiter), these statements returns the null string to *dynarray*. If *fieldno* is 1 and the entire record consists of a single numeric value (and thus contains no field delimiters), these statements return that numeric value.

If *recID* refers to a non-existent record and *fieldno* is not 0, the read operation fails.

Reading to an Object

The *dynarray* argument can be an object reference, allowing **READ** to read data into an object. The following statements are all valid forms of reading to an object:

```
READ @me->prop FROM myfile,1
READ "class"->meth()->prop FROM myfile,1
READ obj->prop FROM myfile,1
READ (obj)->prop FROM myfile,1
```

READ and MATREAD

The various **READ** statements read from a MultiValue file into a dynamic array. The various **MATREAD** statements read from a MultiValue file into a dimensioned array.

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
- MATREAD statement
- WRITE statement
- CLOSE statement
- STATUS function
- Dynamic Arrays

READBLK

Reads a block of data from a sequential file.

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

Arguments

data	Name of a variable used to receive a block of data from a file.
filevar	A file variable name used to refer to the file in Caché MVBasic. This *filevar* is obtained from **OPENSEQ**.
blksize	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.

The *data* argument accepts a single dynamic array reference (A<i>), a single substring reference (A[s,l]), or a substring reference nested inside a dynamic array reference (A<i>[s,l]).

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. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

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]
READLIST dynarray FROM listname [account] [THEN statements] [ELSE statements]
```

Arguments

dynarray	A dynamic array used to receive the field values from the select list.
slist	An active select list, identified by number or name. A numbered select list is specified as an integer from 0 through 10. A named select list is specified as a variable name.
listname account	A saved select list, identified by its assigned *listname* record ID. If the saved select list is in the current account, omit *account*. If the saved select list is in another account, specify the account name, separating *listname* and *account* with a space character.

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 any of the following **SELECT** statements to create a select list: **SELECT**, **SELECTN**, **SELECTV**, **SSELECT**, **SSELECTN**, or **SSELECTV**. These various **SELECT** statements allow you to specify a numbered or named select list, with field identifiers either sorted or not sorted.

The *listname* select list is saved in the &SAVEDLISTS& file. Caché stores this file using the ^SAVEDLISTS global.

The *dynarray* variable must be simple variable name. It cannot include a system variable, an **EQUATE**, a dynamic array reference, or a substring reference.

If an error occurs during **READLIST** processing, Caché sets the *dynarray* variable to the null string ("").

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If the select list pointer has not reached the end of the select list, **READLIST** executes the THEN clause. 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. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

Unlike **READNEXT** and **READPREV**, **READLIST** does not clear the select list when it reaches the end of the select list. For this reason, you can follow a **READLIST** statement with a **READPREV** to read individual field identifiers backwards from the end of the select list.

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
- SSELECT statement
- READNEXT statement
- READPREV statement
- Dynamic Arrays

READNEXT

Reads the next field id from a select list.

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

Arguments

fieldval	A variable used to receive a field value from the select list. Optionally, this can be a multilevel specification, with the levels separated by commas: *field*,*value* or *field*,*value*,*subvalue*.
slist	*Optional* — 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. If *slist* is not specified or is the empty string (""), the default select list (0) is accessed.
SETTING *var*	*Optional* — When a read error occurs, sets the local variable *var* to the operating system's error return code. Successful completion returns 0; error return codes are platform-specific. The SETTING clause is executed before the THEN, or ELSE clause. Provided for jBASE compatibility.

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 *fieldval* variable. Optionally, **READNEXT** can be used to read successive values or successive subvalues within a field, by specifying a multilevel *fieldval* variable. (**READNEXT** can also be used to read successive index identifiers; this is described below.)

You can use any of the following **SELECT** statements to create a select list: **SELECT**, **SELECTN**, **SELECTV**, **SSELECT**, **SSELECTN**, or **SSELECTV**. These various **SELECT** statements allow you to specify a numbered or named select list, with field identifiers either sorted or not sorted.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. **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. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

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.

READNEXT reads the next field identifier in a select list. **READPREV** reads the previous field identifier in a select list. If **READNEXT** reaches the end of the select list, it clear the select list. For this reason, a subsequent **READPREV** cannot read backwards from the end of the select list.

Reading an Index

You can use **READNEXT** to perform successive reads on an index. The index must have been opened using an **OPENINDEX** statement. and then selected into a named select list with a **SELECTINDEX** statement.

You can also perform successive reads on an index using the **READNEXT KEY** statement.

Emulation

Caché MVBasic, by default, uses select list 0 as the default select list for both internal and external use. By default, D3, Reality, R83, POWER95, MVBase, and IN2 emulations use two distinct default select lists, one internal and one external. The default external select list is 0, and the default internal select list is 10. When **READNEXT** first accesses the external select list (list 0), it moves this list to the internal select list (10). Thus subsequent **READNEXT** operations can continue to access this select list, regardless of modifications to list 0. This emulation behavior can be set using **$OPTIONS PICK.SELECT**.

UniData sets SYSTEM(11) to the **SELECT** count when using Select List 0. Each invocation of **READNEXT** decrements this SYSTEM(11) count. **READNEXT** does not decrement the @SELECTED count.

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

- READPREV statement
- SELECT statement
- SSELECT statement
- SELECTINDEX statement
- READNEXT KEY statement
- GETLIST statement
- READLIST statement
- Dynamic Arrays

READNEXT KEY

Reads the next key and item id from an index.

```
READNEXT KEY keyname,itemID FROM slist [THEN statements] [ELSE statements]
```

Arguments

keyname	A variable used to receive the key name from the select list.
itemID	A variable used to receive the key item ID from the select list.
slist	A select list to an existing index. A named select list specified as a variable name.

Description

The **READNEXT KEY** statement reads successive key identifiers from a select list, one key identifier per invocation. **READNEXT KEY** returns both the key name and the key item ID. The key identifier is read from the *slist* select list into the *itemID* variable.

READNEXT KEY is used on a select list created by either a **SELECT** or a **SELECT ATKEY**.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. **READNEXT KEY** 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 key identifier is the null string. **READNEXT KEY** executes the ELSE clause if the select list pointer has reached the end of the select list, or the select list does not exist. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

Examples

The following example used **OPENINDEX** to open an index to VOC on the attribute F1. The **SELECT** selects this index to a select list. The **READNEXT KEY** reads an item from the select list:

```
OPENINDEX 'VOC','F1' TO Idx.Fp ELSE ABORT
SELECT Idx.Fp TO Idx.List
READNEXT KEY Idx,Id FROM Idx.List
```

See Also

- OPENINDEX statement
- SELECT statement
- SELECT ATKEY statement
- SELECTINDEX statement
- READNEXT statement

READPREV

Reads the previous field id from a select list.

```
READPREV fieldval [FROM slist]
    [SETTING var] [[THEN statements] [ELSE statements]]
```

Arguments

fieldval	A variable used to receive a field value from the select list. Optionally, this can be a multilevel specification, with the levels separated by commas: *field*,*value* or *field*,*value*,*subvalue*.
slist	*Optional* — 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. If not specified, the default select list (0) is accessed.
SETTING *var*	*Optional* — When a read error occurs, sets the local variable *var* to the operating system's error return code. Successful completion returns 0; error return codes are platform-specific. The SETTING clause is executed before the THEN or ELSE clause. Provided for jBASE compatibility.

Description

The **READPREV** statement reads successive field identifiers from a select list in reverse order, one field identifier per invocation. The field identifier is read from the *slist* select list into the *fieldval* variable. Optionally, **READPREV** can be used to read successive values or successive subvalues within a field (in reverse order), by specifying a multilevel *fieldval* variable.

READPREV reads the previous field identifier in a select list. **READNEXT** reads the next field identifier in a select list.

You can use any of the following **SELECT** statements to create a select list: **SELECT**, **SELECTN**, **SELECTV**, **SSELECT**, **SSELECTN**, or **SSELECTV**. These various **SELECT** statements allow you to specify a numbered or named select list, with field identifiers either sorted or not sorted.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. **READPREV** executes the THEN clause if the select list pointer has not reached the beginning of the select list. The THEN clause is executed even when a field identifier is the null string. **READPREV** executes the ELSE clause if the select list pointer has reached the beginning of the select list, or the select list does not exist. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

If **READPREV** reaches the beginning of the select list, it clears the select list. For this reason, a subsequent **READNEXT** cannot read the first item on the select list. Similarly, if a **READNEXT** reads the last item of a select list, the list is cleared. A subsequent **READPREV** cannot be used to read backwards from the end of the select list.

READPREV and **READNEXT** read 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. **READLIST** does not clear the select list. Therefore, you can follow a **READLIST** with a **READPREV** to read the last field identifier in the select list.

Examples

The following example illustrates the use of the **READPREV** statement. **SELECT** copies all of the field mark identifiers into Select List 4. **READNEXT** reads the next field mark identifier from Select List 4 into the *area* variable. **READPREV** reads the previous 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"
READNEXT area FROM 4
  PRINT area;  ! returns "Northeast"
READNEXT area FROM 4
  PRINT area;  ! returns "Southeast"
READPREV area FROM 4
  PRINT area;  ! returns "Northeast"
```

The following example uses **READLIST** to advance to the end of the select list, and then uses **READPREV** to read the last item in the select list:

```
regions="Northeast":@FM:"Southeast":@FM:"Northwest":@FM:"Southwest"
SELECT regions TO 4 ON ERROR PRINT "Select failed"
READLIST area FROM 4
  PRINT area;  ! returns "Northeast^Southeast^Northwest^Southwest"
READPREV area FROM 4
  PRINT area;  ! returns "Southwest"
```

See Also

- READLIST statement
- READNEXT statement
- SELECT statement
- SSELECT statement
- GETLIST statement
- Dynamic Arrays

READSEQ

Reads a line of data from a sequential file.

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

Arguments

data	Name of a variable used to receive a line of data from a file.
FROM *filevar*	A file variable name used to refer to the file in Caché MVBasic. This *filevar* 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.

The *data* argument accepts a single dynamic array reference (A<i>), a single substring reference (A[s,l]), or a substring reference nested inside a dynamic array reference (A<i>[s,l]).

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 an ON ERROR clause, which is executed if the file is located but could not be read. If no ON ERROR clause is present, the ELSE clause is taken for this type of error condition. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

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. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

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

RECORDLOCKL, RECORDLOCKU

Locks a record in a MultiValue file.

```
RECORDLOCKL filevar, recID [ON ERROR statements] [LOCKED statements]
RECORDLOCKU filevar, recID [ON ERROR statements] [LOCKED statements]
```

Arguments

filevar	A file variable name used to refer to a MultiValue file. This *filevar* is supplied by the **OPEN** statement.
recID	The record ID of a record to be locked, specified as an integer.

Description

The **RECORDLOCK** statements are used to lock a record in a MultiValue file.

- **RECORDLOCKL** performs a shared lock on a record. It permits other users to also get a **RECORDLOCKL** on the record, but prevents other uses from getting a **RECORDLOCKU** on the record or an exclusive **FILELOCK** on the file.
- **RECORDLOCKU** performs an update (exclusive) lock on a record. It prevents other uses from getting a **RECORDLOCKL** or **RECORDLOCKU** on the record or a **FILELOCK** of any type on the file.

The **RECORDLOCK** statements take the file identifier *filevar*, defined by the **OPEN** statement.

You can optionally specify a LOCKED clause. This clause is executed if the record to be locked has already been locked by another user. The clause is executed if the level of lock requested conflicts with an existing lock. This clause is optional, but strongly recommended; if no LOCKED clause is specified, program execution waits indefinitely for the conflicting lock to be released. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the LOCKED keyword and the first line.

You can optionally specify an ON ERROR clause. If file lock fails, the ON ERROR clause is executed. This may occur if *filevar* does not refer to a currently open file. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

You can check the status of file locks and record locks using the RECORDLOCKED function.

Lock Promotion

If you have a shared lock on a record, then request an exclusive (update) lock on the same record, MVBasic attempts to get the exclusive lock. If it is successful, your shared lock is promoted to an exclusive lock. The result is that you hold one exclusive lock, not two locks.

Releasing Record Locks

Use **RELEASE** to release individual record locks. **CLOSE** releases all record locks held on the specified file. **ABORT** and **STOP** release all record locks held by the current process.

An update record lock is automatically released when you write data to the record using **WRITE** or **WRITEV**. The **WRITEU** and **WRITEVU** commands do not release the update record lock.

An update record lock is automatically released when you delete the record using **DELETE**. The **DELETEU** command does not release the update record lock.

See Also

- FILELOCK statement
- OPEN statement
- RELEASE statement
- RECORDLOCKED function

RELEASE

Releases record locks.

```
RELEASE [filevar [,recID]] [ON ERROR statements]
```

Arguments

filevar	*Optional* — A file variable name used to refer to a MultiValue file. This *filevar* is supplied by the **OPEN** statement.
recID	*Optional* — The record ID for which record locks are to be released. If not specified, all record locks and the file lock on *filevar* are released.

Description

A **RELEASE** statement with no argument releases all record locks held by the current process that were applied at the current @LEVEL execution level. (This differs from native UniData behavior, which releases all locks held by the current process on all levels.)

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.

Records are locked using the RECORDLOCKU and RECORDLOCKL statements. You can check the status of record locks (and file locks) using the RECORDLOCKED function.

An update record lock is automatically released when you write data to the record using **WRITE** or **WRITEV**. The **WRITEU** and **WRITEVU** commands do not release the update record lock.

An update record lock is automatically released when you delete the record using **DELETE**. The **DELETEU** command does not release the update record lock.

CLOSE releases all record locks held on the specified file. **ABORT** and **STOP** release all record locks held by the current process.

A file is locked using the FILELOCK statement. **RELEASE** with no *recID* can be used to release a locked file. This is equivalent to issuing a FILEUNLOCK statement.

You can optionally specify an ON ERROR clause. If a record lock release fails, the ON ERROR clause is executed. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

See Also

- OPEN statement
- FILEUNLOCK statement
- RECORDLOCKL statement
- RECORDLOCKU statement
- RECORDLOCKED function

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 value FROM dynarray [AT pos] SETTING delim
```

Arguments

value	A variable used to receive the extracted element value.
dynarray	A dynamic array from which successive data values are to be extracted.
AT *pos*	*Optional* — A variable specifying the initial starting position in *dynarray* as an integer character count. *pos* must be specified as a local variable, not as a numeric literal. The AT clause is provided for compatibility with D3 and UniData systems.
delim	A local variable that resolves to an integer code for the dynamic array delimiter type. *delim* must be specified as a local variable, not as a numeric literal. *delim* can accept a single dynamic array reference (A<i>), a single substring reference (A[s,l]), or a substring reference nested inside a dynamic array reference (A<i>[s,l]).

Description

The **REMOVE** statement efficiently extracts successive data values from a dynamic array. The extracted element value is placed in the *value* variable. **REMOVE** operates on a single dynamic array level; you specify the level delimiter using the *delim* argument. **REMOVE** maintains an internal pointer so that repeated calls return successive element values. When the last element value has been extracted, **REMOVE** sets *value* 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 *delim* variable resolves to an integer code with one of the following values:

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

See Also

- REVREMOVE statement
- SETREM statement
- EXTRACT function
- GETREM function
- REMOVE function

RETURN

Returns from a subroutine or function.

```
RETURN[(retval)] [TO label]
```

Arguments

retval	*Optional (Functions Only)* — An expression that evaluates to the return value for a user-defined function. The return value must be enclosed in parentheses. If not specified, an empty string is returned.
TO *label*	*Optional (Subroutines Only)* — Any valid label. The *label* name can be optionally followed by a colon (:)

Description

The **RETURN** statement is used to end execution of a user-defined subroutine or function and return control to the statement that invoked the subroutine or function.

Subroutines

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.

Functions

The **RETURN** statement ceases execution of a function and returns *retval* to the location where the function was invoked. If no *retval* is specified, an empty string is returned.

Before invoking a user-defined external function, it is necessary to locally define the function using the **DEFFUN** statement.

See Also

- CALL statement
- GOSUB statement
- END statement
- FUNCTION statement
- Labels

REVREMOVE

Extracts sequential elements of a dynamic array in reverse order.

```
REVREMOVE value FROM dynarray SETTING delim
```

Arguments

value	A variable used to receive the extracted element value.
dynarray	A dynamic array from which successive data values are to be extracted.
delim	A local variable that resolves to an integer code for the dynamic array delimiter type. *delim* must be specified as a local variable, not as a numeric literal. *delim* can accept a single dynamic array reference (A<i>), a single substring reference (A[s,l]), or a substring reference nested inside a dynamic array reference (A<i>[s,l]).

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 *value* variable. **REVREMOVE** operates on a single dynamic array level; you specify the level delimiter using the *delim* 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 *value* 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 *delim* variable resolves to an integer code with one of the following values:

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
- SETREM statement
- EXTRACT function
- GETREM function
- REMOVE function

ROLLBACK

Reverts all changes made during the current transaction.

```
ROLLBACK [TRANSACTION | 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 **ROLLBACK** must be specified between the **BEGIN TRANSACTION** and **END TRANSACTION** statements. Following a **ROLLBACK**, program execution skips to the line of code following the **END TRANSACTION** statement.

The TRANSACTION or WORK keywords are optional and provides no functionality. They are 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. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

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.

Locks and Transactions

File locks and record locks that were taken out during a transaction are released at the end of a transaction. If there are nested transactions, the release of locks taken out during the inner transactions is delayed until the completion of the outermost transaction. This release of locks is part of a successful **COMMIT** or **ROLLBACK** operation. Locks are described in the LOCK statement.

Unaffected by ROLLBACK

- The contents of spooler and form queues, and any print jobs queued or in progress.
- The contents of the &PH& file and any spawned PHANTOM (background) processes.
- The contents of the &COMO& file used to keep an audit trail of terminal inputs.

Example

The following example performs database operations within a transaction. It sets a variable x, which determines whether the transaction should be committed or rolled back.

```
PRINT "Before the transaction"
BEGIN TRANSACTION
 .
 .
 .
IF x=0
  THEN COMMIT
  END
  ELSE ROLLBACK
    THEN PRINT "Rollback successful"
    ELSE PRINT "Rollback failed"
  END
PRINT "This should not print"
END TRANSACTION
PRINT "Transaction resolved"
```

See Also

- BEGIN TRANSACTION statement
- END TRANSACTION statement
- COMMIT statement

RQM

Suspends processing for a specified duration.

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

Arguments

seconds	*Optional* — An integer count of seconds. If omitted, execution is suspended for 1 second.
time	A wakeup time, specified in 24-hour format as hh:mm[:ss], or in 12-hour format as hh:mm[:ss]AM or hh:mm[:ss]PM.

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. You can specify *seconds* as an integer or a fraction. If *seconds* is a decimal number, it is rounded to the nearest whole second.

You can specify *time* in either 24-hour or 12-hour format. A 24-hour time is specified as hh:mm[:ss]. A 12-hour time is specified as hh:mm[:ss]{AM | PM}. In both formats, spaces are not permitted, leading zeros may be omitted, and the seconds component of the time is optional. The following are all valid 24-hour format *time* values: 02:34, 2:34:00, 14:34, 14:34:00. The following are all valid 12-hour format *time* values: 2:34PM, 02:34PM, 2:34:00PM. 2:34AM. Midnight can be represented by 24:00, 00:00, 12:00PM, 00:00PM, or 00:00AM. An invalid *time* argument generates a syntax error.

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
- SLEEP command in *Caché MultiValue Commands Reference*

SEEK

Repositions the file pointer for a sequential file.

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

Arguments

filevar	A file variable name used to refer to the file in Caché MVBasic. This *filevar* is obtained from **OPENSEQ**.
offset	*Optional* — A positive or negative integer count of bytes used to reposition the file pointer relative to the *relto* position. By default, *offset* is 0.
relto	*Optional* — 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. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

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

SEEK(ARG.)

Points to the next command line argument.

```
SEEK(ARG.[,n]) [THEN statements] [ELSE statements]
```

Arguments

n	*Optional* — An integer specifying which command line argument to point to. The default is the first unread argument (the next argument).

Description

The **SEEK(ARG.)** statement points to a command line argument. Each time you invoke **SEEK(ARG.)** it updates a command line pointer. Therefore, repeated invocation of **SEEK(ARG.)** without the *n* argument results in sequentially pointing to each command line argument in left-to-right order.

The keyword `ARG.` (note the period at end of this keyword) and the surrounding parentheses are mandatory.

You can use the optional *n* value to point to a command line argument by its integer position in the command line argument list. Command line arguments are counted from 1. If *n*=0, **SEEK(ARG.)** points to the next command line argument.

SEEK(ARG.) considers all values following the program name to be command line arguments. Command line arguments are separated by blank spaces; a blank space within a quoted string is not treated as a command line argument separator.

You can optionally specify a THEN clause, an ELSE clause, or both a THEN and an ELSE clause. If moving the pointer to the command line argument is successful, the THEN clause is executed. If there are no command line arguments, no more command line arguments, or if you specify a value of *n* that does not correspond to a command line argument, **SEEK(ARG.)** executes the ELSE clause. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

The **GET(ARG.)** statement both moves the command line argument pointer and retrieves the argument value. The **SEEK(ARG.)** statement just moves the command line argument pointer. The **EOF(ARG.)** function returns whether or not the command line argument pointer is past the end of the list of command line arguments.

See Also

- GET(ARG.) statement
- EOF(ARG.) function

SELECT, SELECTN, SELECTV

Selects items into a select list.

```
SELECT dynarray [TO listnum] [SETTING var] [ON ERROR statements]
SELECT [filevar] [TO listnum] [SETTING var] [ON ERROR statements]
SELECT dynarray TO listname [SETTING var] [ON ERROR statements]
SELECT [filevar] TO listname [SETTING var] [ON ERROR statements]

SELECTN dynarray [TO listnum] [ON ERROR statements]
SELECTN [filevar] [TO listnum] [ON ERROR statements]

SELECTV dynarray TO listname [ON ERROR statements]
SELECTV [filevar] TO listname [ON ERROR statements]
```

Arguments

dynarray	Any valid dynamic array of Field Values.
filevar	*Optional* — A local variable used as the file identifier of an open MultiValue file. This variable is set by the **OPEN** statement. If omitted, the default file variable is used.
TO *listnum*	*Optional* — A numbered select list, specified as an integer from 0 through 10. You must specify a *listnum* from 1 through 10; *listnum* 0 is not valid for Caché MVBasic. If omitted, select list 0 is used.
TO *listname*	A named select list, specified as a local variable name. (See Emulation section below.)
SETTING *var*	*Optional* — When an error occurs, sets the local variable *var* to the operating system's error return code. Successful completion returns 0; error return codes are platform-specific. The SETTING clause is executed before the ON ERROR clause. Provided for jBASE compatibility.

Description

The **SELECT** statements select the field identifiers from a MultiValue file or a dynamic array and place 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.

- **SELECT** can select into a numbered select list or a named select list. (See “Emulation” section below.)
- **SELECTN** can only select into a numbered select list.
- **SELECTV** can only select into a named select list.

Note: You can use SELECTE to copy numbered Select List 0 to a named select list.

Unless otherwise stated, all documented **SELECT** behavior also applies to **SELECTN** and **SELECTV**.

SELECT statements sort the contents of a select list or file into Caché storage order: first the empty string, then canonical numbers in ascending numeric order, then strings in string collation order. They then place the results in a select list. **SELECT** does not sort the contents of a file system directory or the elements of a dynamic array. These are copied into a select list in the order listed. To sort dynamic array elements, first use **SELECT** to copy the elements into a select list, then use **SELECT** or **SSELECT** on that 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. The *statements* argument can be the

NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

When you are finished using an assigned select list, you can use the **CLEARSELECT** statement to reset the select list.

Note: **SELECT** and FORMLIST are functionally identical.

SELECT filevar

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.

SELECT filevar places a direct reference to *filevar* in the newly created select list; the select list does not contain a copy of the file contents. Therefore, any subsequent change to the *filevar* file will immediately change the contents of this previously selected select list. To avoid this, you can specify $OPTIONS FSELECT before invoking **SELECT filevar**. If the FSELECT flag option is activated, **SELECT filevar** creates a select list containing a copy of the indices from the file referenced by *filevar*; subsequent changes to the *filevar* file will have no effect on the contents of this previously selected select list. The FSELECT option is off (inactive) by default. Using the FSELECT option makes the length of the select list immediately available after executing the **SELECT filevar** statement. However, performing a **SELECT filevar** with the FSELECT option enabled has poorer performance than a **SELECT filevar** without FSELECT.

Note: Use of FSELECT only applies to a **SELECT filevar** executed directly as an MVBasic statement. Indirect execution of **SELECT filevar**, such as **EXECUTE 'SELECT filevar'**, does not apply the FSELECT setting. Executing **SELECT filevar** from the MultiValue Shell does not apply the FSELECT setting.

Selecting an Index

You can use **SELECT** with a named select list to select an entire index. The index must have been opened using an **OPENINDEX** statement. After selecting the index, you can read individual index items using the **READNEXT KEY** statement.

If you wish to select only part of an index, you can use the **SELECT ATKEY** or **SELECTINDEX** statement.

SELECT and SSELECT

The SSELECT (sorted select) statements sort in ordinary string collation order. **SSELECT filevar** always creates a select list containing a copy of the indices from the file referenced by *filevar*, regardless of the **$OPTIONS FSELECT** setting. The **SELECT** statements are otherwise comparable to the corresponding **SSELECT** statements.

Emulation

By default, **SELECT** can select to a numbered select list or to a named select list. Any TO clause variable that resolves to an integer from 0 through 10 is treated as a numbered select list; any other value is treated as a named select list. **SELECT** uses select list 0 as the default select list for both internal and external use. These are the defaults for Caché, jBASE, and UniData emulation.

D3, IN2, MVBase, PICK, R83, POWER95, Reality, and Ultimate set **$OPTIONS VAR.SELECT**. This requires that the select list specified in the TO clause must be a named select list; **SELECT** behaves like **SELECTV**. These emulations return an error when you specify a numeric value for the TO clause. Select List 0 is used as the default when you omit the TO clause.

INFORMATION, PIOpen, Prime, and UniVerse set **$OPTIONS SELECT.ANY** This requires that the select list specified in the TO clause must be a numbered select list; **SELECT** behaves like **SELECTN**. These emulations return an error when you specify a non-numeric value for the TO clause.

D3, IN2, MVBase, R83, POWER95, and Reality set **$OPTIONS PICK.SELECT**. This causes **SELECT** to use two distinct default select lists, one internal and one external. The default external select list is 0, and the default internal select list is 10.

UniData sets **$OPTIONS FSELECT** by default; for Caché and all other emulations FSELECT is inactive by default. This causes **SELECT** to set both @SELECTED and SYSTEM(11) when using Select List 0. For any other select list, only @SELECTED is set. All other emulations only set @SELECTED.

D3, IN2, jBASE, MVBase, PICK, R83, POWER95, and Ultimate set **$OPTIONS NO.RESELECT**. This prevents the reselecting of a select list; a second **SELECT** is ignored when referencing an active unused or partially used select list. For D3, jBASE, MVBase, R83, POWER95, and Ultimate, **$OPTIONS ARRAY.RESELECT** is also set by default, overriding NO.RESELECT for a dynamic array.

Examples

The following example illustrates the use **SELECT dynarray**. **SELECT** copies all of the field mark identifiers into Select List 3. The @SELECTED system variable contains the number of elements selected (in this case, 4). Each iteration of **READNEXT** reads the next field mark identifier from Select List 3 into the *area* variable:

```
regions="Northeast":@FM:"Southeast":@FM:"Northwest":@FM:"Southwest"
SELECT regions TO 3 ON ERROR PRINT "Select failed"
  PRINT @SELECTED
FOR x=1 TO 5
  READNEXT area FROM 3
  PRINT area
NEXT
```

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"
  PRINT @SELECTED
FOR x=1 TO 5
  READNEXT area FROM rfields
  PRINT area
NEXT
```

See Also

- OPEN statement
- OPENINDEX statement
- READNEXT statement
- READNEXT KEY statement
- SELECT ATKEY statement
- SELECTE statement
- SELECTINDEX statement
- CLEARSELECT statement
- FORMLIST statement
- @SELECTED system variable
- Dynamic Arrays

SELECT ATKEY

Selects a specified key type into a select list.

```
SELECT ivar [TO varname] ATKEY keytype[,recID[,vmcount]] [ON ERROR statements]
```

Arguments

ivar	A local variable used as the index identifier of an open MultiValue file. This variable is set by the **OPENINDEX** statement.
TO *varname*	*Optional* — Either a named select list for an index, specified as a variable name, or a select list number. If omitted, select list 0 is used.
keytype	The specified index key to select. You can specify the empty string (`ATKEY ''`) if you wish MVBasic to ignore the ATKEY clause and take the *recID* and/or *vmcount* values.
recID	*Optional* — A specified record within the index key value at which to start processing.
vmcount	*Optional* — If the record specified in *recID* is a dynamic array, *vmcount* specifies which element to start from. Specified as an integer value, beginning with 1 (the default).

Description

The **SELECT ATKEY** statement selects the index identifiers of the specified type from an index file and places them in a select list. You can then use **READNEXT KEY** to read this select list, one index identifier at a time. Selecting to a select list overwrites any previous values for that select list.

There are three ways to select an index

- **SELECT** selects the entire index into a named select list. You then use **READNEXT KEY** to read individual index items.
- **SELECT ATKEY** selects the specified index key into a named select list. You then use **READNEXT KEY** to read individual index items.
- **SELECTINDEX** selects all the unique index keys into a named select list. You then use **READNEXT** to read individual index items.

For all three types of index **SELECT**, you must specify an index file opened using the **OPENINDEX** statement.

The optional ON ERROR clause specifies one or more MVBasic statements to execute if the **SELECT ATKEY** operation fails. For example, if you specify an invalid *ivar* the ON ERROR statements are executed. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

When you are finished using an assigned select list, you can use the **CLEARSELECT** statement to reset the select list.

Examples

The following example used **OPENINDEX** to open an index to VOC on the attribute F1. The **SELECT ATKEY** selects S-type keys from this index to a select list. The **READNEXT KEY** reads an item from the select list:

```
OPENINDEX 'VOC','F1' TO IdxFp ELSE ABORT
SELECT IdxFp TO IdxList ATKEY "S"
READNEXT KEY Idx,Id FROM IdxList
```

See Also

- OPENINDEX statement
- SELECT statement
- SELECTINDEX statement
- READNEXT KEY statement
- CLEARSELECT statement

SELECTE

Copies select list 0 to a named select list.

```
SELECTE TO varname
```

Arguments

varname	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** or **SELECTN** 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
- SSELECT statement
- Dynamic Arrays

SELECTINDEX

Selects an index.

```
SELECTINDEX indexname[,akey] FROM filevar TO varname
```

Arguments

indexname	The name of a defined index, specified as a quoted string.
akey	*Optional* — A specific index key value, specified as a quoted string.
filevar	A local variable name assigned to the index file by the **OPENINDEX** statement.
varname	A named select list for an index, created by **SELECTINDEX**. *varname* is specified as a variable name.

Description

The **SELECTINDEX** statement is used to select the unique keys of an index for use by the **READNEXT** statement. The index must already have been opened using the **OPENINDEX** statement.

There are three ways to select an index

- **SELECT** selects the entire index into a named select list. You then use **READNEXT KEY** to read individual index items.
- **SELECT ATKEY** selects the specified index key into a named select list. You then use **READNEXT KEY** to read individual index items.
- **SELECTINDEX** selects all the unique index keys into a named select list. You then use **READNEXT** to read individual index items.

Example

```
OPENINDEX 'VOC','F1' TO idxfp
SELECTINDEX 'F1' FROM idxfp TO idxlist
FOR x=1 TO 5
  READNEXT id FROM idxlist
  PRINT id
NEXT
```

See Also

- OPENINDEX statement
- SELECT statement
- SELECT ATKEY statement
- READNEXT statement
- CLEARSELECT statement

SETREM

Positions the remove pointer in a dynamic array.

```
SETREM position ON dynarray
```

Arguments

position	A positive integer specifying the number of bytes to increment the pointer in the dynamic array.
dynarray	A dynamic array in which the pointer is positioned.

Description

The **SETREM** statement positions a pointer within a dynamic array. This statement is commonly used to position a pointer for the **REMOVE** or **REVREMOVE** statements to extract data values from the dynamic array.

You can use the **GETREM** function to return the character position in *dynarray* of the **SETREM** pointer. **REMOVE** and **REVREMOVE** also modify this internal pointer so that repeated calls return successive element values.

See Also

- REMOVE statement
- REVREMOVE statement
- EXTRACT function
- GETREM function
- REMOVE function

SLEEP

Suspends processing for a specified duration.

```
SLEEP [seconds]
SLEEP time
```

Arguments

seconds	*Optional* — An integer count of seconds. If omitted, execution is suspended for 1 second.
time	A wakeup time, specified in 24-hour format as hh:mm[:ss], or in 12-hour format as hh:mm[:ss]AM or hh:mm[:ss]PM.

Description

The **SLEEP** 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 **SLEEP** with no argument, it suspends program execution for one second. You can specify *seconds* as an integer or a fraction. If *seconds* is a decimal number, it is rounded to the nearest whole second.

You can specify *time* in either 24-hour or 12-hour format. A 24-hour time is specified as hh:mm[:ss]. A 12-hour time is specified as hh:mm[:ss]{AM | PM}. In both formats, spaces are not permitted, leading zeros may be omitted, and the seconds component of the time is optional. The following are all valid 24-hour format *time* values: 02:34, 2:34:00, 14:34, 14:34:00. The following are all valid 12-hour format *time* values: 2:34PM, 02:34PM, 2:34:00PM. 2:34AM. Midnight can be represented by 24:00, 00:00, 12:00PM, 00:00PM, or 00:00AM. An invalid *time* argument generates a syntax error.

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
- SLEEP command in *Caché MultiValue Commands Reference*

SSELECT, SSELECTN, SSELECTV

Selects and sorts items into a select list.

```
SSELECT dynarray [TO listnum] [ON ERROR statements]
SSELECT [filevar] [TO listnum] [ON ERROR statements]
SSELECT dynarray TO listname [ON ERROR statements]
SSELECT [filevar] TO listname [ON ERROR statements]

SSELECTN dynarray [TO listnum] [ON ERROR statements]
SSELECTN [filevar] [TO listnum] [ON ERROR statements]

SSELECTV dynarray TO listname [ON ERROR statements]
SSELECTV [filevar] TO listname [ON ERROR statements]
```

Description

The **SSELECT** statements sort the contents of a select list or file into string collation order and place the results in a select list. The **SSELECT** statements sort the contents of a file system directory into string collation order and place the results in a select list. The **SSELECT** statements copy the elements of a dynamic array into a select list in the order listed; they do not sort dynamic array elements. To sort dynamic array elements, use **SELECT** to copy the elements into a select list, then use **SSELECT** on that select list.

When sorting the contents of a select list, **SSELECT** removes duplicate values. Therefore, an output select list may contain fewer items than the input select list.

The output select list can be a numbered select list or a named select list. The “N” and “V” command name suffixes specify whether the output select list is a numbered select list or a named select list.

The **SSELECT** statements sort in ordinary string collation order. The **SELECT** statements sort in Caché storage order: first the empty string, then canonical numbers in ascending numeric order, then strings in string collation order. The **SSELECT** statements are otherwise comparable to the corresponding **SELECT** statements.

The optional ON ERROR clause specifies one or more MVBasic statements to execute if the **SSELECT** operation fails. For example, if you specify an invalid *listnum* the ON ERROR statements are executed. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

See Also

- SELECT statement
- Dynamic Arrays

STATUS

Provides file status information.

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

Arguments

dynarray	A dynamic array used by **STATUS** to hold file information as Field elements.
filevar	A file variable name specifying the file from which status information is to be returned. This *filevar* is obtained from **OPEN** or **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 **OPEN** or **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 *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

To display individual status fields, use angle bracket syntax. The following example displays the 20th field, which is the full file path:

```
STATUS statdyn FROM filevar
PRINT statdyn<20>
```

Field 1 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.

Field 20 of *dynarray* contains the full file path of the open file.

Field 21 of *dynarray* contains a numeric code for the file type, as follows: -2 sequential file; -1 dir-type file; 0 global with the full record in a single node; 1 global with each attribute in a separate subnode.

Examples

The following example opens a sequential file on a Windows system and determines its status. It prints out two status fields: the full pathname and the file type (in this case, -2):

```
myfile='c:\InterSystems\Cache\Dev\mv\samples\CommandExample'
OPENSEQ myfile TO filevar ELSE STOP 201,myfile
STATUS statdyn FROM filevar
   PRINT statdyn<20>
   PRINT statdyn<21>
CLOSESEQ filevar
```

The following example opens the VOC file and determines its status. It prints out two status fields: the file pathname (in this case, a global variable) and the file type (in this case, 0):

```
OPEN 'VOC' TO myvoc ELSE STOP 201,'VOC'
STATUS statdyn FROM myvoc
   PRINT statdyn<20>
   PRINT statdyn<21>
CLOSE myvoc
```

See Also

- OPEN statement
- OPENSEQ statement
- READSEQ statement
- SEEK statement
- WRITESEQ statement

STOP, STOPE, STOPM

Terminates program execution and returns to the calling environment.

```
STOP [errcode [,val1[,val2]]]
STOPE [errcode [,val1[,val2]]]
STOPM [message]
```

Arguments

errcode	*Optional* — A MultiValue error code; commonly (but not always) specified as a positive integer. The error code can be specified as a literal or as a expression that resolves to a literal value. A non-numeric literal value must be specified as a quoted string.
val	*Optional* — A comma-separated list of one or more literal values to insert into the error message corresponding to *errcode*. These insert values can be specified as literals or as expressions that resolves to a literal value. A non-numeric literal value must be specified as a quoted string.
message	*Optional* — An expression that resolves to a literal error message text, specified as a quoted string.

Description

All forms of the **STOP** statement are used to terminate program execution and return control to the calling environment. If you specify an argument, these statements return an error message before terminating program execution.

STOP and **STOPE** return MultiValue error messages. They are nearly functionally identical; both return the specified error code and corresponding error message. **STOPE** always returns both the error code and the error message. This includes error messages missing *val* insert values. **STOP** always returns the error code; it only returns the error message if you have specified at least one of the *val* insert values required to complete the error message, or if the error message does not require any insert values. For a list of error codes and corresponding error messages, see Error Messages in the *Caché MultiValue Commands Reference*.

STOPM returns the literal message text specified in *message*.

When you call an MVBasic routine from a non-MultiValue environment, a **STOP** statement clears the entire execution stack and either terminates the process or returns to the Terminal prompt.

Examples

The following Windows example shows a common use of **STOP** as an ELSE clause statement:

```
foo="c:\foofile"
OPEN foo TO myfile ELSE STOP 201,foo
```

STOP returns the error message: `[201] Unable to open file 'c:\foofile'.`

The following examples show the difference between **STOPE** and **STOP** when the error message requires an insert value that the command does not provide:

```
OPEN foo TO myfile ELSE STOPE 201
```

STOPE returns: `[201] Unable to open file ''.`

```
OPEN foo TO myfile ELSE STOP 201
```

STOP returns: `[201]`

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.

During debugging, **STOP** terminates the debugging session. The debugger treats an **ABORT** as an error condition; the debugger performs a break operation to allow for examination of the condition causing the **ABORT**.

See Also

- ABORT statement
- ERRMSG statement
- BREAK statement
- ObjectScript: QUIT command

SUBROUTINE

Defines an external subroutine.

```
SUBROUTINE [name][(arglist)]
   [statements]
    RETURN
```

Arguments

name	*Optional* — Any valid name to assign to the subroutine.
arglist	*Optional* — List of variables specifying arguments that are passed to the **SUBROUTINE** when it is called. Multiple arguments are separated by commas. The *arglist* is enclosed with parentheses.
statements	A group of statements to be executed within the body of the **SUBROUTINE**.

Description

The **SUBROUTINE** statement defines an external subroutine. A **SUBROUTINE** is a separate procedure that can take arguments, perform a series of statements, and change the values of its arguments.

The **SUBROUTINE** statement is very similar to FUNCTION, except that **FUNCTION** always returns a value. A **SUBROUTINE** generally does not return a value. (You can use the **SUBR** function to call an external subroutine that returns a value.)

There cannot be a label on the **SUBROUTINE** statement line. There can only be one **SUBROUTINE** statement in an external subroutine (no nested subroutines). The **SUBROUTINE** statement must be the first line in the external subroutine, with the following exceptions: comment lines, **$OPTIONS** statements, **$COPYRIGHT** statements, and **DIM** statements that do not dimension a static array. For example, `DIM Var()` and `DIM abc` are permitted, but `DIM Var(2)` is not.

The *name* argument allows you to identify the external subroutine; it is not (strictly speaking) required to define or invoke an external subroutine. If *name* is omitted, either of the following syntactic forms are permitted: `SUBROUTINE (arglist)` or `SUBROUTINE(arglist)`.

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 *arglist*, 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
- FUNCTION statement
- CALL statement
- GOSUB statement
- SUBR function

SWAP

Replaces all instances of a substring in a variable.

```
SWAP oldstring WITH newstring IN variable
```

Arguments

oldstring	The substring to be replaced. An expression that resolves to a valid string or numeric.
newstring	The replacement substring. An expression that resolves to a valid string or numeric. To delete *oldstring*, specify the empty string ("").
variable	An existing variable containing a string value. *variable* may be a dynamic array. *variable* accepts a single dynamic array reference (A<i>), a single substring reference (A[s,l]), or a substring reference nested inside a dynamic array reference (A<i>[s,l]).

Description

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

The values of *oldstring* and *newstring* 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 string replacement.

To remove all instances of *oldstring* from *variable*, specify the null string ("") as the *newstring* value. The null string ("") cannot be used as the *oldstring* 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** statement for substring replacement.

SWAP and **CHANGE** both perform string substitution, and are functionally identical. **CONVERT** performs character-for-character substitution.

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:"Philadephia Penn."
SWAP "Penn." WITH "PA" IN cities
```

See Also

- CHANGE function
- CONVERT statement
- CONVERT function
- CHANGE function
- Strings

TCLREAD

Copies the terminal command line into a variable.

```
TCLREAD variable
```

Arguments

variable	A variable used to hold the command line.

Description

The **TCLREAD** statement copies the Terminal Control Language (TCL) command line into *variable*. This allows parameters to be passed from TCL to the MVBasic program.

THROW

Throws an exception from a TRY block to a CATCH exception handler.

```
THROW [oref]
```

Arguments

oref	*Optional* — A user-defined object reference.

Description

The **THROW** statement explicitly issues an exception from within a block of code defined by a **TRY** statement. Issuing a **THROW** transfers execution from the **TRY** block to the corresponding **CATCH** exception handler.

THROW is used to issue an explicit exception. MVBasic issues an implicit exception when a runtime exception occurs. A runtime exception generates an exception object which it throws to a **CATCH** exception handler.

THROW has two forms:

- Without an argument
- With an argument

THROW without an Argument

Argumentless **THROW** transfers exception processing to the corresponding **CATCH** exception handler. No object is pushed on the stack, but the %New() method is called.

THROW with an Argument

THROW *oref* specifies a user-defined object reference, which it throws to the **CATCH** statement.

Arguments

expression

A user-defined object reference (oref). For example, `THROW "Sample.MyException"->%New("Example Error",45)`. The creation and population of this exception object is the responsibility of the programmer.

Examples

The following example shows the use of **THROW**:

```
TRY
  PRINT "about to issue a THROW statement"
  THROW "Sample.MyException"->%New("Example Error",45,"Sample Program")
  PRINT "this should not display"
CATCH myvar
  PRINT "this is the exception handler"
  PRINT :myvar->Name,"Error Name"
  PRINT :myvar->Code,"Error Code Number"
  PRINT :myvar->Location,"Error Location"
END TRY
PRINT "this is where the code falls through"
```

See Also

- CATCH statement
- TRY statement

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.

Locks and Transactions

File locks and record locks that were taken out during a transaction are released at the end of a transaction. If there are nested transactions, the release of locks taken out during the inner transactions is delayed until the completion of the outermost transaction. This release of locks is part of a successful **TRANSACTION COMMIT** or **TRANSACTION ABORT** operation. Locks are described in the LOCK statement.

See Also

- TRANSACTION START statement
- TRANSACTION COMMIT statement

TRANSACTION COMMIT

Commits all changes made during the current transaction.

```
TRANSACTION COMMIT {THEN statements  | ELSE statements }
```

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. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

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.

Locks and Transactions

File locks and record locks that were taken out during a transaction are released at the end of a transaction. If there are nested transactions, the release of locks taken out during the inner transactions is delayed until the completion of the outermost transaction. This release of locks is part of a successful **TRANSACTION COMMIT** or **TRANSACTION ABORT** operation. Locks are described in the LOCK statement.

See Also

- TRANSACTION START statement
- TRANSACTION ABORT statement

TRANSACTION START

Begins a transaction.

```
TRANSACTION START {THEN statements | ELSE statements}
```

Description

The **TRANSACTION START** statement initiates a transaction. There is no command to demarcate the end of 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. If neither a **TRANSACTION COMMIT** nor a **TRANSACTION ABORT** is issued, the transaction remains open until the end of the program, at which time it is automatically rolled back.

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. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

You can use multiple **TRANSACTION START** statements to create nested transactions.

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.

Locks and Transactions

File locks and record locks that were taken out during a transaction are released at the end of a transaction. If there are nested transactions, the release of locks taken out during the inner transactions is delayed until the completion of the outermost transaction. This release of locks is part of a successful **TRANSACTION COMMIT** or **TRANSACTION ABORT** operation. Locks are described in the LOCK statement.

See Also

- TRANSACTION COMMIT statement
- TRANSACTION ABORT statement

TRY

Identifies a block of code to monitor for exceptions during execution.

```
TRY
   statements
CATCH exceptionvar
   statements
END TRY
```

Description

The **TRY** statement takes no arguments. It is used to identify one or more Caché MVBasic code statements between the **TRY** keyword and the **CATCH** keyword. This block of code is protected code for structured exception handling. If an exception occurs within this block of code, Caché sets *exceptionvar* to an object describing the exception, then transfers execution to an exception handler, identified by the **CATCH** statement. This is known as throwing an exception. If no exception occurs, execution continues with the next Caché MVBasic statement after the **END TRY** statement.

An exception may occur as a result of a runtime exception, such as attempting to divide by 0, or it may be explicitly propagated by issuing a **THROW** statement.

A **TRY** block must be immediately followed by a **CATCH** block. The paired **TRY** and **CATCH** are terminated by an **END TRY** statement.

Examples

In the following examples, the **TRY** code block is executed. It attempts to set the local variable *a*. In the first example, the code completes successfully, and the **CATCH** statements are skipped over. In the second example, the code fails an Err error indicating division by zero, and execution is passed to the **CATCH** statement.

TRY succeeds:

```
TRY
  PRINT "about to divide by one"
  a=7/1
  PRINT "this line is executed"
CATCH myvar
  PRINT "this is the exception handler"
  PRINT "Error name: ",myvar->Name
END TRY
PRINT "this is where the code falls through"
```

TRY fails:

```
TRY
  PRINT "about to divide by zero"
  a=7/0
  PRINT "this should not display"
CATCH myvar
  PRINT "this is the exception handler"
  PRINT "Error name: ",myvar->Name
END TRY
PRINT "this is where the code falls through"
```

See Also

- CATCH statement
- THROW statement

UNLOCK

Releases a process lock.

```
UNLOCK expression
```

Arguments

expression	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. **UNLOCK ""** is equivalent to **UNLOCK 0**.

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

filevar	A file variable name used to refer to the file in Caché MVBasic. This *filevar* 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 executed if the end-of-file write fails. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

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 {ON | TO} filevar,recID
   [SETTING var] [ON ERROR statements] [LOCKED statements] [[THEN statements] [ELSE
 statements]]

WRITEU data {ON | TO} filevar,recID
   [SETTING var] [ON ERROR statements] [LOCKED statements] [[THEN statements] [ELSE
 statements]]

WRITEV data {ON | TO} filevar,recID,fieldno
   [SETTING var] [ON ERROR statements] [LOCKED statements] [[THEN statements] [ELSE
 statements]]

WRITEVU data {ON | TO} filevar,recID,fieldno
   [SETTING var] [ON ERROR statements] [LOCKED statements] [[THEN statements] [ELSE
 statements]]
```

Arguments

data	Data to write to the MultiValue file. Can be an expression or variable that resolves to a dynamic array or some other literal value.
filevar	A local variable used as the file identifier of an open MultiValue file. This variable is set by the **OPEN** statement. You can specify either ON or TO as the keyword.
recID	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 *recID* are case-sensitive. Additional naming conventions are described below.
fieldno	The field number of the field to write. Used with **WRITEV** and **WRITEVU**.
SETTING *var*	*Optional* — When an error occurs, sets the local variable *var* to the operating system's error return code. Successful completion returns 0; error return codes are platform-specific. The SETTING clause is executed before the ON ERROR, THEN, or ELSE clause. Provided for jBASE compatibility.

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, then releases the update record lock
- WRITEU writes a record, retaining the update record lock
- WRITEV writes a field within a record, then releases the update record lock
- WRITEVU writes a field within a record, retaining the update record lock

You can optionally specify a LOCKED clause. This clause is executed if the write operation could not acquire an update record lock due to lock contention. The LOCKED clause is optional, but strongly recommended; if no LOCKED clause is specified, program execution waits indefinitely for the conflicting lock to be released. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the LOCKED keyword and the first line.

You can optionally specify an ON ERROR clause, which is executed when the operation fails and generates an error code. For example, attempting to write to a read-only file. If you do not specify an ON ERROR clause, the ELSE clause is taken for an error code condition, as well as for an unsuccessful write. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

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 does not complete successfully, the ELSE clause is executed. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

If the **WRITE** has neither an ON ERROR clause nor an ELSE clause, a failed write operation generates a <WRITE> error and halts program execution.

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

The following are naming conventions for a valid MultiValue *recID*:

- 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.

Record Locks

RECORDLOCKU performs an update (exclusive) lock on a record. This update record lock is automatically released when you write data to the record using **WRITE** or **WRITEV**. The **WRITEU** and **WRITEVU** commands do not release the update record lock. You can check the status of an update record lock using the RECORDLOCKED function. You can explicitly release an update record lock using the RELEASE command.

Writing a Field to a Record

WRITEV and **WRITEVU** writes a field within a record. They search the record string for the delimited piece specified by the *fieldno* count, replace it, then rewrite the record. If the *fieldno* is higher than the number of field delimiters, these statements append the field to the end of the record. If the entire record consists of a single numeric value (and thus contains

no field delimiters), these statements convert the record value to a string before appending the specified field value. If the *fieldno* is 0, a new field is appended to the beginning of the record.

WRITE and MATWRITE

The various **WRITE** statements write a dynamic array (or a string value) to a MultiValue file record. The various **MATWRITE** statements write a dimensioned array to a MultiValue file record.

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
- MATWRITE statement
- STATUS function
- 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

data	Data to write to the sequential file. Can be an expression or variable that resolves to a literal value.
filevar	A file variable name used to refer to the file in Caché MVBasic. This *filevar* 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. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

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

WRITELIST

Saves a select list.

```
WRITELIST dynarray ON listname [SETTING var]
WRITELIST dynarray TO listname [SETTING var]
```

Arguments

dynarray	A select list supplied by the **SELECT** statement. A dynamic array of elements separated by field mark delimiters.
listname	A name assigned to the saved select list.
SETTING *var*	*Optional* — a local variable used to receive a numeric error code if the operation fails.

Description

The **WRITELIST** statement saves a select list. Once you have saved a select list, you can use **GETLIST** to activate the saved select list so that it can be read by **READNEXT**.

The *listname* select list is saved in the &SAVEDLISTS& file. Caché stores this file using the ^SAVEDLISTS global.

You can use either the ON or TO keyword. The ON keyword is preferred; the TO keyword is provided for jBASE compatibility.

The optional SETTING clause is executed if the **WRITELIST** operation fails and an error code is generated. The *var* variable is set to this numeric error code. The SETTING clause is provided for jBASE compatibility.

Emulation

In jBASE emulation, if *listname* is 0 or "", **WRITELIST** creates a new select list 0, saves to select list 0, and makes this select list active. Caché and other emulations create select list 0 and save to select list 0, but do not retain it as an active select list.

See Also

- DELETELIST statement
- SELECT statement

WRITESEQ, WRITESEQF

Writes a line of data to a sequential file.

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

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

Arguments

data	Data to write to the sequential file. Can be an expression or variable that resolves to a literal value.
filevar	A file variable name used to refer to the file in Caché MVBasic. This *filevar* 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 an ON ERROR clause, which is executed when the operation fails and generates an error code. For example, specifying an invalid *filevar*, or attempting to write to a read-only file. If you do not specify an ON ERROR clause, the ELSE clause is taken for an error code condition, as well as for an unsuccessful write. The *statements* argument can be the NULL placeholder keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line; there must be a line break between the ON ERROR keyword and the first line.

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 does not complete successfully, the ELSE clause is executed. The *statements* argument can be the NULL keyword, a single statement, or a block of statements terminated by the END keyword. A block of statements has specific line break requirements: each statement must be on its own line and cannot follow a THEN, ELSE, or END keyword on that line.

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).

I/O Buffering

By default, **WRITESEQ** operations are written to an I/O buffer. This buffer is automatically assigned as part of the **OPENSEQ** operation. I/O buffering significantly improves overall performance, but means that write operations are not immediately applied to the sequential file.

WRITESEQF is identical to **WRITESEQ**, except that it does not use I/O buffering. **WRITESEQF** is useful for logging operations which must be immediately written to disk. However, because writing directly to a sequential file can significantly effect performance, **WRITESEQF** is not recommended for most data update operations.

Caché MVBasic provides two statements that override **WRITESEQ** I/O buffering. The **FLUSH** statement immediately writes the current contents of the I/O buffer to the sequential file. The **NOBUF** statement disables the I/O buffer for the duration of the sequential file open. That is, all subsequent **WRITESEQ** operations are immediately executed on the sequential file, exactly as if they were **WRITESEQF** operations.

New Sequential File

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
- FLUSH statement
- NOBUF statement
- SEEK statement
- STATUS statement
- STATUS function

$XECUTE

Executes an ObjectScript command.

```
$XECUTE expression
```

Arguments

expression	An expression that evaluates to one or more valid ObjectScript commands, specified as a quoted string. The *expression* string delimiter character cannot be used within *expression*. For example, if the *expression* string contains double quotation marks (delimiting an ObjectScript string literal), you must enclose *expression* with either single quote marks (') or backslash (\) characters.

Description

$XECUTE is used to invoke an ObjectScript command from within Caché MVBasic. **$XECUTE** executes ObjectScript commands that result from the process of expression evaluation of the specified argument. Each **$XECUTE** argument must evaluate to a string containing ObjectScript commands. The string must not contain a tab character at the beginning or a <Return> at the end. The string must be no longer than a valid ObjectScript program line.

In effect, the **$XECUTE** argument is like calling a one-line subroutine. It is terminated when the end of the argument is reached or an ObjectScript **QUIT** command is encountered. After Caché executes the argument, it returns control to the point immediately after the **$XECUTE** argument.

Each invocation of **$XECUTE** places a new context frame on the call stack for your process. The ObjectScript **$STACK** special variable contains the current number of context frames on the call stack.

Local Variables

Variables in MVBasic are local, private variables. They are hidden from the ObjectScript code being executed by a **$XECUTE** statement. Therefore, **$XECUTE** can only be used for ObjectScript code that does not access MVBasic variables or expressions containing MVBasic variables.

If you wish to execute ObjectScript that uses MVBasic variables, your MVBasic code must pass those variables as actual parameters to an external ObjectScript routine.

Invoking Other Command Shells

You can use the **EXECUTE**, **PERFORM**, and **CHAIN** commands to issue MultiValue commands from within Caché MVBasic.

You can use the **PCPERFORM** command to issue an operating system command from within Caché MVBasic.

Examples

The following example executes the subroutine that is the value of CosSub.

```
CosSub="WRITE ! FOR I=1:1:5 { WRITE ?I*5,I+1 }"
$XECUTE CosSub
```

Returns:

```
2 3 4 5 6
```

Notes

$XECUTE and Objects

You can use **$XECUTE** to call object methods and properties and execute the returned value, as shown in the following examples:

```
$XECUTE patient.Name
$XECUTE "WRITE patient.Name"
```

$XECUTE and FOR

If the **$XECUTE** argument contains an ObjectScript **FOR** command, the scope of the **FOR** is the remainder of the argument. When the outermost **FOR** in an **$XECUTE** argument is terminated, the **$XECUTE** argument is also terminated.

$XECUTE and DO

If the **$XECUTE** argument contains an ObjectScript **DO** command, Caché executes the routine or routines specified in the **DO** argument or arguments. When it encounters a **QUIT**, it returns control to the point immediately following the **DO** argument.

For example, in the following commands, Caché executes the routine ROUT and returns to the point immediately following the **DO** argument to write the string “DONE”.

```
$XECUTE 'DO ^ROUT WRITE !,"DONE"'
```

$XECUTE and GOTO

A ObjectScript command specified in **$XECUTE** cannot specify an ObjectScript label. An ObjectScript command specified in **$XECUTE** cannot access an MVBasic label. Therefore the use of **GOTO** within **$XECUTE** is not supported.

$XECUTE and QUIT

There is an implied **QUIT** at the end of each **$XECUTE** argument.

Nested Invocation of $XECUTE

Caché supports the use of the ObjectScript **XECUTE** command within the **$XECUTE** argument. However, you should use nested invocation of **$XECUTE** with caution because it can be difficult to determine the exact flow of processing at execution time.

Execution Time for Commands Called by $XECUTE

The execution time for code called within **$XECUTE** can be slower than the execution time for the same code encountered in the body of a routine. This is because Caché compiles source code that is specified with the **$XECUTE** command or that is contained in a referenced global variable each time it processes the **$XECUTE**.

See Also

- CHAIN statement
- EXECUTE statement
- PERFORM statement
- PCPERFORM statement
- ObjectScript XECUTE command

Caché MultiValue Basic Functions

@ (at sign)

Sets screen cursor position or screen display option.

```
@(x[,y])
@(code[,arg])
```

Arguments

x	An expression that resolves to a positive integer specifying the number of columns to indent the horizontal position of the screen cursor. 0=column 1 (no indent), 1=indent 1 column.
y	*Optional* — An expression that resolves to a positive integer specifying the vertical line position of the screen cursor. 0=top of screen. If omitted, defaults to the current line.
code	An expression that resolves to a negative integer specifying a screen display option code.
arg	*Optional* — An expression that resolves to an integer argument required by certain *code* values.

Description

The **@** function has two forms. If the first argument is a positive integer or zero, it sets the cursor position. If the first argument is a negative integer, it sets a screen display option.

Cursor Positioning

The **@** function (with a positive first argument) changes the horizontal and/or vertical position of the screen cursor. To change only the horizontal position, specify `@(x)`. To change only the vertical position, specify `@(0,y)`.

The **@** function *does not change* the ObjectScript $X and $Y special variables.

Screen Display Options

The **@** function (with a negative first argument) changes a screen display option. The following *code* options are supported:

-1	Clear screen and position cursor at home location (top left). For wyse terminals, -1 clears the screen, except for protected fields.
-2	Position cursor at home location (top left).
-3	Clear the screen from the current cursor position to the end of the screen.
-4	Clear the screen from the current cursor position to the end of the line.
-5	Start blinking text.
-6	Stop blinking text.
-7	Start protected field. (See -62)
-8	End protected field. (See -62)
-9	Back space. You can supply an optional 2nd argument specifying the number of backspaces to perform. The default is 1 backspace.

-10	Back line. Moves up a line without resetting cursor column. You can supply an optional 2nd argument specifying the number of lines to go back. The default is to go back one line (go to the previous line).
-13	Start reverse video. This displays white characters on a black background.
-14	Stop reverse video.
-15	Start underlining.
-16	Stop underlining.
-17	Insert line. Moves up a line and resets cursor to column 1. You can supply an optional 2nd argument specifying the number of lines to insert.
-18	Delete line. Resets cursor to column 1.
-20	Set insert (overtype) mode.
-21	Reset insert mode to normal mode (the default).
-23	Disable keyboard input and screen display.
-24	Reenable keyboard input and screen display (the default).
-29	Use 80–column line width (the default).
-30	Use 132–column line width.
-31	Turn off blinking cursor position indicator.
-32	Turn on blinking cursor position indicator (the default).
-34	Move cursor forward (insert blank space). You can supply an optional 2nd argument specifying the number of spaces to advance the cursor. 0 or 1 both advance the cursor 1 space. The default is to advance the cursor 1 space.
-37	Set foreground (text) color. You can supply an optional 2nd argument specifying the color, as follows: 0=black, 1=red, 2=green, 3=yellow, 4=blue, 5=magenta, 6=turquoise, 7=white, 8=no change, 9=black. Higher numbers have no effect. The default is red. The blinking cursor remains black.
-38	Set background color. You can supply an optional 2nd argument specifying the color, as follows: 0=black, 1=red, 2=green, 3=yellow, 4=blue, 5=magenta, 6=turquoise, 7=white, 8=no change, 9=white. Higher numbers have no effect. The default is red. The blinking cursor changes to a contrast color to the background color.
-42	Disable keyboard input.
-43	Reenable keyboard input (the default).
-50	Move cursor up (reverse line feed). You can supply an optional 2nd argument specifying the number of lines to move up. The default is 1.
-56	Enables arrow keys / numeric keypad (the default). Arrow keys can be used to move the cursor left and right within the command line, or to retrieve previous command lines.
-57	Disables arrow keys / numeric keypad. All arrow keys are equivalent to the Enter key.
-58	Start bold text.
-59	Stop bold text.
-62	Enable protected fields (see -7 and -8).

-63	Disable protected fields (see -7 and -8).
-108	Sounds the bell.

Emulation

In D3 emulation, @(-11) enables protected fields and @(-12) disables protected fields. D3 also supports @(-57) through @(-64).

In jBASE and Reality emulation, @(-128) through @(-191) are supported. Start blinking text with -131, -138, -139, -142, or -143. Start reverse text with -140 or -141. Start underline text with -144, -145, -152, or -153. Start bold text with -160 or -161. Start blink/underline with -146, -147, -150, -151, -154, -155, -158, or -159. Start reverse/underline with -148, -149, -156, or -157. Start blink/bold with -162, -163, or -166. You can use -137 to turn off any combination of bold, blinking, reverse, or underline text.

In MVBase emulation, @(-57) through @(-64) are used for dimmed foreground colors.

In Ultimate emulation, @(-1) Clear screen; @(-2) Cursor home; @(-3) Clear to end of screen; @(-4) Clear to end of line; @(-5) Blink on; @(-6) Blink off; @(-7) Protected field on; @(-8) Protected field off; @(-9) Cursor left; @(-10) Cursor up; @(-11) Cursor down; @(-12) Cursor right; @(-13) Printer on; @(-14) Printer off; @(-15) Printer on (enable slave port in transparent mode); @(-16) Printer on (initiate slave local print); @(-17) Underline on; @(-18) Underline off; @(-19) Reverse on; @(-20) Reverse off; @(-21) Delete line; @(-22) Insert line; @(-23) Scroll up; @(-24) Bold on; @(-25) Bold off; @(-26) Delete character; @(-27) Insert character; @(-28) Insert on; @(-29) Insert off; @(-33) 80 column screen; @(-34) 132 column screen; @(-50) Graphics on; @(-51) Graphics off; @(-52) Blink on; @(-53) Blink off; @(-54) Reverse on; @(-55) Reverse off; @(-58) Underline on; @(-59) Underline off; @(-66) Dim (half intensity) on; @(-67) Dim (half intensity) off; @(-80) Set 80 column mode; @(-82) Set 132 column mode; @(-108) sounds the bell.

In UniData emulation, @(-19) sounds the bell.

ABS

Returns the absolute value of a number.

```
ABS(number)
```

Arguments

number	An expression that resolves to 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** returns a number in canonical form; it 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 the empty string ("") 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

dynarray	An expression that resolves to a dynamic array containing numeric elements.

Description

The **ABSS** function returns a dynamic array containing the absolute value of each numeric element of *dynarray*. The absolute value of a number is its unsigned magnitude. **ABSS** returns numbers in canonical form; it removes signs, and leading and trailing zeros from the element values. If a *dynarray* element is a missing element, an empty 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 11ý-22ý-33ý44
PRINT ABSS(a);    ! returns 11ý22ý33ý44
```

The following example uses the **ABSS** function with a dynamic array that has missing and non-numeric elements:

```
b = -11:@VM:"":@VM:"-7dwarves":@VM:@VM:"dwarves"
PRINT ABSS(b);    ! returns 11ý0ý7ý0ý0
```

See Also

- ABS function
- NEGS function
- Dynamic Arrays

ACCESS

Returns information about the current MultiValue file called from a dictionary.

```
ACCESS(code)
```

Arguments

code	A literal integer value in the range 1 through 11 (inclusive). You cannot specify *code* as an expression. A *code* value containing a fractional portion is truncated to an integer. A code value outside of the range 1 through 11 generates a syntax error.

Description

The **ACCESS** function returns information about the current Item being processed in CMQL when the routine is called from a DICTIONARY CALL conversion code. The called routine is called for every value in the DICT item attributes. If specified in DICT item attribute 7, the routine is also called each time a break-on occurs.

The information returned by **ACCESS** depends on the value of *code*. The following *code* values are supported:

1	The *filevar* for the data portion of the file. A *filevar* is assigned by the **OPEN** statement.
2	The *filevar* for the dictionary portion of the file. A dictionary *filevar* is assigned by the **OPEN** statement using the DICT keyword.
3	A dynamic array containing the current item from the file.
4	Counter of the number of items processed. Defaults to 0.
5	The attribute number specified in attribute 2 of the DICT item that is calling the current routine. Defaults to 0.
6	The current value number being processed. (1 is returned for single valued attributes.) Defaults to 0.
7	The current subvalue number being processed. (1 is returned if there are no subvalues.) Defaults to 0.
8	Number of detail lines processed since the last break. (This *code* is only valid if the DICT item attribute 7 is specified.) Defaults to 0.
9	The current break level. Set to 0 when processing a detail line. (This *code* is only valid if the DICT item attribute 7 is specified.) Defaults to 0.
10	Item ID.
11	File name.

ACCESS is provided for compatibility with the D3 (PICK) implementation of MultiValue Basic.

See Also

- OPEN statement
- STATUS statement
- FILEINFO function
- STATUS function

ACOS

Returns the arc-cosine of an angle.

```
ACOS(number)
```

Arguments

number	An expression that resolves to a number in the range -1 to 1 (inclusive). Values outside of this range generate an <ILLEGAL VALUE> error.

Description

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

By default, Caché MVBasic trig functions return results in degrees. To return results in radians, set **$OPTIONS 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 **ACOS** function to return the arc-cosine of an angle:

```
PRINT ACOS(-0.5):" in degrees"
PRINT ACOS(-0.5)*(3.1415/180)):" in radians"
```

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

dynarray	An expression that resolves to a 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 a null string, missing, or a non-numeric value, **ADDS** parses its value as 0 (zero).

If the two dynamic arrays have different numbers of elements, the returned dynamic array has the number of elements of the longer dynamic array. By default, the shorter dynamic array is padded with 0 value elements for the purpose of the arithmetic operation. You can also use the REUSE function to define behavior when specifying two dynamic arrays with different numbers of elements.

You can use the **NUMS** function to determine if the elements in a dynamic array are numeric. You can use the SUBS (subtraction), MULS (multiplication), DIVS or DIVSZ (division), MODS (modulo division), and PWRS (exponentiation) 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 11ý22ý33ý44
PRINT ADDS(a,b);  ! returns 21ý31ý41ý51
```

See Also

- CATS function
- DIVS function
- DIVSZ function
- MODS function
- MULS function
- PWRS function
- SUM function
- SUMMATION function
- SUBS function
- Dynamic Arrays

ALPHA

Determines if a string is alphabetic or not.

```
ALPHA(string)
```

Arguments

string	An expression that resolves to a string.

Description

If *string* consists entirely of alphabetic characters, **ALPHA** returns 1. Otherwise, **ALPHA** returns 0. Note that blank spaces are non-alphabetic characters. Dynamic array separator characters are considered to be alphabetic characters. On a Unicode system **ALPHA** recognizes Unicode letters as 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

dynarray	An expression that resolves to a dynamic array of boolean values.

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 empty string, a missing element, or a string containing any non-numeric character, **ANDS** parses its value as 0.

A single leading plus or minus sign is parsed as a numeric character. Multiple leading plus and minus signs are treated as numeric characters in a number, but not in a numeric string. A numeric string with multiple leading plus and minus signs causes **ANDS** to treat the element value as non-numeric.

If the two dynamic arrays have different numbers of elements, the returned dynamic array has the number of elements of the longer dynamic array. By default, the shorter dynamic array is padded with 0 value elements for the purpose of the logical comparison. You can also use the REUSE function to define behavior when specifying two dynamic arrays with different numbers of elements.

Caché MVBasic also supports the logical operators & and AND.

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 1ý0ý1ý0
```

The following example performs an AND test on two dynamic arrays of different lengths:

```
a=1:@VM:0:@VM:1:@VM:0
b=1:@VM:1:@VM:1:@VM:1:@VM:1:@VM:0
PRINT ANDS(a,b)
      ! returns 1ý0ý1ý0ý0ý0
```

See Also

- ORS function
- NOTS function
- Dynamic Arrays
- Operators

ASCII

Converts a string from EBCDIC to ASCII.

```
ASCII(string)
```

Arguments

string	An expression that resolves to a string.

Description

The **ASCII** function takes a string of characters and returns the EBCDIC code representation for each character. If you supply a string of EBCDIC code characters, **ASCII** returns the corresponding ASCII character(s). This is the inverse of the **EBCDIC** function. The *string* cannot contain Unicode characters.

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("ABCDEFG")
astring=ASCII(estring)
PRINT astring
  ! returns "ABCDEFG"
```

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

number	An expression that resolves to a number or numeric string in the range -1 to 1 (inclusive). Values outside of this range generate an <ILLEGAL VALUE> error.

Description

The **ASIN** function returns the trigonometric arc-sine of *number*. An arc-sine is the inverse of a sine.

By default, Caché MVBasic trig functions return results in degrees. To return results in radians, set **$OPTIONS 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

var	A user variable. If *var* is not a valid variable name, MVBasic issues a syntax error.

Description

The **ASSIGNED** function determines whether a user 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 empty string (""), or is assigned an unassigned variable.

The input *var* can be a local variable, a global variable, or a process-private global variable. It can be with or without subscripts.

Note: **ASSIGNED** should not be used on system variables (@ variables). It always returns 0 for all @ variables, whether or not the @ variable currently has a value.

The **UNASSIGNED** function is the functional opposite of the **ASSIGNED** function.

The **COMMON** statement initializes variables as unassigned in Caché MVBasic. Array variable initialization varies with different MultiValue emulations.

You can use the **$KILL** statement to unassign variables.

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

- COMMON statement
- $KILL statement
- UNASSIGNED function

ATAN

Returns the arctangent of a number.

```
ATAN(number)
```

Arguments

number	An expression that resolves to 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. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.

By default, Caché MVBasic trig functions return results in degrees. To return results in radians, set **$OPTIONS RADIANS**. 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/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

bitstring	A bit string, specified as an expression that resolves to a positive integer. For example, the integer 64 specifies the bitstring `1000000`. The maximum *bitstring* value is 9223372036854775807.

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**:

	bitstring1 = 0	*bitstring1* = 1
bitstring2 = 0	0	0
bitstring2 = 1	0	1

A *bitstring* can be expressed as either a number or as a string. A number is converted to canonical form, with leading plus signs and leading and trailing zeros omitted. If either argument evaluates to the null string or a non-numeric string 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

bitstring	The bit string, specified as an expression that resolves to a positive integer. For example, the integer 64 specifies the bitstring `1000000`. The maximum *bitstring* value is 9223372036854775807.
bitno	The bit position in *bitstring* to set to its opposite value. An expression that resolves to a positive integer. Bit positions are counted right to left, beginning with position 0. The maximum *bitno* value is 62. A fractional *bitno* is truncated to its integer portion. A negative *bitno* generates a <FUNCTION> error.

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 or a non-numeric string 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

bitstring	A bit string, specified as an expression that resolves to a positive integer. For example, the integer 64 specifies the bitstring `1000000`. The maximum *bitstring* value is 9223372036854775807.

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**:

	bitstring1 = 0	*bitstring1* = 1
bitstring2 = 0	0	1
bitstring2 = 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 or a non-numeric string 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 1111111, 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

bitstring	The bit string, specified as an expression that resolves to a positive integer. For example, the integer 64 specifies the bitstring `1000000`. The maximum *bitstring* value is 9223372036854775807.
bitno	The bit position in *bitstring* to set to 0. An expression that resolves to a positive integer. Bit positions are counted right to left, beginning with position 0. The maximum *bitno* value is 62. A fractional *bitno* is truncated to its integer portion. A negative *bitno* generates a <FUNCTION> error.

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 or a non-numeric string 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

bitstring	The bit string, specified as an expression that resolves to a positive integer. For example, the integer 64 specifies the bitstring `1000000`. The maximum *bitstring* value is 9223372036854775807.
bitno	The bit position in *bitstring* to set to 1. An expression that resolves to a positive integer. Bit positions are counted right to left, beginning with position 0. The maximum *bitno* value is 62. A fractional *bitno* is truncated to its integer portion. A negative *bitno* generates a <FUNCTION> error.

Description

The **BITSET** function sets a single bit of *bitstring* to 1 at the bit location specified by *bitno*. Both values are specified as positive integers. *bitno* always sets the specified bit to 1. If the bit specified 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** sets it to 1 (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 or a non-numeric string 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.

If *bitno* is specified as a decimal fraction it is truncated to its integer component.

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 either 0 or 1. It then sets the bit position specified in *bitno* to bit value 1:

```
PRINT BITSET(0,0);  ! Sets bit position 0 to 1; returns integer 1
PRINT BITSET(0,1);  ! Sets bit position 1 to 1; returns integer 2
PRINT BITSET(1,0);  ! Sets bit position 0 to 1; returns integer 1
PRINT BITSET(1,1);  ! Sets bit position 1 to 1; returns integer 3
```

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 *bitstring* and *bitno* with null string values. The null string is parsed as 0:

```
PRINT BITSET("",1);   ! Returns 2; same as BITSET(0,1)
PRINT BITSET(1,"");   ! Returns 1; same as BITSET(1,0)
PRINT BITSET("","");  ! Returns 1; same as BITSET(0,0)
```

See Also

- BITRESET function
- BITNOT function
- BITTEST function

BITTEST

Tests the value of the specified bit in a bitstring.

```
BITTEST(bitstring,bitno)
```

Arguments

bitstring	The bit string, specified as an expression that resolves to a positive integer. For example, the integer 64 specifies the bitstring `1000000`. The maximum *bitstring* value is 9223372036854775807.
bitno	The bit position in *bitstring* to return the value of. An expression that resolves to a positive integer. Bit positions are counted right to left, beginning with position 0. The maximum *bitno* value is 62. A fractional *bitno* is truncated to its integer portion. A negative *bitno* generates a <FUNCTION> error.

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 or a non-numeric string 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

bitstring	A bit string, specified as an expression that resolves to a positive integer. For example, the integer 64 specifies the bitstring `1000000`. The maximum *bitstring* value is 9223372036854775807.

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**:

	bitstring1 = 0	*bitstring1* = 1
bitstring2 = 0	0	1
bitstring2 = 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 or a non-numeric string 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

BYTE

Returns the character corresponding to the specified character code.

```
BYTE(charcode)
```

Arguments

charcode	An expression that resolves to an integer code that identifies a character. For 8-bit characters, the value in *charcode* 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 **BYTE** function takes a character code and returns the corresponding character. The **SEQ** function takes a character and returns the corresponding ASCII character code. The *charcode* must be a positive, base-10 integer. A fractional number is truncated to its integer portion. A negative number, empty string, or non-numeric value returns the empty string.

Numbers from 0 to 31 are the same as standard, nonprintable ASCII codes. For example, **BYTE**(10) returns a linefeed character.

Note: **BYTE**, **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 **BYTE** function returns a single character. The corresponding 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 **BYTE** function to return the character associated with the specified character code:

```
PRINT BYTE(65);    ! Returns A.
PRINT BYTE(97);    ! Returns a.
PRINT BYTE(37);    ! Returns %.
PRINT BYTE(62);    ! Returns >.
```

The following example uses the **BYTE** 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 BYTE(letter)
  letter=letter+1
NEXT
```

See Also

- CHAR function
- UNICHAR function
- CHARS function
- SEQ function
- ObjectScript: $CHAR function

BYTELEN

Returns the number of bytes in a string.

```
BYTELEN(string)
```

Arguments

string	An expression that resolves to a string or number.

Description

The **BYTELEN** function returns the number of bytes in a specified string. **BYTELEN** counts bytes. Therefore, on a Unicode implementation of Caché each character is counted as 2 bytes; a Unicode instance of Caché counts two bytes per character even when *string* contain no Unicode characters. Use the **LEN** function to count characters, rather than 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 **BYTELEN** 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.

Examples

The following example uses the **BYTELEN** function to return the number of bytes in a string on a Unicode system:

```
PRINT BYTELEN("InterSystems");  ! Returns 24
PRINT BYTELEN(+0099.900);       ! Returns 8
PRINT BYTELEN("0099.900");      ! Returns 16
PRINT BYTELEN(CHAR(960));       ! Returns 2
PRINT BYTELEN("");              ! Returns 0
```

See Also

- COUNT function
- LEN function

CALCULATE

Returns the results of an I-type calculation.

```
CALCULATE(ITypeDictItem)
```

Arguments

ITypeDictItem	A valid virtual attribute. Must be a compiled I-type in the dictionary opened as @DICT.

Description

The **CALCULATE** function evaluates an itype expression defined in a dictionary item against data in an MVBasic program and returns the result.

CALCULATE reads the dictionary item *ITypeDictItem* from the file opened to the @DICT variable. It then evaluates the itype expression defined in attribute 2 of the dictionary item, using the data in @ID and @RECORD. Calculate also sets the @CONV, @FORMAT, and @HEADER system variables to attributes 3, 5, and 4 of the dictionary item respectively. These can be used with the **OCONV** and **FMT** functions to format the results of **CALCULATE**.

Before using **CALCULATE** you must open a file to the @DICT system variable, and assign values to @ID and @RECORD. If the itype expression uses other @variables (for example @FILE.NAME) then these need to be set as well.

CALCULATE and ITYPE Compared

The **CALCULATE** function is similar to the **ITYPE** function:

- The **ITYPE** function argument is a variable into which a dictionary item has already been read, or an itype expression assigned. The **ITYPE** function allows on-the-fly creation of itype expressions
- The **CALCULATE** function argument must be the name of an existing dictionary item which will be read by the function.

Example

The following example opens the Myfile file to the item variable, and the Myfile dictionary to the @DICT special variable. It then reads through the item variable by @ID, and uses **CALCULATE** to calculate a total of the records in item. **CALCULATE** also sets values for the @CONV and @FORMAT system variables used by the **OCONV** and **FMT** functions.

```
OPEN 'Myfile' TO item ELSE STOP 201,'MyFile'
OPEN 'DICT','Myfile' TO @DICT ELSE STOP 201,'DICT MyFile'
SELECT item TO 0
LOOP WHILE READNEXT @ID FROM 0
   DO
   READ @RECORD FROM item,@ID
   total += CALCULATE(amt_due)
   REPEAT
convtotal = OCONV(total,@CONV)
fmttotal = FMT(convtotal,@FORMAT)
PRINT fmttotal
END
```

See Also

- System Variables

CATS

Concatenates the values of corresponding elements in two dynamic arrays.

```
CATS(dynarray1,dynarray2)
```

Arguments

dynarray	An expression that resolves to a 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 element is missing, no concatenation is performed for that element, and the element value from the other dynamic array is returned.

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.

Caché MVBasic converts numbers to canonical form (resolving signs, removing leading and trailing zeros, removing a leading plus sign, removing a trailing decimal point) before concatenating. Caché MVBasic *does not* convert numeric strings to canonical form before concatenating.

If the two dynamic arrays have different numbers of elements, the returned dynamic array has the number of elements of the longer dynamic array. By default, the shorter dynamic array is padded with null string ("") value elements for the purpose of the concatenation operation. You can use the REUSE function to concatenate a default value (instead of the null string) when the dynamic arrays differ in length.

You can use the REUSE function with **CATS** to concatenate the same value to all of the elements of a dynamic array. You can use the **SPLICE** function to concatenate the elements of two dynamic arrays, supplying a separator character (or string of characters) that is inserted between the components of each element.

Examples

The following example uses the **CATS** function to concatenate the elements of two dynamic arrays:

```
ucase="A":@VM:"B":@VM:"C":@VM:"D"
lcase="aa":@VM:"bb":@VM:"cc":@VM:"dd"
PRINT CATS(ucase,lcase)
      ! returns AaaýBbbýCccýDdd
```

The following example concatenates two dynamic arrays of different lengths containing empty strings and missing elements:

```
ucase="A":@VM:"":@VM:@VM:"D"
lcase="aa":@VM:@VM:"cc":@VM:"dd":@VM:"":@VM:"ff"
PRINT CATS(ucase,lcase)
      ! returns AaaýýccýDddýýff
```

See Also

- REUSE function
- SPLICE function
- Dynamic Arrays

CHANGE

Replaces a substring in a string.

```
CHANGE(string,subout,subin[,occurrences[,begin]])
```

Arguments

string	The string in which substring substitutions are made. An expression that resolves to a string or numeric. *string* may be a dynamic array.
subout	The substring to be replaced. An expression that resolves to a string or numeric.
subin	The substring to be inserted in place of *subout*. An expression that resolves to a string or numeric.
occurrences	*Optional* — The number of occurrences of *subout* to replace with *subin*. An expression that resolves to a positive integer. If omitted, all occurrences are replaced. If used with *begin*, you can specify an *occurrences* value of -1 indicating that all occurrences of *subout* from the *begin* point to the end of the string are to be replaced.
begin	*Optional* — Which occurrence of *subout* to begin replacement with. An expression that resolves to a positive integer. If omitted, or specified as 0 or 1, replacement begins with the first occurrence of *subout*.

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, or a non-numeric string, 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:"Philadephia 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

charcode	An expression that resolves to a base-10 integer that identifies a character. For 8-bit characters, *charcode* must be a positive integer in the range 0 through 255. For 16-bit characters, *charcode* must be a positive integer 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**, **BYTE**, 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 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

- BYTE function
- 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

dynarray	An expression that resolves to a dynamic array of base-10 integers that identify characters. For 8-bit characters, each element value must be a positive integer in the range 0 through 255. For 16-bit characters, each element value must be a positive integer 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 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 AýBýCýD
```

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

string	An expression that resolves to a string.

Description

The **CHECKSUM** function generates a cyclic redundancy code (also called a checksum) corresponding to *string*. It returns this checksum as a positive 5-digit integer. A checksum can be used to determine if data has been modified or if it was incompletely transmitted. **CHECKSUM** uses an 8-bit byte sum mode to calculate the checksum.

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. Canonical and non-canonical forms of the same number have different checksums.

All string and numeric values, including zero, return a 5-digit checksum. However, if *string* is a null string 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
- $ZCRC function in ObjectScript

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(remove,replace,string)
```

Arguments

remove	One or more characters to be removed and replaced. An expression that resolves to a string or numeric.
replace	One or more characters to be inserted in place of the corresponding characters in *remove*. An expression that resolves to a string or numeric.
string	The string in which character substitutions are made. An expression that resolves to a string or numeric. *string* may be a dynamic array.

Description

The **CONVERT** function edits the value of *string* by replacing all instances of each single character in *remove* with the corresponding single characters in *replace* and returning 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 *remove* and a null string in *replace*. 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 *remove* and the replacement character in *replace*. 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 *remove* and the corresponding replacement characters in *replace*. 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 *remove*. First specify those to be replaced, then those to be removed. Specify the corresponding replacement characters in *replace*, 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 *remove* and *replace* 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 *remove* contains more characters than *replace*, the unpaired characters are deleted from the returned string. If *replace* contains more characters than *remove*, 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.

Emulation

The order of the **CONVERT** arguments differs in different emulation modes. In Caché MVBasic, the order is `remove,replace,string`. In MultiValue emulation modes, the argument order is as follows:

- `remove,replace,string`: for UniVerse, UniData, PICK, Prime, INFORMATION, PIOpen.
- `string,remove,replace`: for jBASE, Reality, Ultimate, MVBase, D3, POWER95, IN2, R83.

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

number	An expression that resolves to a number 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. This ratio is in the range of 1 to -1 (inclusive).

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 ratio.
```

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

number	An expression that resolves to a number 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. This ratio is in the range of 1 to -1 (inclusive).

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 ratio.
```

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

string	The string to search for instances of *substring*. An expression that resolves to a string.
substring	A substring to match against *string*. An expression that resolves to a string.

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 ("") **COUNT** returns a count of 0. If *substring* is an empty string, **COUNT** returns a count equal to the number of characters in *string*.

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

dynarray	The array of elements that are to be searched for instances of *substring*. An expression that resolves to a dynamic array.
substring	A substring to match against each element in *dynarray*. An expression that resolves to a string.

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 missing *dynarray* element or an element containing the empty string ("") always 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 a dynamic array element is an empty string ("") or a missing element, **COUNTS** returns a count of 0 for that element. If *substring* is an empty string, **COUNTS** returns a count for each element equal to the number of characters in that element.

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")
```

The following example returns the count of the zeros in each element. Conversion of numbers to canonical form eliminates leading and trailing zeros. Numeric strings are not converted to canonical form. The missing element and the null string element return 0 regardless of the *substring* value:

```
nums=000.1:@VM:0:@VM:@VM:"":@VM:0123.00:@VM:1230:@VM:"007.00"
PRINT COUNTS(nums,0); ! Returns 0ý1ý0ý0ý0ý1ý4
```

The following example specifies the null string as the *substring* value. It returns the count of characters in each element. Conversion of numbers to canonical form eliminates leading and trailing zeros. Numeric strings are not converted to canonical form. The missing element and the null string element return 0:

```
nums=000.1:@VM:0:@VM:@VM:"":@VM:0123.00:@VM:1230:@VM:"007.00"
PRINT COUNTS(nums,""); ! Returns 2ý1ý0ý0ý3ý4ý6
```

See Also

- COUNT function
- LENS function
- Dynamic Arrays

$DATA ($D)

Checks if a variable contains data.

```
$DATA(variable,target)
$D(variable,target)
```

Parameters

variable	The variable whose status is to be checked. *variable* may be specified as a variable or an object property with the syntax obj->property. If *variable* is not a valid variable or property name, MVBasic issues a syntax error.
target	*Optional* — A variable into which **$DATA** returns the current value of *variable*. *target* may be specified as a variable or an object property with the syntax obj->property. If *target* is not a valid variable or property name, MVBasic issues a syntax error.

Description

You can use **$DATA** to test whether a variable contains data before attempting an operation on it. **$DATA** returns status information about the specified variable. The *variable* parameter can be the name of any variable (local variable, process-private global, or global), and can include a subscript (an array element).

The possible status values that may be returned are as follows:

Status Value	Meaning
0	The variable is undefined and has no descendents.
1	The variable contains data and has no descendants. Note that the null string ("") qualifies as data.
10	The variable is undefined, but has descendants that contain data. Status 10 identifies an array element that has descendants (contains a downward pointer to another array element) but does not itself contain data.
11	The variable contains data and has descendents. Status 11 identifies a defined array element that has descendants (contains a downward pointer to another array element that also contains data). Variables of this type can be referenced in expressions.

Note: Status values 1 and 11 indicate only the presence of data, not the type of data.

If $DATA(var) returns either 0 or 10, any direct reference to *var* will result in an <UNDEFINED> error. For more information on <UNDEFINED> errors, refer to the $ZERROR special variable.

You can also use the **EXISTS** function to determine if a variable is defined and whether a dimensioned array element has descendants (subnodes).

Parameters

variable

The *variable* can be a local variable, a process-private global, or a global, with or without subscripts. It can be a multidimensional object property. If a global variable, it can include an extended global reference. If a subscripted global variable, it can be a naked global reference.

Note: **$DATA** should not be used on system variables (@ variables). It always returns 0 for all @ variables, whether or not the @ variable currently has a value.

target

An optional parameter. Specify the name of a local variable, a process-private global, or a global variable, with or without subscripts. This *target* variable does not need to be defined.

- If *variable* contains data and *target* is defined, **$DATA** copies the *variable* value to *target*.
- If *variable* contains data and *target* is undefined, **$DATA** creates the *target* variable and copies the *variable* value to *target*.
- If *variable* does not contain data and *target* is undefined, *target* remains undefined.
- If *variable* does not contain data and *target* is defined, the existing *target* value remains unchanged.

variable and *target* may be the same variable.

Examples

In the following example, a multidimensional property is used as the *variable* value. This example returns the names of all defined namespaces to the *target* parameter:

```
obj = "%ResultSet"->%New("%SYS.Namespace:List")
obj->Execute()
crt $DATA(obj->Data)               ! returns 0
obj->Next()
crt $DATA(obj->Data)               ! returns 10
crt $DATA(obj->Data("Nsp"),targ)   ! returns 1
crt targ                           ! returns namespace name
obj->Next()
crt $DATA(obj->Data("Nsp"),targ)   ! returns 1
crt targ                           ! returns namespace name
```

A similar program returns the same information using the $GET function.

Notes

Naked Global References

$DATA sets the naked indicator when used with a global variable. The naked indicator is set even if the specified global variable in not defined (Status Value = 0).

Subsequent references to the same global variable can use a naked global reference.

For further details on using **$DATA** with global variables and naked global references, see Using Multidimensional Storage (Globals) in *Using Caché Globals*.

Global References in a Networked Environment

Using **$DATA** to repeatedly reference a global variable that is not defined (for example, **$DATA(^x(1))** where ^*x* is not defined) always requires a network operation to test if the global is defined on the ECP server.

Using **$DATA** to repeatedly reference undefined nodes within a defined global variable (for example, **$DATA(^x(1))** where any other node in ^*x* is defined) does not require a network operation once the relevant portion of the global (^*x*) is in the client cache.

For further details, refer to Developing Distributed Applications in the *Caché Distributed Data Management Guide*.

$DATA and $ORDER

For related information, see **$ORDER**. Since **$ORDER** selects the next element in an array that contains data, it avoids the need to perform **$DATA** tests when looping through array subscripts.

See Also

- ASSIGNED function
- EXISTS function
- $ORDER function
- UNASSIGNED function
- Using Multidimensional Storage (Globals) in *Using Caché Globals*

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 MultiValue determines local time (and date) as follows:

- It determines the current Coordinated Universal Time (UTC) from the system clock.
- It adjusts UTC to the local time zone by using the value of the Caché special variable $ZTIMEZONE.
- It applies local time variant settings (such as Daylight Saving Time) for that time zone from the host operating system.

Caché MVBasic 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 **OCONV** function to convert date from internal format to display format.

```
PRINT DATE()
PRINT OCONV(DATE(),"D")
```

See Also

- TIMEDATE function
- OCONV function
- System Variables
- ObjectScript: $HOROLOG special variable
- SQL: NOW function

DCOUNT

Returns the number of delimited substrings in a string.

```
DCOUNT(string,delimiter)
```

Arguments

string	The string to search for instances of *delimiter*. An expression that resolves to a string.
delimiter	One or more characters used as a delimiter in *string*. An expression that resolves to a string.

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 number of characters in the *string*, plus 1.

If *string* is an empty string ("") **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

dynarray	An expression that resolves to a dynamic array.
f	An expression that resolves to an integer. Specifies the Field level of the dynamic array on which to perform the deletion. Fields are counted from 1.
v	*Optional* — An expression that resolves to an integer. Specifies the Value level of the dynamic array on which to perform the deletion. Values are counted from 1 within a Field.
s	*Optional* — An expression that resolves to an integer. Specifies 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 YorkýLondonýChicagoýBostonýLos Angeles"
PRINT DELETE(cities,1,2)
    ! Returns: "New YorkýChicagoýBostonýLos Angeles"
```

Emulation

UniData systems differ in how they handle *f*, *v*, and *s* arguments set to 0. The $OPTIONS ATTR.0IS1 (“zero is one”) provides support for this UniData feature. UniData systems ignore *v* and *s* arguments that are set to a negative number.

See Also

- DEL statement
- COUNTS function
- EXTRACT function
- Dynamic Arrays

DIV

Integer division of two values.

```
DIV(numstr1,numstr2)
```

Arguments

numstr1	The dividend. An expression that resolves to a number or numeric string.
numstr2	The divisor. An expression that resolves to a non-zero number or numeric string.

Description

The **DIV** function divides the value of *numstr1* by *numstr2*, and returns the integer quotient. It discards the fractional remainder. If a *numstr* value is a null string or a non-numeric value, **DIV** parses its value as 0 (zero).

Attempting to divide by zero generates a <DIVIDE> error, ending execution of the function and invoking an error trap handler, if available.

To perform exact division with a fractional quotient, use the division operator (/). To perform modulo division, use the **MOD** or **REM** function.

To perform division on the elements of a dynamic array, use the **DIVS** (divide corresponding elements, generate error on a zero divisor value), **DIVSZ** (divide corresponding elements, return 0 for a zero divisor value), and **MODS** (modulo division of corresponding elements) functions The **DIVS** and **DIVSZ** functions can return fractional numbers as the result (quotient) of a division operation.

Examples

The following examples use the **DIV** function to return the integer quotient 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
- DIVSZ function
- MODS function
- Operators

DIVS

Divides the corresponding elements in two dynamic arrays (zero divide not allowed).

```
DIVS(dynarray1,dynarray2)
```

Arguments

dynarray1	The dividend. An expression that resolves to a dynamic array of numeric values.
dynarray2	The divisor. An expression that resolves to a 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 empty string or a non-numeric value, **DIVS** parses its value as 0 (zero).

DIVS can return fractional numbers as the result (quotient) of a division operation. The **DIV** function can only return the integer portion of the result (quotient) of a division operation; the fractional portion is truncated.

The **DIVS** and DIVSZ functions are identical, with one difference:

- When **DIVS** encounters a 0 divisor, attempting to divide by zero generates a <DIVIDE> error, ending execution of the function and invoking an error trap handler, if available.
- When **DIVSZ** encounters a 0 divisor, it returns 0 for that element.

If the two dynamic arrays have different numbers of elements, by default the shorter dynamic array is padded so that the returned dynamic array has the number of elements of the longer dynamic array. If the shorter dynamic array is the dividend (*dynarray1*), it is padded with the required number of elements with the value of 0. If the shorter dynamic array is the divisor (*dynarray2*), it is padded with the required number of elements with the value of 1. You can also use the REUSE function to define behavior when specifying two dynamic arrays with different numbers of elements.

You can use the **NUMS** function to determine if the elements in a dynamic array are numeric. You can use the ADDS (addition), SUBS (subtraction), MULS (multiplication), MODS and MODSZ (modulo division), and PWRS (exponentiation) functions to perform other arithmetic operations on the corresponding elements of two dynamic arrays.

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 1.1ý44ý0ý-.1666666666667
```

See Also

- ADDS function
- DIVSZ function
- MODS function
- MODSZ function
- MULS function
- PWRS function

- SUBS function
- Dynamic Arrays

DIVSZ

Divides the corresponding elements in two dynamic arrays (zero divide allowed).

```
DIVSZ(dynarray1,dynarray2)
```

Arguments

dynarray1	The dividend. An expression that resolves to a dynamic array of numeric values.
dynarray2	The divisor. An expression that resolves to a dynamic array of numeric values.

Description

The **DIVSZ** and **DIVS** functions are identical, with one difference:

- When **DIVSZ** encounters a 0 divisor, it returns 0 for that element.
- When DIVS encounters a 0 divisor, it generates a <DIVIDE> error, ending execution of the function.

The **DIVSZ** 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 empty string or a non-numeric value, **DIVSZ** parses its value as 0 (zero).

DIVSZ and **DIVS** can return fractional numbers as the result (quotient) of a division operation. The **DIV** function can only return the integer portion of the result (quotient) of a division operation; the fractional portion is truncated.

If the two dynamic arrays have different numbers of elements, by default the shorter dynamic array is padded so that the returned dynamic array has the number of elements of the longer dynamic array. If the shorter dynamic array is the dividend (*dynarray1*), it is padded with the required number of elements with the value of 0. If the shorter dynamic array is the divisor (*dynarray2*), it is padded with the required number of elements with the value of 1. You can also use the REUSE function to define behavior when specifying two dynamic arrays with different numbers of elements.

You can use the **NUMS** function to determine if the elements in a dynamic array are numeric. You can use the ADDS (addition), SUBS (subtraction), MULS (multiplication), MODS and MODSZ (modulo division), and PWRS (exponentiation) functions to perform other arithmetic operations on the corresponding elements of two dynamic arrays.

Examples

The following example uses the **DIVSZ** 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 DIVSZ(a,b)
      ! returns 1.1ý44ý0ý-.1666666666667
```

The following example uses **DIVSZ** to divide the elements of two dynamic arrays, when the divisor array contains zero values:

```
a=11:@VM:22:@VM:0:@VM:-7:@VM:6
b=10:@VM:0:@VM:10:@VM:"":@VM:2
PRINT DIVSZ(a,b)
      ! returns 1.1ý0ý0ý0ý3
```

See Also

- ADDS function
- DIVS function

- MODS function
- MODSZ function
- MULS function
- PWRS function
- SUBS function
- Dynamic Arrays

DOWNCASE

Coverts alphabetic characters to lowercase.

```
DOWNCASE(string)
```

Arguments

string	An expression that resolves to a string.

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. By default it does not convert Unicode letters on a Unicode Caché instance; it passes Unicode letters through unmodified. To perform case conversion on letters in other character sets, you must set the appropriate locale.

The **OCONV** 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

string	An expression that resolves to a string or a numeric. *string* may be a dynamic array.

Description

The **DQUOTE** function returns *string* enclosed in double quotation marks. The quotation marks are part of the resulting string. Therefore, using **DQUOTE** increases the length of *string* by 2 characters. If *string* is the null string (""), **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.

Note: Some MultiValue Basic implementations (D3, for example) use **DQUOTE** and **SQUOTE** to extract quoted substrings from within a string. The Caché MVBasic quote functions do not support this functionality. Use the **FIELD** function or the [ ] operator to extract quoted substrings.

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
```

The following example uses the **DQUOTE** function to enclose a string in double quotation marks:

```
str1 = "Hello"
str2 = 'Hello'
str3 = \Hello\
PRINT str1:str2:str3;  ! Returns HelloHelloHello
PRINT LEN(str1),LEN(str2),LEN(str3);  ! Returns 5   5   5
q1 = DQUOTE(str1)
q2 = DQUOTE(str2)
q3 = DQUOTE(str3)
PRINT q1:q2:q3;       ! Returns "Hello""Hello""Hello"
PRINT LEN(q1),LEN(q2),LEN(q3);    ! Returns 7    7    7
```

Note that the quote marks are not simply string delimiters, but are part of the returned string.

See Also

- QUOTE function
- SQUOTE function
- LEN function
- PRINT statement

DTX

Converts a number from decimal to hexadecimal.

```
DTX(decnum[,width])
```

Arguments

decnum	An expression that resolves to an integer.
width	*Optional* — An expression that resolves to a positive integer. *width* specifies 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 *decnum* is a positive integer, **DTX** returns the number of hexadecimal digits needed to express it. If *decnum* is a negative integer, **DTX** returns high values. For example, DTX(-1) returns FFFFFFFFFFFFFFFF. If you specify *decnum* as a fractional number, **DTX** generates a <FUNCTION> error.

The optional *width* argument pads the return value with leading zeros. If *width* is equal to or smaller than the number of hexadecimal digits in the return value, *width* is ignored. If *width* is larger than the needed number of hexadecimal digits, **DTX** pads the returned hexadecimal number with leading zeros. With a negative *decnum*, if *width* is larger than 16, it pads the returned hexadecimal number with leading zeros. If you specify *width* as a fractional number, **DTX** truncates it to the integer portion. If you specify *width* as a negative number, *width* is ignored.

If *decnum* is zero, the null string, or a non-numeric string, **DTX** returns 0; *width* padding is applied. 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 return positive integers 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"
```

The following examples return negative integers converted to hexadecimal:

```
PRINT DTX(-199);        ! Returns "FFFFFFFFFFFFFF39"
PRINT DTX(-199,4);      ! Returns "FFFFFFFFFFFFFF39"
PRINT DTX(-199,17);     ! Returns "0FFFFFFFFFFFFFF39"
```

The following examples all return zero. Zero padding is provided, if specified:

```
PRINT DTX(0);       ! Returns "0"
PRINT DTX(-0);      ! Returns "0"
PRINT DTX(0,4);     ! Returns "0000"
PRINT DTX("",4);    ! Returns "0000"
PRINT DTX("foo",4); ! Returns "0000"
```

See Also

- XTD function

EBCDIC

Converts a string from ASCII to EBCDIC.

```
EBCDIC(string)
```

Arguments

string	An expression that resolves to a string.

Description

The **EBCDIC** function takes a string of characters and returns the ASCII code representation for each character. If you supply a string of ASCII code characters, **EBCDIC** returns the corresponding EBCDIC character(s). This is the inverse of the **ASCII** function. The *string* cannot contain Unicode characters.

If *string* is a number, it is converted to canonical representation before **EBCDIC** processing. If *string* is a quoted numeric string, no conversion is performed before **EBCDIC** processing.

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 ASCII code string:

```
astring=ASCII("ABCDEFG")
estring=EBCDIC(astring)
PRINT estring
  ! returns "ABCDEFG"
```

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

EOF(ARG.)

Returns whether the command line pointer is past the last argument.

```
EOF(ARG.)
```

Arguments

None. The keyword `ARG.` (note the period at end of this keyword) is the only allowed value, and is mandatory. The `ARG.` keyword is not case-sensitive.

Description

The **EOF(ARG.)** function returns a boolean value indicating whether the command line pointer is positioned beyond the last command line argument. It returns 1 if the command line pointer is positioned beyond the last command line argument. Otherwise, it returns 0.

The **GET(ARG.)** statement moves the command line argument pointer and retrieves the argument value. The **SEEK(ARG.)** statement moves the command line argument pointer without retrieving a value.

See Also

- GET(ARG.) statement
- SEEK(ARG.) statement

EQS

Performs an equality comparison on elements of two dynamic arrays.

```
EQS(dynarray1,dynarray2)
```

Arguments

dynarray	An expression that resolves to a 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** converts numeric values to canonical form, removing signs and leading and trailing zeros, before making the comparison. **EQS** does not convert numeric strings to canonical form before making the comparison. If an element is missing, or has a null string value 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 the two dynamic arrays have different numbers of elements, the returned dynamic array has the number of elements of the longer dynamic array. By default, unmatched elements return 0 (not equal). That is, the **EQS** comparison of each element in the longer dynamic array that has no corresponding element in the shorter dynamic array always returns 0 (not equal), even when the value of the longer array element is 0 or the null string, or is a missing element within the dynamic array. You can also use the REUSE function to define behavior when specifying two dynamic arrays with different numbers of elements.

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 1ý0ý0ý1
```

The following example compares various element values to 0:

```
a=0:@VM:0:@VM:0:@VM:0
b="":@VM:-0.00:@VM:@VM:"foo"
PRINT EQS(a,b)
      ! returns 1ý1ý1ý1
```

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

string	An expression that resolves to a string.
substring	An expression that resolves to a substring found within *string*.
replacement	An expression that resolves to the substring used to replace *substring*.
occurrence	*Optional* — An expression that resolves to an integer count specifying how many occurrences of *substring* to replace. The default is to replace all occurrences. A value of 0 replaces all occurrences. When using *begin*, you must specify an *occurrence* value.
begin	*Optional* — An integer count specifying the instance of *substring* with which to begin replacement. The default is to begin with the first instance of *substring*.

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 empty string (""), **EREPLACE** removes instances of *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 the specified instances of a substring:

```
x="The slow brown fox slowly leapt"
PRINT EREPLACE(x,"slow","quick",1)
   ! Returns "The quick brown fox slowly leapt"
PRINT EREPLACE(x,"slow","quick",0,2)
   ! Returns "The slow 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

EXISTS

Returns the existence status of variables and their dimensioned array subnodes.

```
EXISTS(varname)
```

Arguments

varname	Name of a variable to test for existence, and/or the presence of dimensioned array subnodes.

Description

The **EXISTS** function returns an integer code indicating whether a variable is defined (1) or not defined (0). It can also indicate that the specified variable is not defined, but that the variable has defined subscripts. The *varname* parameter can be the name of any variable (local variable, process-private global, or global), and can include a subscript (an array element).

EXISTS returns an integer code indicating that the specified variable is:

- 0: undefined and has no subnodes.
- 1: defined and has no subnodes.
- 2: undefined but has defined subnodes.
- 3: defined and has defined subnodes.

Similar information can be returned using the **$DATA** function.

Note: **EXISTS** should not be used on system variables (@ variables). It always returns 0 for all @ variables, whether or not the @ variable currently has a value.

Example

The following example shows the four possible **EXISTS** return values. The specified variables are all process-private globals:

```
^||a="salt"
^||b(1)="carrot"
^||c="fruit"
^||c(1)="apple"
PRINT EXISTS(^||a); ! returns 1
PRINT EXISTS(^||b); ! returns 2
PRINT EXISTS(^||c); ! returns 3
PRINT EXISTS(^||z); ! returns 0
```

See Also

- DIM statement
- ASSIGNED function
- $DATA function
- ISOBJECT function
- Variables

EXP

Returns e (the base of natural logarithms) raised to a power.

```
EXP(number)
```

Arguments

number	An expression that resolves to a number within the following range: On a Windows system, if the value of *number* is greater than 335.601, a <MAXNUMBER> error occurs; if the value of *number* is less than -295.424, **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. If *number* is 0, the null string (""), or a non-numeric value, **EXP** parses *number* as 0 and returns 1.

The **EXP** function complements the action of the **LN** function and is sometimes referred to as the antilogarithm.

In 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

dynarray	An expression that resolves to a dynamic array.
f	An expression that resolves to an integer specifying the Field level of the dynamic array from which to access the data. Fields are counted from 1.
v	*Optional* — An expression that resolves to an integer specifying the Value level of the dynamic array from which to access the data. Values are counted from 1 within a Field.
s	*Optional* — An expression that resolves to 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, or a non-numeric value 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, or a non-numeric value has no effect on retrieving the data value in the level above it.

You can also use the <> 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,"")
```

Emulation

UniData systems differ in how they handle *f*, *v*, and *s* arguments set to 0. The $OPTIONS ATTR.0IS1 (“zero is one”) provides support for this UniData feature. UniData systems ignore *v* and *s* arguments that are set to a negative number.

See Also

- FIND statement
- FINDSTR statement
- REMOVE statement
- REPLACE function
- Dynamic Arrays
- Variables

FADD

Adds two floating point numbers.

```
FADD(num1,num2)
```

Arguments

num	An expression that resolves to a numeric value.

Description

The **FADD** function adds two numbers and returns the result. If a *num* value is 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

num1	The dividend. An expression that resolves to a number or numeric string.
num2	The divisor. An expression that resolves to a non-zero number or numeric string.

Description

The **FDIV** function divides the value of *num1* by *num2*, and returns the quotient. If a value is 0, a null string, or a non-numeric value, **FDIV** parses it as 0 (zero). If *num1* is 0, **FDIV** returns a result of 0. If *num2* is 0, **FDIV** generates a <DIVIDE> 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

string	An expression that resolves to a string. 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.
delimiter	An expression that resolves to a single character, specified as a number or a string. This character is used as a delimiter to identify substrings. This character cannot also be used as a data value within *string*. The delimiter characters used in dynamic arrays are listed in the Dynamic Arrays general concepts page of this manual.
count	An expression that resolves to an integer that specifies which substring to return from the target string. Substrings are separated by a *delimiter*, 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 *count* value of 0, a negative number, the null string, or a non-numeric string is the same as *count*=1.
range	*Optional* — An expression that resolves to an integer specifying the number of delimited substrings to return, starting with *count*. 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.

Emulation

By default Caché MVBasic permits only a single-character *delimiter*. jBASE emulation permits a multi-character *delimiter*. This option is set using the FULL.DELIM option.

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
```

The following example uses *count* and *range*:

```
colors="Red^Green^Blue^Yellow^Orange^Black"
PRINT FIELD(colors,"^",2,3)
```

Returns “Green^Blue^Yellow”.

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

dynarray	An expression that resolves to a dynamic array. The source dynamic array from which a dynamic array of substrings is to be extracted.
delimiter	An expression that resolves to a single character, specified as a number or a string. This character is used as a delimiter to identify substrings within elements. This character cannot also be used as a data value within *dynarray*. The delimiter characters used in dynamic arrays are listed in the Dynamic Arrays general concepts page of this manual.
count	An expression that resolves to an integer that specifies which substring to return from each element of *dynarray*. Substrings are separated by a *delimiter*, 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 *count* value of 0, a negative number, the null string, or a non-numeric string is the same as *count*=1.
range	*Optional* — An expression that resolves to an integer specifying the number of delimited substrings to return for each element, starting with *count*. 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: 617ý401ý603ý508
```

See Also

- FIELD function
- GROUP function
- Strings
- Dynamic Arrays

FIELDSTORE

Replaces data in a delimited string.

```
FIELDSTORE(string,delimiter,count,multiple,newval)
```

Arguments

string	An expression that resolves to a string. The source string to be modified. *string* can be a dynamic array.
delimiter	An expression that resolves to a single character that serves as a delimiter within *string*
count	An expression that resolves to an integer that specifies which delimited string to use as the starting point for the replacement operation.
multiple	An expression that resolves to an integer specifying how many delimited strings to replace with *newval*.
newval	An expression that resolves to a string. The data to be inserted.

Description

The **FIELDSTORE** replaces one or more delimited substrings in *string* with a specified *newval*, then returns the resulting string. The source *string* remains unchanged. **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 return a string that replaces the first delimited substring in the 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 return a string that replaces the second Value Mark delimited substring in the dynamic array:

```
cities="New York":@VM:"London":@VM:"Chicago":@VM:"Boston":@VM:"Los Angeles"
PRINT cities
    ! Returns: "New YorkýLondonýChicagoýBostonýLos Angeles"
PRINT FIELDSTORE(cities,CHAR(253),2,1,"Providence")
    ! Returns: "New YorkýProvidenceýChicagoýBostonýLos 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 YorkýLondonýChicagoýBostonýLos Angeles"
PRINT FIELDSTORE(cities,CHAR(253),2,3,"Providence")
    ! Returns: "New YorkýProvidenceýLos 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

filevar	An expression that resolves to a file variable name used to refer to the file in Caché MVBasic.
key	An expression that resolves to an integer code used to specify what file information to return. Available values are 0 through 3.

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. If *filevar* is not valid and *key* is 1 through 3 (inclusive), **FILEINFO** returns the empty string.

The following are the available *key* options and return values:

0	File variable: 1 if *filevar* is valid. Otherwise 0.
1	VOC name: The VOC name of a MultiValue file. For example, `myfile`. If *filevar* does not refer to a MultiValue file, returns a null string.
2	Pathname or global name: For a MultiValue file, the name of a Caché global variable. For example, `^|"USER"|myfile`. 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 storage type: 0=unknown. 1=top level global (static hashed file) the default for MultiValue data files. 2=subscripted global subnode. 4=directory. 5=sequential file.

Other MultiValue implementations may support higher *key* option values; these are not supported by Caché MVBasic.

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

number	An expression that resolves to a number or a numeric string.
precision	*Optional* — An expression that resolves to an integer specifying the number of decimal digits of precision. The default is 4.
mode	*Optional* — An expression that resolves to a boolean flag that specifies whether to round or truncate *number*. 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 fractional digits. The *precision* is the maximum number of fractional 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

string	An expression that resolves to a string or number to be formatted for display.
format	An expression that resolves to a string consisting of positional letter and number codes specifying the display format for *string*.

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:

```
wfRnm
```

w	*Optional* — The overall width of the display field, specified as a positive integer. Used to impose a uniform width (number of characters) on *string*. Different operations are performed if *w* is larger or smaller than the length of *string*, as described below.
f	*Optional* — A fill character, specified as a single character. (Certain fill characters, as described below, must be specified as a quoted string.) You must specify *w* to use *f*. If you specify *w*, but do not specify *f*, it defaults to the space character.
R	*Optional* — The letter “R” or “L” specifying right or left justification. This letter code is not case-sensitive. If you do not specify a letter code, **FMT** defaults to left justification. (The letters “T” and “U” are synonyms for “L”).
n	*Optional* — A positive integer in the range 0 through 9 that specifies the number of fractional digits to the right of the decimal place. If you specify *n*, it must either be the only code in *format*, or it must be preceded by the letter “R” or “L”. If you do not specify *n*, **FMT** defaults to number of fractional digits in *string*. Zero-padding and rounding are applied as needed.
m	*Optional* — A positive integer in the range 0 through 9 that specifies the repositioning of the decimal point. Both *n* and *m* must be specified. The number 4 specifies do not reposition the decimal point (the default). Integers higher than 4 move the decimal point to the left; integers lower than 4 move the decimal point to the right. Zero-padding and rounding are applied as needed.

There are two basic uses of *format*:

- To return fractional numbers in a standard form.
- To return strings in a standard form.

FMT also supports a different format to support other legacy platform styles: Rfw.

Formatting Numbers

For fractional numbers, the most basic *format* is `"Rn"`, where “R” is either the letter “R” specifying right justification or the letter “L” specifying left justification, and *n* is the number of digits to the right of the decimal point to display. If *string* is an integer or has fewer fractional digits than *n*, zero padding is added. If *string* has more digits than *n*, the number is rounded to the specified number of fractional digits. If *n* is zero, the number is rounded to an integer and the decimal point is removed. If *string* is less than 1, specifying *n* supplies a zero (0) to the left of the decimal point. If *string* contains any character other than a number, the decimal point character, or a plus or minus sign, **FMT** does no zero padding or rounding.

A more complex example of *format* is `"10#R5"`, where “10” is the overall width of the display field elements, “#” is the fill character to use to fill out a display field element. Because “R” indicates right justification, these fill characters will appear to the left of the *string* value. The *n* value of 5 indicates that the *string* value is to have 5 digits to the right of the decimal place. When an *n* value is present, **FMT** only formats a number; a non-numeric string is returned unchanged.

If *string* is 0, **FMT** applies numeric formatting; it returns the value zero with *n* fractional digits and *m* shifting of the decimal point. If *string* is the null string (""), **FMT** numeric formatting returns the null string. This null string behavior is emulation-dependent: Caché, jBASE, and UniData emulations treat a null *string* as null. The other MultiValue emulations treat a null *string* as zero, and apply numeric formatting.

Formatting Strings

For strings, the most basic *format* is `"wf"`, where “w” is an integer specifying width and *f* is a literal fill character (for example `"9^"`). You can use *w* (width) and *f* (fill) formatting to make a display field a standard width. By default, the justification is “L” (left); you can, of course, specify “R” for right justification.

The *w* (width) value may be larger than, equal to, or smaller than the number of characters (including the decimal point) of *string*. If *string* is a fractional number, *w* is applied after **FMT** adjusts the number of fractional digits (by rounding or zero padding).

- If *w* is greater than the length of *string*, **FMT** appends *f* fill characters to *string* making the resulting string *w* characters in length. If “L” (left justification) fill characters are applied to the end of the string; if “R” (right justification) fill characters are applied to the beginning of string.
- If *w* is equal to the length of *string* (after rounding or zero padding of fractional digits), no operation is performed.
- If *w* is less than the length of *string*, **FMT** inserts a Text Mark (@TM, CHAR(251)) character after every *w* count of characters. If “L” (left justification), characters are counted forward from the beginning of the string; if “R” (right justification), characters are counted backward from the end of the string. **FMT** then appends *f* fill characters so that all Text Mark delimited substring elements are *w* characters long (the Text Mark itself is not counted). If “L” (left justification) fill characters are applied to the end of the string; if “R” (right justification) fill characters are applied to the beginning of string.

For example:

- In wfRnm format: `FMT("ABC","15.R")`, where '15' is the Width, and '.' is the Fill character.
- In Rfw format: `FMT("ABC","R.15")`, where '15' is the Width, and '.' is the Fill character.

The fill character is optional; if omitted, filling is done with blank spaces. The fill character cannot be the same as the *format* string delimiter character. If the fill character is a number, the backslash (\), or the letters “L”, “R”, “T”, or “U” it must be enclosed in string delimiter quotes that are different than the *format* string. For example: `"10'0'R2"`. You cannot use the backslash as a string delimiter for the fill character.

FMT is CEMU dependent. So, for example, for CEMU ULTIMATE, the fill character must be one of the following: '#%*'.

Implicit Formatting

The same formatting codes can be used with the **CRT**, **PRINT**, or **DISPLAY** commands. This is known as implicit formatting, because the **FMT** function is not specified. For example:

```
PRINT 1.2 "R4"          ;! Returns 1.2000
```

Which is exactly equivalent to:

```
PRINT FMT(1.2,"R4")     ;! Returns 1.2000
```

The formatting codes apply only to the argument that they immediately precede. For example, the following two statements are functionally identical:

```
CRT "Over":"There" "R#20"
CRT "Over":FMT("There","R#20")
```

Implicit formatting is just one of the ways that these commands can interpret a second argument. Many of the OCONV function conversion codes can also be used with implicit (or explicit) formatting. For example, date conversion:

```
  PRINT OCONV(14100,"D");   ! "08 AUG 2006"
  PRINT 14100 "D";          ! "08 AUG 2006"
  PRINT FMT(14100,"D");     ! "08 AUG 2006"
```

Because the letter codes “R” and “L” are used as formatting (**FMT**) codes, the corresponding **OCONV** conversion codes cannot be used for implicit formatting.

An expression can be used to specify an implicit formatting string, with the following limitation: a Caché global variable cannot be used for implicit formatting. This is because the caret (^) that Caché uses to indicate a global is often interpreted in these contexts as the exponentiation operator. A Caché global can be used for explicit formatting in the **FMT** function.

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
- OCONV function
- RIGHT function
- DISPLAY statement
- CRT statement
- PRINT statement
- ObjectScript $MVFMT function, described in the *Caché ObjectScript Reference*

FMTS

Formats each element of a dynamic array for display.

```
FMTS(dynarray,format)
```

Arguments

dynarray	An expression that resolves to a dynamic array to be formatted for display.
format	An expression that resolves to a string consisting of positional letter and number codes specifying the display format for the elements of *dynarray*.

Description

The **FMTS** function returns the *dynarray* value with each element formatted as specified by *format*. This formatting may include justification, character filling, and the rounding or zero padding of numeric element values. The most common use for **FMTS** is to provide a uniform display format for fractional numbers.

The *format* string has the following format:

```
wfRn
```

w	*Optional* — The overall width of the display field, specified as a positive integer. Used to impose a uniform width (number of characters) for each element of *dynarray*. Different operations are performed if *w* is larger or smaller than the length of an element, as described in the FMT function.
f	*Optional* — A fill character, specified as a single character. If the fill character is a number, the backslash (\), or the letters “L”, “R”, or “T” it must be enclosed in string delimiter quotes. You must specify *w* to use *f*. If you specify *w*, but do not specify *f*, it defaults to the space character.
R	*Optional* — The letter “R” or “L” specifying right or left justification. This letter code is not case-sensitive. If you do not specify a letter code, **FMTS** defaults to left justification.
n	*Optional* — The number of fractional digits to the right of the decimal place, specified as a positive integer. If you specify *n*, it must either be the only code in *format*, or it must be preceded by the letter “R” or “L”. If you do not specify *n*, **FMTS** defaults to number of fractional digits in *string*.

There are two basic uses of *format*:

- To return fractional numbers in a standard form. **FMTS** can be used to round a fractional number to an integer or to a specified number of fractional digits. If the specified number of fractional digits is larger than the number of fractional digits in the element value, **FMTS** zero pads the additional digits.
- To return strings in a standard form. **FMTS** can left justify or right justify a string and add a fill character before or after to make each element contain the same number of characters.

For further details on *format* codes, refer to the FMT function.

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:"4.123456":@VM:"0"
PRINT FMTS(nums,"R4")
    ! Returns: 1.2000ý2.4500ý3.0000ý4.1235ý0.0000
```

The following example uses “wfL” formatting to format the elements of a dynamic array so that all elements display seven characters. Note that the ^ character is used as the fill character:

```
flints="FRED":@VM:"BARNEY":@VM:"WILMA":@VM:"PEBBLES"
PRINT FMTS(flints,"7^L")
    ! Returns: FRED^^^ýBARNEY^ýWILMA^^ýPEBBLES
PRINT FMTS(flints,"7^R")
    ! Returns: ^^^FREDý^BARNEYý^^WILMAýPEBBLES
```

See Also

- FMT function
- LEN function
- RIGHT function
- Dynamic Arrays

FMUL

Multiplies two floating point numbers.

```
FMUL(num1,num2)
```

Arguments

num	An expression that resolves to a number or numeric string.

Description

The **FMUL** function multiplies two numbers and returns the product. If a *num* value is 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

FOLD

Divides a string into substring units separated by a delimiter.

```
FOLD(string,length[,delim])
```

Arguments

string	An expression that resolves to a string or numeric expression.
length	An expression that resolves to an integer specifying the maximum number of characters per substring. If you specify a fractional value, **FOLD** truncates it to its integer portion. By default, any value less than 1 returns the empty string (see Emulation section below).
delim	*Optional* — An expression that resolves to the delimiter character to use. If omitted or set to the empty string ("") the default is @FM. This argument is provided for D3 compatibility. (Note that in D3 the default delimiter is @VM.)

Description

The **FOLD** function returns the specified string as a string divided into subunits by delimiter characters. This delimiter character by default is the @FM (also known as @AM) field mark character. **FOLD** places field mark delimiters as follows:

- If a space character is encountered within *length* number of characters, **FOLD** replaces the space character with a field mark delimiter, then begins counting *length* characters from that point. If there are multiple space characters within *length*, **FOLD** only replaces the last space character prior to reaching the *length* character count. If the *length* character is a space character, **FOLD** replaces it with a field mark delimiter.
- If a space character is *not* encountered within *length* number of characters, **FOLD** inserts a field mark delimiter, then begins counting *length* characters from that point.

FOLD does not place a field mark delimiter before the first character or after the last character of *string*, unless the first or the last character is a space character. If the input string contains a field mark delimiter character, it is counted as an ordinary character. Note that because field mark delimiters replace spaces, but are inserted between non-space characters, the returned string can be vary from being the same length to being significantly longer than the input string.

FOLD counts characters, not bytes. You can use the **LEN** function to determine the number of characters in a string. You can use the **LENS** function to determine the number of characters in each delimited substring.

The *string* argument can be a quoted string or a numeric expression. If *string* is the empty string, **FOLD** returns the empty string.

If *length* is equal to or larger than the number of characters in *string*, *string* is returned unchanged. If *length* is less than 1 or a non-numeric string, **FOLD** returns the empty string.

If *string* is a numeric expression, prior to performing the **FOLD** operation 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. Numeric strings are not converted to canonical form.

Emulation

You can change the behavior of **FOLD** by setting the following $OPTIONS statement values:

- FOLD.DELIM.VM sets the delimiter character to @VM (value mark) rather than @FM (field mark). This provides compatibility with D3 applications.
- FOLD.LEN.1 sets the behavior for a *length* of less than 1, by having **FOLD** default to a *length* of 1. Otherwise, a value less than 1 (for example, 0, .5, or –1) returns the empty string. This provides compatibility with jBASE applications.

Examples

The following example uses the **FOLD** function to return a string delimited by Field Marks (þ) into fixed-length units specified by *length*:

```
PRINT FOLD("InterSystems",3); ! Returns "IntþerSþystþems"
PRINT FOLD(+0099.900,2);       ! Returns "99þ.9"
PRINT FOLD("+0099.900",2);      ! Returns "+0þ09þ9.þ90þ0"
```

The following example uses the **FOLD** function to return a string delimited according to the spaces in the source string and the *length* count:

```
PRINT FOLD("The quick brown fox",19);
   ! Returns "The quick brown fox"
PRINT FOLD("The quick brown fox",16);
   ! Returns "The quick brownþfox"
   ! (þ delimiter replaces the last space;
   !  character 16 is a space)
PRINT FOLD("The quick brown fox",15);
   ! Returns "The quick brownþfox"
   ! (þ delimiter inserted at count=15)
PRINT FOLD("The quick brown fox",14);
   ! Returns "The quickþbrown fox"
   ! (þ delimiter replaces the last space prior
   !  to count=14)
PRINT FOLD("The quick brown fox",5);
   ! Returns "Theþquickþbrownþfox"
PRINT FOLD("The quick brown fox",4);
   ! Returns "Theþquicþkþbrowþnþfox"
PRINT FOLD("The quick brown fox",3);
   ! Returns "Theþquiþckþbroþwnþfox"
PRINT FOLD("The quick brown fox",2);
   ! Returns "Thþeþquþicþkþbrþowþnþfoþx"
```

See Also

- BYTELEN function
- LEN function
- LENS function

FSUB

Subtracts two floating point numbers.

```
FSUB(num1,num2)
```

Arguments

num1	The minuend. An expression that resolves to a number or numeric string.
num2	The subtrahend. An expression that resolves to a number 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 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

dynarray	An expression that resolves to a 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 converts numbers to canonical form, resolving multiple signs and removing leading and trailing zeros from element values before making the comparison. If an element value is a missing element, a null string, or a non-numeric value, **GES** assigns it a value of 0 for the purpose of this comparison.

If the two dynamic arrays have different numbers of elements, the returned dynamic array has the number of elements of the longer dynamic array. By default, the shorter dynamic array is padded with 0 value elements for the purpose of comparison. You can also use the REUSE function to define behavior when specifying two dynamic arrays with different numbers of elements.

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 1ý1ý0ý1
```

The following example compares dynamic arrays of different lengths. Note that missing elements within the dynamic array (@VM:@VM) are compared, but unmatched elements from the longer array always return 1:

```
a=11:@VM:21:@VM:@VM:41
b=10:@VM:@VM:30:@VM:40:@VM:50:@VM:60
PRINT GES(a,b)
      ! returns 1ý1ý0ý1ý1ý1
PRINT GES(b,a)
      ! returns 0ý0ý1ý0ý1ý1
```

See Also

- EQS function
- GTS function
- LES function
- LTS function
- Dynamic Arrays

$GET

Returns the data value of a specified variable.

```
$GET(variable[,default])
```

Parameters

variable	A local, global, or process-private global variable, subscripted or unsubscripted. The variable may be undefined. *variable* may be specified as an object property with the syntax obj->property.
default	*Optional* — An expression that resolves to a value to be returned if the variable is undefined. If *default* is a variable it must be defined, even when not used.

Description

$GET returns the data value of a specified variable. The handling of undefined variables depends on whether you specify a *default* parameter.

- **$GET**(*variable*) returns the value of the specified variable, or the null string if the variable is undefined. The *variable* parameter value can be the name of any variable, including a subscripted array element (either local or global).
- **$GET**(*variable*,*default*) provides a default value to return if the variable is undefined. If the variable is defined, **$GET** returns its value.

Handling Undefined Variables

$GET defines handling behavior if a *specified* variable is undefined. The basic form of **$GET** returns a null string ("") if the specified variable is undefined.

$DATA tests if a specified variable is defined. It returns 0 if the variable is undefined.

The *Undefined* property of the Config.Miscellaneous class defines handling behavior for *all* undefined variables system-wide. The **Undefined()** method of the %SYSTEM.Process class defines handling behavior for *all* undefined variables for the current process. Setting this property or method has no effect on **$GET** or **$DATA** handling of specified variables.

Parameters

variable

The variable whose data value is to be returned. It can be a local variable, a global variable, or a process-private global variable. It can be either subscripted or unsubscripted. It can be a multidimensional object property. The variable does not need to be a defined variable. The variable can be defined and set to the null string (""). If a global variable, it can contain an extended global reference. If a subscripted global variable, it can be specified using a naked global reference. Even when referencing an undefined subscripted global variable, *variable* resets the naked indicator, affecting future naked global references, as described below.

$GET should not be used on system variables (@ variables). It always returns the null string for all @ variables, whether or not the @ variable currently has a value.

default

The data value to be returned if *variable* is undefined. It can be any expression, including a local variable, a global variable, or a process-private global variable, either subscripted or unsubscripted. *default* can be a system variable (@ variable), with or without a non-null value.

If *default* is a local variable, a global variable, or a process-private global variable, it must be defined variable. If *default* is an undefined variable, **$GET** issues an <UNDEFINED> error, even when *variable* is defined.

If *default* is a global variable, it can contain an extended global reference. If a subscripted global variable, it can be specified using a naked global reference. If present, *default* resets the naked indicator, affecting future naked global references.

Examples

In the following example, the variable *test* is defined and the variable *xtest* is undefined:

```
test="banana"
tdef=$GET(test)
tundef=$GET(xtest)
PRINT tdef    ! $GET returned value of test
PRINT tundef  ! $GET returned null string for xtest
PRINT $GET(xtest,"none")
              ! $GET returns default of "none" for undefined xtest
```

In the following example, a multidimensional property is used as the *variable* value. This example returns the names of defined namespaces:

```
obj = "%ResultSet"->%New("%SYS.Namespace:List")
obj->Execute()
crt $GET(obj->Data,"none") ! returns "none"
obj->Next()
crt $GET(obj->Data,"none") ! returns "none"
crt $GET(obj->Data("Nsp")) ! returns "%SYS"
obj->Next()
crt $GET(obj->Data("Nsp")) ! returns next namespace
obj->Next()
crt $GET(obj->Data("Nsp")) ! returns next namespace
```

A similar program returns the same information using the $DATA function.

Notes

$GET Compared to $DATA

$GET provides an alternative to **$DATA** tests for both undefined variables ($DATA=0) and array nodes that are downward pointers without data ($DATA=10). If the variable is either undefined or a pointer array node without data, **$GET** returns a null string ("") without an undefined error.

Note that **$DATA** tests are more specific than **$GET** tests because they allow you to distinguish between undefined elements and elements that are downward pointers only.

See Also

- $DATA function
- $ZUTIL(18) Set Undefined Variable Handling function in *Caché ObjectScript Reference*
- $ZUTIL(69,0) Set Undefined Variable Handling System-wide function in *Caché ObjectScript Reference*
- Using Multidimensional Storage (Globals) in *Using Caché Globals*

GETENV

Returns the value of the specified environment variable.

```
GETENV(name)
```

Arguments

name	An expression that resolves to the name of an environment variable, specified as a quoted string.

Description

The **GETENV** function returns the current value of the specified environment variable. Environment variable names are not case-sensitive.

If the specified *name* is a literal or a defined variable that is not an environment variable, **GETENV** returns an empty string. If *name* is a system variable (@ variable) that has a current value, **GETENV** returns an empty string; if *name* is an @ variable that does not have a current non-null value, **GETENV** generates an <ILLEGAL VALUE> error.

On a Windows system, you can display a list of all of your environment variables by issuing the SH MultiValue command from the MultiValue Shell prompt, as follows:

```
USER:SH set
```

To create a file containing this list, you can use the following MultiValue command:

```
USER:[ WRITE $ZF(-1,"set>c:\temp\myenvset.txt")
```

Examples

The following example returns the PATH environment variable:

```
PRINT GETENV("PATH")
```

The following example returns the operating system environment variable:

```
PRINT GETENV("OS")
```

The following example returns the current username environment variable:

```
PRINT GETENV("USERNAME")
```

See Also

- SYSTEM function

GETPTR

Returns print channel details.

```
GETPTR(channel)
```

Arguments

channel	An expression that resolves to an integer specifying an existing print channel. Valid values are 0 through 255.

Description

The **GETPTR** function returns a string consisting of a comma-separated list of channel settings. These are the same settings defined using the SETPTR command, as described in *The Caché MultiValue Spooler*.

The *channel* can be specified as an integer from 0 through 255 (inclusive). Integers outside this range return the empty string. Fractional values are truncated to the integer portion. If *channel* is the empty string ("") or a non-numeric value, **GETPTR** returns channel settings for Channel 0.

Example

The following example uses the **GETPTR** function to return the channel settings for *channel* 1:

```
PRINT GETPTR(1)
```

It returns a string such as the following: `0,132,66,3,3,1, EJECT`. For an explanation of these values, refer to SETPTR in *The Caché MultiValue Spooler*.

See Also

- GETPU function
- SETPTR command, in The Caché MultiValue Spooler.

GETPU

Returns the name of the output device for a print channel.

```
GETPU(channel)
```

Arguments

channel	An expression that resolves to an integer specifying an existing print channel. Valid values are 0 through 255.

Description

The **GETPU** function returns the name of the output device most recently used by the specified print channel. **GETPU** can be used both for **SETPTR** mode 1 (spooler) and **SETPTR** mode 3 (&HOLD& file) channel assignments.

- For mode 1 (spooler) output, the returned value is a Caché global name, such as ^%MV.SPOOL(13). This global is the spooler to which the job is being spooled. **GETPU** in this mode only returns a value when a print job is open; otherwise, it returns a null string, with one exception. If the print job is closed, but the previous print job was marked as a HOLD job using the **SETPTR** HOLD option, **GETPU** returns the global name for that print job.
- For mode 3 (&HOLD&) output, the returned value is the full path name of a file that **GETPU** creates to contain the spooler output. For example, a Windows file such as: `c:\InterSystems\cache\mgr\user\&hold&\P#0000_0025`. If a print job is currently open, **GETPU** returns details about the print job. If there is no print job open and the application executed a **SETPTR** command, **GETPU** returns the job ID that will be created when the next print job is created. If there is neither an open print job nor a **SETPTR** setting, **GETPU** returns the last job created.

If the *channel* value is a nonexistent print channel or an invalid value, **GETPU** returns the empty string.

See Also

- GETPTR function
- SETPTR command in the "Spooler Commands" chapter of *The Caché MultiValue Spooler*.

GETREM

Returns the position of the Remove pointer in a dynamic array.

```
GETREM(dynarray)
```

Arguments

dynarray	An expression that resolves to a dynamic array. *dynarray* must be a variable, it cannot be a literal dynamic array string.

Description

The **GETREM** function returns a positive integer indicating the current position of the remove pointer within a dynamic array. This remove pointer can be explicitly incremented by the **SETREM** statement, and is automatically incremented/decremented by the **REMOVE** function, the **REMOVE** statement, and the **REVREMOVE** statement. **GETREM** returns this pointer value. This value is an integer count character position within *dynarray*.

REMOVE and **REVREMOVE** extract successive data elements from a dynamic array. They establish a pointer specifying the position for the next extract from that dynamic array. 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 these operations, **GETREM** returns 0. If *dynarray* is changed, its pointer is reset to 0. If *dynarray* is a literal string, **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

- SETREM statement
- REMOVE statement
- REVREMOVE statement
- REMOVE function
- Dynamic Arrays

GROUP

Returns the specified substring, based on a delimiter.

```
GROUP(string,delimiter,count[,range])
```

Arguments

string	An expression that resolves to 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.
delimiter	An expression that resolves to 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 *string*. The delimiter characters used in dynamic arrays are listed in the Dynamic Arrays general concepts page of this manual.
count	An expression that resolves to an integer that specifies the substring to return from the target string. Substrings are separated by a *delimiter*, 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 *count* value of 0, a negative number, the null string, or a non-numeric string is the same as *count*=1.
range	*Optional* — An expression that resolves to an integer specifying the number of delimited substrings to return, starting with *count*. 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.

Emulation

By default Caché MVBasic permits only a single-character *delimiter*. jBASE emulation permits a multi-character *delimiter*. This option is set using the FULL.DELIM option.

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
```

The following example uses *count* and *range*:

```
colors="Red^Green^Blue^Yellow^Orange^Black"
PRINT GROUP(colors,"^",2,3)
```

Returns “Green^Blue^Yellow”.

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

dynarray	An expression that resolves to a 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 converts numbers to canonical form, resolving multiple signs and removing leading and trailing zeros from element values before making the comparison. If an element value is a missing element, a null string, or a non-numeric value, **GTS** assigns it a value of 0 for the purpose of this comparison.

If the two dynamic arrays have different numbers of elements, the returned dynamic array has the number of elements of the longer dynamic array. By default, the shorter dynamic array is padded with 0 value elements for the purpose of comparison. You can also use the REUSE function to define behavior when specifying two dynamic arrays with different numbers of elements.

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*. It also performs a **GES** (greater than or equal) comparison on the same dynamic arrays:

```
a=10:@VM:-22:@VM:-33:@VM:45
b=10:@VM:-23:@VM:0:@VM:44
PRINT GTS(a,b)
      ! returns 0ý1ý0ý1
PRINT GES(a,b)
      ! returns 1ý1ý0ý1
```

The following example compares dynamic arrays of different lengths. Note that missing elements within the dynamic array (@VM:@VM) are compared, but unmatched elements from the longer array always return 1:

```
a=11:@VM:21:@VM:30:@VM:@VM:51
b=10:@VM:@VM:30:@VM:40:@VM:50:@VM:60:@VM:70
PRINT GTS(a,b)
      ! returns 1ý1ý0ý0ý1ý1ý1
PRINT GTS(b,a)
      ! returns 0ý0ý1ý1ý0ý1ý1
```

See Also

- EQS function
- GES function
- LES function
- LTS function

- Dynamic Arrays

ICONV

Converts a value from external format to internal format.

```
ICONV(ostring,code)
```

Arguments

ostring	An expression that resolves to a string or an integer. It specifies a value in external (output) format.
code	An expression that resolves to a conversion code string. This conversion code specifies the type of conversion to perform. Conversion is from external format to internal format. Conversion codes are not case-sensitive. For a complete list of conversion codes, refer to the Conversion Codes table in the *MultiValue Basic Quick Reference*.

Description

The **ICONV** function is a general-purpose conversion function used to convert from external (output) format to internal (storage) format. The type of conversion is specified by a *code* string that is specific to the type of data to be converted. These conversion codes are as follows:

Time conversion: Internal times are stored as the number of seconds elapsed since midnight. **ICONV** accepts fractional seconds. You can specify the time in 24-hour format (22:45:33) or 12-hour format with the AM/PM suffixes specified for your locale (10:45:33PM) . These suffixes are not case-sensitive. In 12-hour format, you can specify noon or midnight as either 0 or 12. You can specify a number of hours greater than 23 (27:45:33) and **ICONV** calculates the total number of elapsed seconds; you cannot specify a number of minutes or seconds greater than 59. You can use any non-numeric character as a time delimiter (22.45.33); however, you cannot use a period (.) for both the time delimiter and to indicate fractional seconds. You can prefix a time value with a minus sign; **ICONV** prefixes the returned internal time with a minus sign.	"MT"

Date conversion: Takes as input date strings in both numeric formats (9/22/2010, 2010–09–22) and text formats (Oct 22, 2010). If you omit the year portion of the date, it defaults to the current year. If you input a six-digit numeric string without separators, the string is parsed as YYMMDD. These external dates are converted to an internal date integer. 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. The earliest permitted date is December 31, 1840 which is represented internally as -46385. 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*; by default, all two-digit years are expanded to years in the range 1900 through 1999. "DI" can be used to convert dates in numeric format or dates in text format using the NLS locale settings. For example “Feb 22, 2010” or “22 FEBRUARY 2010”. **OCONV** and implicit conversion can convert numeric dates using “DI”, but cannot convert text format dates.	"D" / "DI"
Letter case conversion: “MCL” converts all letters to lower case. “MCU” converts all letters to upper case. You can also use **OCONV** to perform letter case conversions.	"MCL" / "MCU"
Mask Character Alphabetic: converts *ostring* by removing all non-alphabetic characters, returning only the alphabetic characters.	“MCA”
Mask Character Both Alphabetic and Numeric: converts *ostring* by removing all punctuation characters, returning only the alphabetic and numeric characters.	“MCB”
Mask Character Numeric: converts *ostring* by removing all non-numeric characters, returning only the number characters 0 through 9. (Note that plus and minus signs and the decimal point are removed.)	“MCN”
Decimal-to-hexadecimal conversion.	"MCXD"
Hexadecimal-to-decimal conversion.	"MCDX"
Character-to-code conversion: all characters in the input string are converted to their corresponding hexadecimal integer codes. Use “MCXA” or "MCX" to output 8-bit code values; use “MCXW” to output 16-bit (wide) code values.	"MCXA" / "MCX" or "MCXW"

Code-to-character conversion: one or more hexadecimal codes in the input string are converted to their corresponding characters. Use “MCAX” or "MX" for two-byte code input values; use “MCWX” for four-byte (wide) code input values.	"MCAX" / "MX" or "MCWX"
Masked decimal conversion: Converts a fractional number to an integer by moving the decimal point to the right, rounding remaining fractional digits, and resolving numeric display characters such as currency symbols. The returned value is either a positive integer or a negative integer with a leading minus sign. If only *n* is specified, this positive integer value shifts the decimal point that number of places to the right. Any remaining fractional digits are rounded to the resulting integer value. Right side zero padding is added as needed. If both *n* and *m* are specified, it is the *m* value that specifies the number of places to shift the decimal place. (In fact, the *m* value always specifies the decimal shift; when *m* is omitted it defaults to the *n* value.) Masked decimal conversion automatically removes the following formatting characters: leading zeros; a single leading or trailing + sign; a leading $ currency symbol; commas used as numeric group separators (any comma appearing between two number characters); a trailing decimal point. Masked decimal conversion automatically handles negative numbers as follows: a single leading minus sign is retained; a single trailing minus sign converts to a leading minus sign; a positive number enclosed in parentheses converts to a negative number with a leading minus sign. Note that leading sign characters must appear *after* the currency symbol. An *ostring* containing any other characters or character sequences always returns as 0. An empty string *ostring* value returns an empty string.	"MDn[m]" "MLn[m]" "MRn[m]"
Group (delimited substring) extraction: a substring is extracted from *istring*, based on a specified delimiter character (*d*) found in *istring* that indicates the stopping point. This delimiter cannot be a number character or a dynamic array level delimiter character. The optional *s* integer specifies the number of delimiters to skip from the beginning of the string before starting the extract. The default is to start at the beginning of the string. The *n* integer specifies the number of delimiters to count in performing the extract. If *n* is larger than the number of *d* delimiters, the extract continues to the end of the string.	"G[*s*]*dn*"

Length conversion: "L" or "L0": returns the number of characters in *istring*. "Ln": returns the value of *istring* if *n* is the exact number of characters in *istring*. Otherwise returns the empty string. *n* must be a positive non-zero integer. "Ln-m" or "Ln,m": returns the value of *istring* if the number of characters in *istring* is in the range *n* through *m* (inclusive). Otherwise returns the empty string. *n* can be specified as zero in this syntax.	"L" or "L0" "Ln" "Ln-m" or "Ln,m"

The same *code* values can be used in an `<<...>>` inline prompt, to validate (not convert) an interactive input value. Inline prompts can be used in MVBasic statements or MultiValue command line commands. They are described in the *Caché MultiValue Commands Reference*.

The **DATE** and **TIME** functions return internal format values. The **TIMEDATE** function returns external format values.

Note: You can specify the default date and time formats using Caché NLS. Because of operational differences between MV and Caché NLS in the handling of month names, your NLS default date format must represent months as integers.

- If a *code* value is not valid, **ICONV** returns *ostring* unchanged.
- If *ostring* is the empty string (''), **ICONV** returns the empty string for all *code* values except “L” (length). `ICONV('','L')` returns 0.
- If a date *ostring* is not valid, **ICONV** returns the empty string.
- If a time *ostring* is not valid, **ICONV** returns *ostring* unchanged.

You can use the **STATUS** function to determine the success of an **ICONV** conversion. The following status values are supported: 0=successful conversion; 1=invalid *ostring*; 2=invalid *code* value.

The **ICONV** function converts from external format to internal format. The **OCONV** function converts from internal format to external format. Note that the MCDX/MCXD, MCAX/MCXA, and MCWX/MCXW *code* pairs have the opposite meanings in **OCONV**, reversing the **ICONV** operation.

You can use the **ICONVS** function to convert the elements of a dynamic array from external format to internal format.

Examples

The following example 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
```

The following example shows decimal-to-hex and hex-to-decimal conversions. It shows both the **ICONV** conversions and the inverse **OCONV** conversions:

```
HexConversions:
  PRINT ICONV(10,"MCXD");     ! Returns A
  PRINT ICONV(10,"MCDX");     ! Returns 16

  PRINT OCONV(10,"MCXD");     ! Returns 16
  PRINT OCONV(10,"MCDX");     ! Returns A
```

The following example shows character-to-hexcode and hexcode-to-character conversions. It shows both the **ICONV** conversions and the inverse **OCONV** conversions:

```
CharConversions:
  PRINT ICONV("mnop","MCXA");     ! Returns 6D6E6F70
  PRINT ICONV("6D6E6F70","MCAX"); ! Returns mnop

  PRINT OCONV("mnop","MCAX");     ! Returns 6D6E6F70
  PRINT OCONV("6D6E6F70","MCXA"); ! Returns mnop
```

The following example shows masked decimal conversions with rounding and zero padding:

```
MaskedDecimalConversions:
  PRINT ICONV("123.57","MD0");   ! Returns 124      n=0 m=n no decimal shift, rounding
  PRINT ICONV("123.57","MD1");   ! Returns 1236     1 decimal shift, rounding
  PRINT ICONV("123.57","MD2");   ! Returns 12357    2 decimal shift
  PRINT ICONV("123.57","MD3");   ! Returns 123570   3 decimal shift, zero padding
  PRINT ICONV("123.57","MD25");  ! Returns 12357000 n ignored, m=5 decimal shift
  PRINT ICONV("123.57","MD35");  ! Returns 12357000
```

The following example shows masked decimal conversions with valid sign characters:

```
MaskedDecimalConversions:
  PRINT ICONV("123.57","MD2");    ! Returns 12357
  PRINT ICONV("+123.57","MD2");   ! Returns 12357
  PRINT ICONV("-123.57","MD2");   ! Returns -12357
  PRINT ICONV("123.57-","MD2");   ! Returns -12357
  PRINT ICONV("(123.57)","MD2");  ! Returns -12357
  PRINT ICONV("$123.57","MD2");   ! Returns 12357
  PRINT ICONV("$(123.57)","MD2"); ! Returns -12357
```

Multiple sign characters and other non-numeric characters are invalid and return 0.

Emulation

In Caché MultiValue and most emulations, an invalid *ostring* date value (with *code* “D” or “DI”) returns the empty string. An invalid *ostring* numeric value (with *code* “ML”, “MR”, “MD”) returns 0. In UniData emulation an invalid value for these date and numeric codes returns the *ostring* value unchanged.

In Caché MVBasic and most emulations, a six-digit numeric string without separators is parsed as YYMMDD. Thus, the following are equivalent:

```
  PRINT ICONV(861117,"D")
  PRINT ICONV("11/17/86","D")
```

In UniData emulation, a six-digit numeric string without separators is parsed as MMDDYY. Thus, the following are equivalent:

```
  PRINT ICONV(111786,"D")
  PRINT ICONV("11/17/86","D")
```

See Also

- ICONVS function
- OCONV function
- STATUS function
- DATE function
- TIME function

- TIMEDATE function
- Strings

ICONVS

Converts a dynamic array from external format to internal format.

```
ICONVS(odynarray,code)
```

Arguments

odynarray	An expression that resolves to a dynamic array, each element of which specifies a value in external (output) format.
code	An expression that resolves to a conversion code string. This conversion code specifies the type of conversion to perform. Conversion is from external format to internal format. Conversion codes are not case-sensitive. For descriptions of these *code* values, refer to the ICONV function. For a complete list of conversion codes, refer to the Conversion Codes table in the *MultiValue Basic Quick Reference*.

Description

The **ICONVS** function is a general-purpose conversion function used to convert each of the elements of a dynamic array from external (output) format to internal (storage) format. It returns a dynamic array of values. The type of conversion is specified by a *code* string that is specific to the type of data to be converted.

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. Note that the MCDX/MCXD, MCAX/MCXA, and MCWX/MCXW *code* pairs have the opposite meanings in **OCONVS**, reversing the **ICONVS** operation.

If a *code* value is not valid, the *odynarray* is returned unchanged. If a date element value is not valid, the empty string is returned for that element. If a time element value is not valid, the input data value for that element is returned unchanged. If a numeric element value is not valid, zero is returned for that element.

You can use the **ICONV** function to convert a single value from external format to internal format. For further details on conversions, refer to the ICONV function.

Examples

The following example converts dates to a uniform output format. First **ICONVS** performs date conversions from external to internal format, returning a dynamic array of elements containing internal dates. The **OCONVS** performs date conversions from internal to external format, returning all dates in the same default display format:

```
DateConversions:
  x="20 SEP 2006":@VM:"09-21-2006":@VM:"09/22/2006"
  idates=ICONVS(x,"D");
  PRINT idates;   Returns 14143ý14144ý14145
  odates=OCONVS(idates,"D")
  PRINT odates;   Returns 20 SEP 2006ý21 SEP 2006ý22 SEP 2006
```

The following example shows masked decimal conversions with rounding and zero padding:

```
MaskedDecimalConversions:
  x="123.40":@VM:"123.57":@VM:"123.4"
  PRINT ICONVS(x,"MD2")
```

Returns: `12340ý12357ý12340`.

For further examples of format conversion, see ICONV.

See Also

- ICONV function
- OCONVS function
- STATUS function
- DATE function
- TIME function
- TIMEDATE function
- Dynamic Arrays

IFS

Returns a value for each dynamic array element based on the truth value of that element.

```
IFS(dynarray,tdyn,fdyn)
```

Arguments

dynarray	The dynamic array tested for truth values of elements. An expression that resolves to a dynamic array of boolean value elements.
tdyn	An expression that resolves to a dynamic array of replacement values. A *tdyn* element value replaces the corresponding *dynarray* element if the *dynarray* element has a value of True.
fdyn	An expression that resolves to a dynamic array of replacement values. A *fdyn* element value replaces the corresponding *dynarray* element if the *dynarray* element has a value of False.

Description

The **IFS** function returns a dynamic array consisting of element values from the *tdyn* and *fdyn* dynamic arrays. **IFS** evaluates each element of *dynarray* as either true or false. If an element is evaluated as true, the corresponding element from *tdyn* is included in the returned dynamic array. If an element is evaluated as false, the corresponding element from *fdyn* is included in the returned dynamic array.

The **IFS** truth test is as follows:

- 0 (zero) is evaluated as False.
- 1 or any other non-zero numeric is evaluated as True.
- a non-numeric value is evaluated as False.
- an empty string or absent element value is evaluated as False.

IFS does not perform arithmetic evaluation of expressions. For example, the element values “7=7” and “7=8” would both evaluate as True, because they both are non-zero numerics. Similarly, “5–5” or “5*0” evaluate as True.

IFS only recognizes numeric values as logically true or false. It does not recognize “T” or “F” (or any other alphabetic string) as having a logical value. All alphabetic strings evaluate to False.

IFS must have a corresponding *tdyn* or *fdyn* element value (or, preferably, both) in order to evaluate an element.

If *dynarray* contains more than one element, **IFS** returns the number of elements in *dynarray*, if either *tdyn* or *fdyn* contain at least that many elements. Excess *tdyn* or *fdyn* elements are ignored. If either *tdyn* or *fdyn* does not contain enough elements, an empty string element is supplied. If both *tdyn* and *fdyn* do not contain enough elements,**IFS** returns a dynamic array containing the number of elements in the longer of *tdyn* or *fdyn*.

Single Elements in dynarray

If *dynarray* is a numeric or string value, it is treated as a dynamic array with one element. If *dynarray* is a single data value (True or False), the returned dynamic array consists of either all of the *tdyn* true values or all of the *fdyn* false values, depending on the true or false value of *dynarray*. This is shown in the following example:

```
booldyn=1
tdyn="1":@VM:"2":@VM:"3":@VM:"4":@VM:"5"
fdyn="missing1":@VM:"missing2":@VM:"missing3":@VM:"missing4":@VM:"missing5"
PRINT IFS(booldyn,tdyn,fdyn)
```

This program returns the dynamic array `1ý2ý3ý4ý5`.

If *dynarray* contains a single field element value, with no value element or subvalue element values, the returned dynamic array element for that field consists of either all *tdyn* or all *fdyn* values for that field and all of its value and subvalue elements. This is shown in the following example:

```
booldyn=1:@FM:0:@FM
tdyn="1":@VM:"one":@VM:"uno":@FM:"2":@VM:"two":@VM:"due":@FM:"3":@VM:"three":@VM:"tre"
fdyn="A":@VM:"alpha":@VM:"alef":@FM:"B":@VM:"beta":@VM:"bet":@FM:"G":@VM:"gamma":@VM:"gimel"
PRINT IFS(booldyn,tdyn,fdyn)
```

This program returns the dynamic array `1ýoneýunoþBýbetaýbetþ3ýthreeýtre`

If *dynarray* contains a single value element value with no subvalue element values, the returned dynamic array element for that value element consists of either all *tdyn* or all *fdyn* values for that value element and all of its subvalue elements.

Examples

The following example uses the **IFS** function to return a dynamic array based on truth values:

```
booldyn="0":@VM:"1":@VM:"1":@VM:"0":@VM:"1"
tdyn="1":@VM:"2":@VM:"3":@VM:"4":@VM:"5"
fdyn="missing1":@VM:"missing2":@VM:"missing3":@VM:"missing4":@VM:"missing5"
PRINT IFS(booldyn,tdyn,fdyn)
```

This program returns the dynamic array `missing1ý2ý3ýmissing4ý5`.

The following example shows that *tdyn* and *fdyn* may have more elements than *dynarray*:

```
booldyn="0":@VM:"1":@VM:"1"
tdyn="1":@VM:"2":@VM:"3":@VM:"4":@VM:"5"
fdyn="missing1":@VM:"missing2":@VM:"missing3":@VM:"missing4":@VM:"missing5"
PRINT IFS(booldyn,tdyn,fdyn)
```

This program returns the dynamic array `missing1ý2ý3`.

The following examples show what happens when *tdyn* or *fdyn* have fewer elements than *dynarray*:

```
booldyn="0":@VM:"1":@VM:"1":@VM:"0":@VM:"1"
tdyn="1":@VM:"2":@VM:"3":@VM:"4":@VM:"5"
fdyn="missing1":@VM:"missing2":@VM:"missing3":@VM:"missing4"
PRINT IFS(booldyn,tdyn,fdyn)
```

This program returns the dynamic array `missing1ý2ý3ýmissing4ý`. Note that the returned dynamic array does not specify a value for the fifth element.

```
booldyn="0":@VM:"1":@VM:"1":@VM:"0":@VM:"1"
tdyn="1":@VM:"2":@VM:"3":@VM:"4"
fdyn="missing1":@VM:"missing2":@VM:"missing3":@VM:"missing4":@VM:"missing5"
PRINT IFS(booldyn,tdyn,fdyn)
```

This program returns the dynamic array `missing1ý2ý3ýmissing4ýmissing5`.

See Also

- Dynamic Arrays

INDEX

Returns starting position of a substring in a string.

```
INDEX(string,substring,occurs)
```

Arguments

string	The string to be searched for *substring*. An expression that resolves to a string or a numeric value.
substring	An expression that resolves to a substring to locate within *string*.
occurs	An expression that resolves to a positive integer specifying which occurrence of *substring* 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 *occurs* specifies more occurrences of *substring* than appear in *string*.
- If *occurs* is 0, a negative number, the null string, or a non-numeric string.
- If *string* is the null string ("") and *substring* is not.

If *substring* is the null string (""), **INDEX** returns 1. If both *string* and *substring* are the null string (""), **INDEX** returns 1.

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

dynarray	A dynamic array of elements to be searched for *substring*. An expression that resolves to a dynamic array.
substring	An expression that resolves to a substring to locate within *string*.
occurs	An expression that resolves to a positive integer specifying which occurrence of *substring* 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, or a non-numeric string.

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 indices.

```
INDICES(filevar[,indexname])
```

Arguments

filevar	A local variable name assigned to a MultiValue file by the OPEN command when the file was opened.
indexname	*Optional* — An expression that resolves to the name of a secondary index in the file referenced by *filevar*. You can specify this either as the actual index name or as the MVName (the name of the index as stored in the MultiValue dictionary).

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 indices in the file referenced by *filevar*. 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).

The returned dynamic array consists of six fields. These fields can contain multiple values, as follows:

Field 1	Value 1: the index type code value. Possible values include A, D, I, and S (from the MVTYPE parameter). Value 2: always 0. Value 3: unused. Value 4: always 1. Value 5: the path name. Value 6: unused. Value 7: justification code value, R (right justified) or L (left justified). Value 8: uniqueness code value, Y (yes) or N (no). Values 9 through 16: unused. Value 17: the collation name.
Field 2	Property specifications. For type I, this is the MVITYPE. For types A, D, or S, this is the MVTOLOGICAL value (if it exists), otherwise the MVATTRIBUTE (if it exists), otherwise the empty string.
Field 3	Unused.
Field 4	Unused.
Field 5	Unused.
Field 6	MultiValue code value, M (multi-valued) or S (single-valued).
Field 7	Unused.
Field 8	The property name.
Field 9	The MVName (the name of the index as stored in the MultiValue dictionary).

See Also

- OPEN statement
- Dynamic Arrays

INMAT

Returns the number of array elements.

```
INMAT(array)
```

Arguments

array	*Optional* — The name of a dimensioned array; must be a literal name, not enclosed with quotes. If omitted, the most recently parsed dimensioned array.

Description

The **INMAT** function has two uses:

- **INMAT(array)** returns the defined dimensions of the named array.
- **INMAT()** returns the number of elements loaded into the most recently parsed array.

Before invoking **INMAT**, an array must have been dimensioned using the **DIM** (or **DIMENSION**) statement.

INMAT(array) takes the name of a dimensioned array and returns a two-element dynamic array containing the *rows* and *columns* dimensions of the array. If the **DIM** did not specify a *columns* dimension, **INMAT** returns 1 as the *columns* dimension.

Specifying an *array* value that is not a dimensioned array results in a #1039 compile error. The *array* must be an explicitly dimensioned static array. Specifying an implicitly dimensioned (dynamic) array results in a runtime <ARRAY DIMENSION> error.

INMAT() defaults to the most recently loaded array. It returns an integer specifying the number of elements that were loaded using a **MATREAD**, **MATREADU**, or **MATPARSE** statement. If no array has been loaded, or the most recently loaded array was loaded with the null string or with a single element, **INMAT()** returns 1.

Examples

The following example returns the row and column dimensions of a dimensioned array:

```
DIM MyArray(6,5)
CRT INMAT(MyArray)
    ! Returns 6ý5
```

The following example returns the number of dimensioned array elements parsed by **MATPARSE**:

```
DIM MyArray(6,5)
x="Fred":@FM:"Barney":@FM:"Wilma"
MATPARSE MyArray FROM x
CRT INMAT()
    ! Returns 3
```

See Also

- DIM statement
- MATPARSE statement
- MATREAD statement

INSERT

Inserts data in a dynamic array.

```
INSERT(dynarray,f[,v[,s]];expression)
```

Arguments

dynarray	An expression that resolves to the name of a dynamic array.
f	An expression that resolves to an integer specifying the Field level of the dynamic array in which to insert the data. Fields are counted from 1.
v	*Optional* — An expression that resolves to an integer specifying the Value level of the dynamic array in which to insert the data. Values are counted from 1 within a Field.
s	*Optional* — An expression that resolves to an integer specifying the Subvalue level of the dynamic array in which to insert the data. Subvalues are counted from 1 within a Value.
;*expression*	The data to be inserted. An expression that resolves to a string.

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. **INSERT** adds multiple delimiter characters, when needed, to place the data value at the specified location.

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.

To insert a value at the beginning of a *dynarray* set *f* to 1 or 0. To insert a value at the end of a *dynarray* set *f* to -1. If lower level delimiters exist in *dynarray*, setting an upper level to 0, the null string, or a non-numeric value is equivalent to setting it to 1.

Both the **INSERT** function and the **INS** command insert a value into a dynamic array. The **INSERT** function returns a dynamic array containing the insert; it does not change the value of the supplied *dynarray*. The **INS** command changes the value of the supplied *dynarray*.

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 YorkýProvidenceýLondonýChicagoýBostonýLos Angeles"
```

Emulation

UniData systems differ in how they handle *f*, *v*, and *s* arguments set to 0. The $OPTIONS ATTR.0IS1 (“zero is one”) provides support for this UniData feature. UniData systems ignore *v* and *s* arguments that are set to a negative number.

See Also

- INS statement
- COUNTS function

- DELETE function
- EXTRACT function
- FIELDSTORE function
- Dynamic Arrays

INT

Returns the integer component of an expression.

```
INT(string)
```

Arguments

string	An expression that resolves to a number or numeric string.

Description

The **INT** function returns the integer portion of a numeric or numeric string. It returns both positive and negative integers. Fractional numbers are truncated (not rounded) to an integer. A zero, an empty string (""), or a non-numeric string all return the integer 0.

For numbers and numeric strings, prior to determining the integer, 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.

Examples

The following examples all use the **INT** function to return the integer 321:

```
PRINT INT(321.37);        ! Returns 321
PRINT INT(321.99);        ! Returns 321
PRINT INT("--00321.6");   ! Returns 321
PRINT INT("321blastoff"); ! Returns 321
```

See Also

- FIX function
- FMT function
- PRECISION command

ISOBJECT

Returns whether or not a something is a Caché object.

```
ISOBJECT(name)
```

Arguments

name	Any valid expression.

Description

The **ISOBJECT** function returns a boolean value indicating whether the specified *name* corresponds to an existing Caché Object. Returns 1 if *name* is an object; otherwise returns 0.

See Also

- EXISTS function

ITYPE

Returns the I-type value from the file dictionary.

```
ITYPE(itype)
```

Arguments

itype	An expression that resolves to the contents of the I-descriptor.

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). For further details, refer to the list of System Variables.

An invalid value for *itype* generates a <SUBSCRIPT> error.

See Also

- READ statement

KEYIN

Receives a single character of user input.

```
KEYIN([timeout])
```

Arguments

timeout	*Optional* — An expression that resolves to an integer specifying the number of seconds to wait for user input before timing out.

Note that 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. It displays a prompt to receive a single input character. The appearance of this prompt is governed by the terminal emulator, not MVBasic. Program execution continues immediately upon input of a character. No Enter key is required.

In Caché MultiValue the input character is not echoed, regardless of the setting of ECHO. However, echoing is emulation-dependent.

If you specify a *timeout* value, **KEYIN** waits the specified number of seconds for user input. Upon time out it returns an empty string ("").

You can also use the **IN** function to receive a single character of user input, or the **INPUT** statement to receive one or more characters of user input. You can use the `<<...>>` inline prompt to prompt for a user input value to insert in a MVBasic statement or a MultiValue command line command. The `<<...>>` inline prompt is described in the *Caché MultiValue Commands Reference*.

KEYIN does not accept stacked input data. 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
```

Emulation

In Caché, UniVerse, and several other emulations, the input character is never echoed, regardless of the setting of ECHO.

In D3, IN2, jBASE, MVBase, R83, POWER95, Reality, and Ultimate emulations, whether the input character is echoed depends on the setting of ECHO. In these emulations, when you specify ECHO ON the input character is echoed to the terminal; it is never echoed to the printer.

See Also

- IN statement
- INPUT statement

LEFT

Returns a specified number of characters from the left end of a string.

```
LEFT(string,length)
```

Arguments

string	An expression that resolves to a string from which the leftmost characters are returned.
length	An expression that resolves 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

string	An expression that resolves to a string or numeric expression.

Description

The **LEN** function returns the number of characters in a specified string. **LEN** counts characters, not bytes. You can use the **BYTELEN** function to count the number of bytes in a string.

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.

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

- BYTELEN function
- COUNT function
- LENS function

LENS

Returns the length of each element of a dynamic array.

```
LENS(dynarray)
```

Arguments

dynarray	An expression that resolves to a 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 ("") or a missing element 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 3ý4ý3ý7ý0
```

The following example show how **LENS** handles missing dynamic array elements:

```
nums=123:@VM:456:@VM:@VM:789
PRINT LENS(nums);  ! Returns 3ý3ý0ý3
```

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

dynarray	An expression that resolves to a 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 is missing, or has a null string or a non-numeric value, **LES** assigns it a value of 0 for the purpose of this comparison.

If the two dynamic arrays have different numbers of elements, the returned dynamic array has the number of elements of the longer dynamic array. By default, the shorter dynamic array is padded with 0 value elements for the purpose of comparison. You can also use the REUSE function to define behavior when specifying two dynamic arrays with different numbers of elements.

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 1ý0ý1ý0
```

See Also

- EQS function
- GES function
- GTS function
- LTS function
- Dynamic Arrays

$LIST ($LI)

Returns elements in a list.

```
$LIST(list,position,end)
$LI(list,position,end)
```

Parameters

list	An expression that resolves to a Caché list. Because lists contain encoding, *list* must be created using $LISTBUILD or $LISTFROMSTRING, or extracted from another list using **$LIST**.
position	*Optional* — The starting position in the specified list. An expression that resolves to an integer. Valid values are -1 and positive integers.
end	*Optional* — The ending position in the specified list. An expression that resolves to an integer. Valid values are -1, 0, and positive integers.

Description

$LIST returns elements from a list. The elements returned depend on the parameters used.

- **$LIST**(*list*) returns the first element string in the list.
- **$LIST**(*list*,*position*) returns the element indicated by the specified position. The *position* parameter must evaluate to an integer.
- **$LIST**(*list*,*position*,*end*) returns a "sublist" (an encoded list string) containing the elements of the list from the specified start *position* through the specified *end* position.

Parameters

list

An encoded list string containing one or more elements. Lists can be created using **$LISTBUILD** or **$LISTFROMSTRING**, or extracted from another list by using the **$LIST** function. The following are valid *list* arguments:

```
myList = $LISTBUILD("Red","Blue","Green","Yellow")
PRINT $LIST(myList,2);    ! prints Blue
subList = $LIST(myList,2,4)
PRINT $LIST(subList,2);   ! prints Green
```

In the following example, *subList* is not a valid *list* argument, because it is a single element returned as an ordinary string, not an encoded list string:

```
myList = $LISTBUILD("Red","Blue","Green","Yellow")
subList = $LIST(myList,2)
PRINT $LIST(subList,1)
```

The second **$LIST** generates a <LIST> error.

position

The position of a list element to return. List elements are counted from 1. If *position* is omitted, the first element is returned. If the value of *position* is 0 or greater than the number of elements in the list, Caché issues a <NULL VALUE> error. If the value of *position* is negative one (–1), **$LIST** returns the final element in the list.

If the *end* parameter is specified, *position* specifies the first element in a range of elements. Even when only one element is returned (when *position* and *end* are the same number) this value is returned as an encoded list string. Thus, $LIST(x,2) is not identical to $LIST(x,2,2).

end

The position of the last element in a range of elements. You must specify *position* to specify *end*. When *end* is specified, the value returned is an encoded list string. Because of this encoding, such strings should only be processed by other $LIST functions.

If the value of *end* is:

- greater than *position*, an encoded string containing a list of elements is returned.
- equal to *position*, an encoded string containing the one element is returned.
- less than *position*, the null string ("") is returned.
- greater than the number of elements in *list*, it is equivalent to specifying the final element in the list.
- negative one (–1), it is equivalent to specifying the final element in the list.

When specifying *end*, you can specify a *position* value of zero (0). In this case, 0 is equivalent to 1.

Examples

The following two **PRINT** statements both return “RED”, the first element in the list. The first writes the first element by default, the second writes the first element because the *position* parameter is set to 1:

```
PRINT $LIST($LISTBUILD("RED","BLUE","GREEN"))
PRINT $LIST($LISTBUILD("RED","BLUE","GREEN"),1)
```

The following example returns “Blue”, the second element in the list:

```
x=$LISTBUILD("Red","Blue","Green")
PRINT $LIST(x,2)
```

The following example returns “Green White”, a two-element list string beginning with the first element and ending with the second element in the list.

```
x=$LISTBUILD("Green ","White ","Brown ","Black ")
PRINT $LIST(x,1,2)
```

The following example returns “Brown Black”, a two-element list string that begins with the third element and ends with the last element in the list:

```
x=$LISTBUILD("Green ","White ","Brown ","Black ")
PRINT $LIST(x,3,-1)
```

Notes

Invalid Parameter Values

If the expression in the *list* parameter does not evaluate to a valid list, a <LIST> error occurs.

```
x=CHAR(0):CHAR(0):CHAR(0):CHAR(1):CHAR(16):CHAR(27):CHAR(134):CHAR(240)
a=$LIST(x,2);    ! generates a LIST error
```

If the value of the *position* parameter or the *end* parameter is less than -1, invoking the **$LIST** function generates a <RANGE> error.

If the value of the *position* parameter refers to a nonexistent list member and no *end* parameter is used, invoking the **$LIST** function generates a <NULL VALUE> error.

```
list2=$LISTBUILD("Brown","Black")
PRINT $LIST(list2,3);    ! generates a NULL VALUE error
x=$LIST(list2,0);        ! generates a NULL VALUE error
```

If the value of the *position* parameter identifies an element with an undefined value, invoking the **$LIST** function also generates a <NULL VALUE> error.

```
PRINT $LIST("");    ! generates a NULL VALUE error
x=$LISTBUILD("A",,"C")
PRINT $LIST(x,2);   ! generates a NULL VALUE error
```

Two-Parameter and Three-Parameter $LIST

$LIST(*list*,1) is not equivalent to **$LIST**(*list*,1,1) because the former returns a string, while the latter returns a single-element list string. Furthermore, the first can receive a <NULL VALUE> error, whereas the second cannot; if there are no elements to return, it returns a null string.

Unicode

If one Unicode character appears in a list element, that entire list element is represented as Unicode (wide) characters. Other elements in the list are not affected.

The following example shows two lists. The *y* list consists of two elements which contain only ASCII characters. The *z* list consists of two elements: the first element contains a Unicode character (CHAR(960) = the pi symbol); the second element contains only ASCII characters.

```
y=$LISTBUILD("ABC":CHAR(68),"XYZ")
z=$LISTBUILD("ABC":CHAR(960),"XYZ")
PRINT "The ASCII list y elements: "
PRINT $LIST(y,1)
PRINT $LIST(y,2)
PRINT "The Unicode list z elements: "
PRINT $LIST(z,1)
PRINT $LIST(z,2)
```

Note that Caché encodes the first element of *z* entirely in wide Unicode characters. The second element of *z* contains no Unicode characters, and thus Caché encodes it using narrow ASCII characters.

See Also

- $LISTBUILD function
- $LISTDATA function
- $LISTFIND function
- $LISTFROMSTRING function
- $LISTGET function
- $LISTLENGTH function
- $LISTNEXT function
- $LISTSAME function
- $LISTTOSTRING function
- $LISTVALID function

$LISTBUILD ($LB)

Builds a list of elements from the specified expressions.

```
$LISTBUILD(element,...) $LB(element,...)
```

Parameter

element	Element values to include in a Caché list. An expression that resolves to a number or string. Commonly, multiple *element* values are specified, separated by commas. To include a comma within an *element*, make the *element* a quoted string.

Description

$LISTBUILD takes one or more *element* values and returns a Caché list structure containing a corresponding list of elements.

The following functions can be used to create a list:

- **$LISTBUILD**, which creates a list from multiple strings, one string per element.
- **$LISTFROMSTRING**, which creates a list from a single string containing multiple delimited elements.
- **$LIST**, which extracts a sublist from an existing list.

$LISTBUILD is used with the other $LIST functions: **$LISTDATA**, **$LISTFIND**, **$LISTGET**, **$LISTLENGTH**, **$LISTSAME**, and **$LISTTOSTRING**.

Note: **$LISTBUILD** and the other $LIST functions use an optimized binary representation to store data elements. For this reason, equivalency tests may not work as expected with some $LIST data. Data that might, in other contexts, be considered equivalent, may have a different internal representation. For example, **$LISTBUILD(1)** is not equal to **$LISTBUILD(“1”)**.

For the same reason, a list string value returned by **$LISTBUILD** should not be used in character search and parse functions that use a delimiter character, such as **$PIECE** and the two-argument form of **$LENGTH**. Elements in a list created by **$LISTBUILD** are not marked by a character delimiter, and thus can contain any character.

Examples

The following example produces the three-element list "Red,Blue,Green":

```
x=$LISTBUILD("Red","Blue","Green")
PRINT $LIST(x,1,3)
```

Notes

Omitting Parameters

Omitting an element expression yields an element whose value is undefined. For example, the following **$LISTBUILD** statement produces a three-element list whose second element has an undefined value; referencing the second element with the **$LIST** function will produce a <NULL VALUE> error.

```
PRINT $LIST($LISTBUILD("Red",,"Green"),2)
```

However, the following produces a three-element list whose second element is a null string. No error condition exists.

```
PRINT $LIST($LISTBUILD("Red","","Green"),2)
```

Additionally, if a **$LISTBUILD** expression is undefined, the corresponding list element has an undefined value. The following two expressions both produce the same two-element list whose first element is "Red" and whose second element has an undefined value:

```
PRINT $LISTBUILD("Red",)
PRINT $LISTBUILD("Red",Z)
```

Providing No Parameters

Invoking the **$LISTBUILD** function with no parameters returns a list with one element whose data value is undefined. This is not the same as a null string:

```
x=$LISTBUILD()
y=$LISTBUILD("")
PRINT x
PRINT y
```

Nesting Lists

An element of a list may itself be a list. For example, the following statement produces a three-element list whose third element is the two-element list, "Walnut,Pecan":

```
x=$LISTBUILD("Apple","Pear",$LISTBUILD("Walnut","Pecan"))
PRINT $LIST(x,3)
```

Concatenating Lists

The result of concatenating two lists with the Concatenation Operator (:) is another list. For example, the following two **WRITE** statements produce the same list, "A,B,C":

```
PRINT $LIST($LISTBUILD("A","B"):$LISTBUILD("C"),0,-1)
PRINT $LIST($LISTBUILD("A","B","C"),0,-1)
```

A null string ("") is an empty list. For example, the following two expressions each produce the same two-element list:

```
PRINT $LISTBUILD("A","B"):""
PRINT $LISTBUILD("A","B")
```

However, the following two expressions each produce a three-element list:

```
PRINT $LISTBUILD("A","B"):$LISTBUILD("")
PRINT $LISTBUILD("A","B"):$LISTBUILD()
```

Unicode

If one or more characters in a list element is a wide (Unicode) character, all characters in that element are represented as wide characters. To ensure compatibility across systems, **$LISTBUILD** always stores these bytes in the same order, regardless of the hardware platform. Wide characters are represented as byte strings. Therefore, the length reflects the number of bytes, not the number of Unicode characters.

In the following example, the first element has a length of ten: four characters, each two bytes long, plus the length and data type bytes. The data type byte is 02 (Unicode string). Each character is represented by two bytes in little-endian order. The second element has a length of five: three characters, plus the length and data type bytes. The data type byte is 01 (Binary string).

```
z=$LISTBUILD(CHAR(987):"ABC","ABC")
PRINT LEN(z)
```

See Also

- $LIST function
- $LISTDATA function

- $LISTFIND function
- $LISTFROMSTRING function
- $LISTGET function
- $LISTLENGTH function
- $LISTNEXT function
- $LISTSAME function
- $LISTTOSTRING function
- $LISTVALID function

$LISTDATA ($LD)

Indicates whether the specified element exists and has a data value.

```
$LISTDATA(list,position)
$LD(list,position)
```

Parameters

list	An expression that resolves to a Caché list. A Caché list is created using $LISTBUILD or $LISTFROMSTRING, or extracted from another list using $LIST.
position	*Optional* — An expression that resolves to an integer specifying a position in *list*. Valid values are -1, 0, and positive integers.

Description

$LISTDATA checks for data in the requested element in a list and returns a boolean value. **$LISTDATA** returns a value of 1 if the element indicated by the *position* parameter is in the *list* and has a data value. **$LISTDATA** returns a value of a 0 if the element is not in the *list* or does not have a data value.

Parameters

list

A *list* is an encoded string containing multiple elements. A *list* must have been created using $LISTBUILD or $LISTFROMSTRING, or extracted from another list using $LIST.

position

If you omit the *position* parameter, **$LISTDATA** evaluates the first element. If the value of the *position* parameter is -1, it is equivalent to specifying the final element of the list. If the value of the *position* parameter refers to a nonexistent list member, then invoking the **$LISTDATA** function returns a 0.

Examples

The following examples show the results of the various values of the *position* parameter.

All of the following **$LISTDATA** statements return a value of 0 (Y is an undefined variable):

```
x=$LISTBUILD("Red",,Y,"","Green")
PRINT $LISTDATA(x,2);     ! second element is undefined
PRINT $LISTDATA(x,3);     ! third element is undefined variable
PRINT $LISTDATA("");      ! null string
PRINT $LISTDATA(x,0);     ! the 0th position
PRINT $LISTDATA(x,6);     ! 6th position in 5-element list
```

The following **$LISTDATA** statements return a value of 1 for the same five-element list:

```
x=$LISTBUILD("Red",,Y,"","Green")
PRINT $LISTDATA(x);    ! first position (by default)
PRINT $LISTDATA(x,1);  ! first position specified
PRINT $LISTDATA(x,4);  ! fourth position value=null string
PRINT $LISTDATA(x,5);  ! fifth position
PRINT $LISTDATA(x,-1); ! last (5th) position
```

Notes

Invalid Parameter Values

If the expression in the *list* parameter does not evaluate to a valid list, a <LIST> error occurs.

If the value of the *position* parameter is less than -1, invoking the **$LISTDATA** function generates a <RANGE> error.

See Also

- $LIST function
- $LISTBUILD function
- $LISTFIND function
- $LISTFROMSTRING function
- $LISTGET function
- $LISTLENGTH function
- $LISTNEXT function
- $LISTSAME function
- $LISTTOSTRING function
- $LISTVALID function

$LISTFIND ($LF)

Searches a specified list for the requested value.

```
$LISTFIND(list,value,startafter)
$LF(list,value,startafter)
```

Parameters

list	An expression that resolves to a Caché list. A Caché list is created using $LISTBUILD or $LISTFROMSTRING, or extracted from another list using $LIST.
value	An expression that resolves to the desired element value.
startafter	*Optional* — An expression that resolves to a positive integer, specifying a position in *list*. The search starts with the element after this position.

Description

$LISTFIND searches the specified *list* for the first instance of the requested *value*. The search begins with the element after the position indicated by the *startafter* parameter. If you omit the *startafter* parameter, **$LISTFIND** assumes a *startafter* value of 0 and starts the search with the first element (element 1). If the value is found, **$LISTFIND** returns the position of the matching element. If the value is not found, **$LISTFIND** returns a 0. The **$LISTFIND** function will also return a 0 if the value of the *startafter* parameter refers to a nonexistent list member.

Examples

The following example returns 2, the position of the first occurrence of the requested string:

```
x=$LISTBUILD("A","B","C","D")
PRINT $LISTFIND(x,"B")
```

The following example returns 0, indicating the requested string was not found:

```
x=$LISTBUILD("A","B","C","D")
PRINT $LISTFIND(x,"E")
```

The following examples show the effect of using the *startafter* parameter. The first example does not find the requested string and returns 0 because it occurs at the *startafter* position:

```
x=$LISTBUILD("A","B","C","D")
PRINT $LISTFIND(x,"B",2)
```

The second example finds the second occurrence of the requested string and returns 4, because the first occurs before the *startafter* position:

```
y=$LISTBUILD("A","B","C","A")
PRINT $LISTFIND(y,"A",2)
```

The **$LISTFIND** function only matches complete elements. Thus, the following example returns 0 because no element of the list is equal to the string “B”, though all of the elements contain “B”:

```
mylist = $LISTBUILD("ABC","BCD","BBB")
PRINT $LISTFIND(mylist,"B")
```

Notes

Invalid Parameter Values

If the expression in the *list* parameter does not evaluate to a valid list, a <LIST> error occurs.

If the value of the *startafter* parameter is less than -1, invoking the **$LISTFIND** function generates a <RANGE> error.

See Also

- $LIST function
- $LISTBUILD function
- $LISTDATA function
- $LISTFROMSTRING function
- $LISTGET function
- $LISTLENGTH function
- $LISTNEXT function
- $LISTSAME function
- $LISTTOSTRING function
- $LISTVALID function

$LISTFROMSTRING ($LFS)

Creates a list from a string.

```
$LISTFROMSTRING(string,delimiter)
$LFS(string,delimiter)
```

Parameters

string	An expression that resolves to a string to be converted into a Caché list. This string contains one or more elements, separated by a *delimiter*. The *delimiter* does not become part of the resulting Caché list. The *string* cannot contain the quoted string delimiter.
delimiter	*Optional* — The delimiter used to separate substrings (elements) in *string*. An expression that resolves to a string containing one or more characters. If no *delimiter* is specified, the default is the comma (,) character. The *delimiter* cannot contain the character used to delimit the quoted string.

Description

$LISTFROMSTRING takes a quoted string containing delimited elements and returns a Caché list. A list represents data in an encoded format which does not use delimiter characters. Thus a list can contain all possible characters, and is ideally suited for bitstring data. Lists are handled using the various **$LIST** functions.

Parameters

string

A string literal (enclosed in quotation marks), a numeric, or a variable or expression that evaluates to a string. This string can contain one or more substrings (elements), separated by a *delimiter*. The string data elements must not contain the *delimiter* character (or string), because the *delimiter* character is not included in the output list.

delimiter

A character (or string of characters) used to delimit substrings within the input string. It can be a numeric or string literal (enclosed in quotation marks), the name of a variable, or an expression that evaluates to a string.

Commonly, a delimiter is a designated character which is never used within string data, but is set aside solely for use as a delimiter separating substrings. A delimiter can also be a multi-character string, the individual characters of which can be used within string data.

If you specify no delimiter, the default delimiter is the comma (,) character. You cannot specify a null string ("") as a delimiter; attempting to do so results in a <STRINGSTACK> error.

Example

The following example takes a string of names which are separated by a blank space, and creates a list:

```
namestring="Deborah Noah Martha Bowie"
namelist=$LISTFROMSTRING(namestring," ")
PRINT "1st element: ",$LIST(namelist,1)
PRINT "2nd element: ",$LIST(namelist,2)
PRINT "3rd element: ",$LIST(namelist,3)
```

See Also

- $LIST function

- $LISTBUILD function
- $LISTDATA function
- $LISTFIND function
- $LISTGET function
- $LISTLENGTH function
- $LISTNEXT function
- $LISTSAME function
- $LISTTOSTRING function
- $LISTVALID function

$LISTGET ($LG)

Returns an element in a list, or a specified default value if the requested element is undefined.

```
$LISTGET(list,position,default)
$LG(list,position,default)
```

Parameters

list	An expression that resolves to a Caché list. A Caché list is created using $LISTBUILD or $LISTFROMSTRING, or extracted from another list using $LIST.
position	*Optional* — An expression that resolves to a integer, specifying a position in *list*. Permitted values are -1, 0, and positive integers.
default	*Optional* — An expression that resolves to a value to return if the list element has an undefined value.

Description

$LISTGET returns the requested element in the specified list. If the value of the position parameter refers to a nonexistent member or identifies an element with an undefined value, the specified default value is returned.

The **$LISTGET** function is identical to the one- and two-argument forms of the **$LIST** function except that, under conditions that would cause **$LIST** to produce a <NULL VALUE> error, **$LISTGET** returns a default value. See the description of the **$LIST** function for more information on conditions that generate <NULL VALUE> errors.

Parameters

position

The *position* parameter must evaluate to an integer. If it is omitted, by default, the function examines the first element of the list. If the value of the *position* parameter is -1, it is equivalent to specifying the last element of the list.

default

If you omit the *default* parameter, the null string ("") is assumed for the default value.

Examples

The **$LISTGET** functions in the following example both return “A”, the first element in the list:

```
list=$LISTBUILD("A","B","C")
PRINT $LISTGET(list)
PRINT $LISTGET(list,1)
```

The **$LISTGET** functions in the following example both return “C”, the third and last element in the list:

```
list=$LISTBUILD("A","B","C")
PRINT $LISTGET(list,3)
PRINT $LISTGET(list,-1)
```

The **$LISTGET** functions in the following example both return a value upon encountering the undefined 2nd element in the list. The first returns a question mark (?), which the user defined as the default value. The second returns a null string because a default value is not specified:

```
PRINT $LISTGET($LISTBUILD("A",,"C"),2,"?")
PRINT $LISTGET($LISTBUILD("A",,"C"),2)
```

The **$LISTGET** functions in the following example both specify a position greater than the last element in the three-element list. The first returns a null string because the default value is not specified. The second returns the user-specified default value, “ERR”:

```
list=$LISTBUILD("A","B","C")
PRINT $LISTGET(list,4)
PRINT $LISTGET(list,4,"ERR")
```

The **$LISTGET** functions in the following example both return a null string:

```
list=$LISTBUILD("A","B","C")
PRINT $LISTGET(list,0)
PRINT $LISTGET("")
```

Notes

Invalid Parameter Values

If the expression in the *list* parameter does not evaluate to a valid list, a <LIST> error can occur.

If the value of the *position* parameter is less than -1, invoking the **$LISTGET** function generates a <RANGE> error.

See Also

- $LIST function
- $LISTBUILD function
- $LISTDATA function
- $LISTFIND function
- $LISTFROMSTRING function
- $LISTLENGTH function
- $LISTNEXT function
- $LISTSAME function
- $LISTTOSTRING function
- $LISTVALID function

$LISTLENGTH ($LL)

Returns the number of elements in a specified list.

```
$LISTLENGTH(list)
$LL(list)
```

Parameter

list	An expression that resolves to a Caché list. A Caché list is created using $LISTBUILD or $LISTFROMSTRING, or extracted from another list using $LIST.

Description

$LISTLENGTH returns the number of elements in *list*.

Examples

The following example returns 3, because there are 3 elements in the list:

```
PRINT $LISTLENGTH($LISTBUILD("Red","Blue","Green"))
```

The following example returns a 0, because a null string is a valid (zero-element) list.

```
PRINT $LISTLENGTH("")
```

Notes

$LISTLENGTH and Nested Lists

The following example returns 3, because **$LISTLENGTH** does not recognize the individual elements in nested lists:

```
x=$LISTBUILD("Apple","Pear",$LISTBUILD("Walnut","Pecan"))
PRINT $LISTLENGTH(x)
```

See Also

- $LIST function
- $LISTBUILD function
- $LISTDATA function
- $LISTFIND function
- $LISTFROMSTRING function
- $LISTGET function
- $LISTNEXT function
- $LISTSAME function
- $LISTTOSTRING function
- $LISTVALID function

$LISTNEXT

Retrieves elements sequentially from a list.

```
$LISTNEXT(list,ptr,value)
```

Parameters

list	An expression that resolves to a Caché list. A Caché list is created using $LISTBUILD or $LISTFROMSTRING, or extracted from another list using $LIST.
ptr	A pointer to the next element in the list. You must specify *ptr* as a local variable initialized to 0 to point to the beginning of *list*. *ptr* cannot be a global variable or a subscripted variable.
value	A local variable used to hold the data value of a list element. *value* does not have to be initialized before invoking **$LISTNEXT**. *value* cannot be a global variable or a subscripted variable.

Description

$LISTNEXT sequentially returns elements in a *list*. You initialize *ptr* to 0 before the first invocation of **$LISTNEXT**. This causes **$LISTNEXT** to begin returning elements from the beginning of the list. Each successive invocation of **$LISTNEXT** advances *ptr* and returns the next list element value to *value*. The **$LISTNEXT** function returns 1, indicating that a list element has been successfully retrieved.

When **$LISTNEXT** reaches the end of the list, it returns 0, resets *ptr* to 0, and leaves *value* unchanged from the previous invocation. Because *ptr* has been reset to 0, the next invocation of **$LISTNEXT** would start at the beginning of the list.

Caché MVBasic increments *ptr* using an internal address algorithm. Therefore, the only value you should use to set *ptr* is 0.

You can use **$LISTVALID** to determine if *list* is a valid list. An invalid list causes **$LISTNEXT** to generate a <LIST> error.

Not all lists validated by **$LISTVALID** can be used successfully with **$LISTNEXT**. When **$LISTNEXT** encounters a list element with a null value, it returns 1 indicating that a list element has been successfully retrieved, advances *ptr* to the next element, and resets *value* to be an undefined variable. This can happen with any of the following valid lists: *value*=$LB(), *value*=$LB(NULL), *value*=$LB(,), or when encountering an omitted list element, such as the second invocation of **$LISTNEXT** on *value*=$LB("a",,"b").

`$LISTNEXT("",ptr,value)` returns 0, and does not advance the pointer or set *value*.
`$LISTNEXT($LB(""),ptr,value)` returns 1, advances the pointer, and set *value* to the null string ("").

$LISTNEXT and Nested Lists

The following example returns three elements, because **$LISTNEXT** does not recognize the individual elements in nested lists:

```
list=$LISTBUILD("Apple","Pear",$LISTBUILD("Walnut","Pecan"))
ptr=0
count=0
LOOP UNTIL 0=$LISTNEXT(list,ptr,value)
 count=count+1
 CRT value
REPEAT
CRT "End of list:",count,"elements found"
```

Examples

The following example sequentially returns all the elements in the list:

```
 list=$LISTBUILD("Red","Blue","Green")
 ptr=0
 count=0
LOOP UNTIL 0=$LISTNEXT(list,ptr,value)
  count=count+1
  CRT value
REPEAT
 CRT "End of list:",count,"elements found"
```

See Also

- $LIST function
- $LISTBUILD function
- $LISTDATA function
- $LISTFIND function
- $LISTFROMSTRING function
- $LISTGET function
- $LISTLENGTH function
- $LISTSAME function
- $LISTTOSTRING function
- $LISTVALID function

$LISTSAME ($LS)

Compares two lists and returns a boolean value.

```
$LISTSAME(list1,list2)
$LS(list1,list2)
```

Parameters

list	An expression that resolves to a Caché list. A Caché list is created using $LISTBUILD or $LISTFROMSTRING, or extracted from another list using $LIST.

Description

$LISTSAME compares the contents of two lists and returns 1 if the lists are identical. If the lists are not identical, **$LISTSAME** returns 0.

A Caché list can either be a list created using **$LISTBUILD**, or a null string (""). If *list* is not a valid list, you receive a <LIST> error.

Examples

The following example returns 1, because the two lists are identical:

```
x = $LISTBUILD("Red","Blue","Green")
y = $LISTBUILD("Red","Blue","Green")
PRINT $LISTSAME(x,y)
```

The following example returns 0, because the two lists are not identical:

```
x = $LISTBUILD("Red","Blue","Yellow")
y = $LISTBUILD("Red","Blue","Green")
PRINT $LISTSAME(x,y)
```

Notes

Identical Lists

$LISTSAME considers two lists to be identical if the string representations of the two lists are identical. This is not the same equivalence test as the one used by other list operations, which test using the internal representation of a list. This distinction is easily seen when comparing a number and a numeric string, as in the following example:

```
x = $LISTBUILD("365")
y = $LISTBUILD(365)
   IF x'=y { PRINT "number and numeric string lists differ" }
PRINT $LISTSAME(x,y)," number and numeric string lists identical"
```

The **IF** comparison tests the internal representations of these lists (which are not identical). **$LISTSAME** performs a string conversion on both lists, compares them, and finds them identical.

The following example shows two lists with various representations of numeric elements. **$LISTSAME** considers these two lists to be identical:

```
x = $LISTBUILD("360","361","362","363","364","365","366")
y = $LISTBUILD(00360.000,(19*19),+"362",363,364.0,+365,"3"_"66")
PRINT $LISTSAME(x,y)," lists are identical"
```

In the following example, both **$LISTSAME** comparisons returns 0, because these lists are not considered identical:

```
x=$LISTBUILD("Apple","Pear","Walnut","Pecan")
y=$LISTBUILD("Apple","Pear",$LISTBUILD("Walnut","Pecan"))
z=$LISTBUILD("Apple","Pear","Walnut","Pecan","")
PRINT $LISTSAME(x,y)," nested list"
PRINT $LISTSAME(x,z)," null string is list item"
```

The following example returns 1, because the lists are considered identical:

```
x=$LISTBUILD("Apple","Pear","Walnut","Pecan")
y=$LISTBUILD("Apple","Pear")_$LISTBUILD("Walnut","Pecan")
PRINT $LISTSAME(x,y)," concatenate lists"
```

Null String and Null List

A list containing the null string (an empty string) as its sole element is a valid list. The null string by itself is also considered a valid list. However these two (a null string and a null list) are not considered identical, as shown in the following example:

```
PRINT $LISTSAME($LISTBUILD(""),$LISTBUILD(""))," null lists"
PRINT $LISTSAME("","")," null strings"
PRINT $LISTSAME($LISTBUILD(""),"")," null list and null string"
```

Normally, a string is not a valid **$LISTSAME** argument, and **$LISTSAME** issues a <LIST> error. However, the following **$LISTSAME** comparisons complete successfully and return 0. The null string and the string “abc” are compared and found not to be identical. These null string comparisons do not issue a <LIST> error:

```
PRINT $LISTSAME("","abc")
PRINT $LISTSAME("abc","")
```

The following **$LISTSAME** comparisons do issue a <LIST> error, because a list (even a null list) cannot be compared with a string:

```
x = $LISTBUILD("")
PRINT $LISTSAME("abc",x)
PRINT $LISTSAME(x,"abc")
```

See Also

- $LIST function
- $LISTBUILD function
- $LISTDATA function
- $LISTFIND function
- $LISTFROMSTRING function
- $LISTGET function
- $LISTLENGTH function
- $LISTNEXT function
- $LISTTOSTRING function
- $LISTVALID function

$LISTTOSTRING ($LTS)

Creates a string from a list.

```
$LISTTOSTRING(list,delimiter)
$LTS(list,delimiter)
```

Parameters

list	An expression that resolves to a Caché list. A Caché list is created using **$LISTBUILD** or **$LISTFROMSTRING**, or extracted from another list using **$LIST**.
delimiter	*Optional* — A delimiter used to separate substrings. An expression that resolves to a string containing one or more characters. If no *delimiter* is specified, the default is the comma (,) character. The *delimiter* cannot contain the character used to delimit the quoted string.

Description

$LISTTOSTRING takes a Caché list and converts it to a string. In the resulting string, the elements of the list are separated by the *delimiter*.

A list represents data in an encoded format which does not use delimiter characters. Thus a list can contain all possible characters, and is ideally suited for bitstring data. **$LISTTOSTRING** converts this list to a string with delimited elements. It sets aside a specified character (or character string) to serve as a delimiter. These delimited elements can be handled using the **$PIECE** function.

Note: The *delimiter* specified here must not occur in the source data. Caché makes no distinction between a character serving as a delimiter and the same character as a data character.

Parameters

list

A Caché list, which contains one or more elements. A list is created using **$LISTBUILD** or extracted from another list using **$LIST**.

If the expression in the *list* parameter does not evaluate to a valid list, a <LIST> error occurs.

delimiter

A character (or string of characters) used to delimit substrings within the output string. It can be a numeric or string literal (enclosed in quotation marks), the name of a variable, or an expression that evaluates to a string.

Commonly, a delimiter is a designated character which is never used within string data, but is set aside solely for use as a delimiter separating substrings. A delimiter can also be a multi-character string, the individual characters of which can be used within string data.

If you specify no delimiter, the default delimiter is the comma (,) character. You can specify a null string ("") as a delimiter; in this case, substrings are concatenated with no delimiter. To specify a quote character as a delimiter, either use another delimiter for the string (for example, \"\) or use the CHAR() function (34=", 39=', 92=\).

Example

The following example creates a list of four elements, then converts it to a string with the elements delimited by the colon (:) character:

```
namelist=$LISTBUILD("Deborah","Noah","Martha","Bowie")
PRINT $LISTTOSTRING(namelist,":")
```

returns `"Deborah:Noah:Martha:Bowie"`

See Also

- $LIST function
- $LISTBUILD function
- $LISTDATA function
- $LISTFIND function
- $LISTFROMSTRING function
- $LISTGET function
- $LISTLENGTH function
- $LISTNEXT function
- $LISTSAME function
- $LISTVALID function

$LISTVALID

Determines if an expression is a list.

```
$LISTVALID(exp)
$LV(exp)
```

Parameters

exp	An expression.

Description

$LISTVALID determines whether *exp* is a Caché list, and returns a Boolean value: If *exp* is a list, **$LISTVALID** returns 1; if *exp* is not a list, **$LISTVALID** returns 0.

A list can be created using **$LISTBUILD** or **$LISTFROMSTRING**, or extracted from another list using **$LIST**. A list containing the empty string ("") as its sole element is a valid list. The empty string ("") by itself is also considered a valid list.

Examples

The following examples all return 1, indicating a valid list:

```
w = $LISTBUILD("Red","Blue","Green")
x = $LISTBUILD("Red")
y = $LISTBUILD(365)
z = $LISTBUILD("")
CRT $LISTVALID(w)
CRT $LISTVALID(x)
CRT $LISTVALID(y)
CRT $LISTVALID(z)
```

The following examples all return 0. Numbers and strings (with the exception of the null string) are not valid lists:

```
x = "Red"
y = 44
CRT $LISTVALID(x)
CRT $LISTVALID(y)
```

The following examples all return 1. Concatenated, nested, and omitted value lists are all valid lists:

```
w=$LISTBUILD("Apple","Pear")
x=$LISTBUILD("Walnut","Pecan")
y=$LISTBUILD("Apple","Pear",$LISTBUILD("Walnut","Pecan"))
z=$LISTBUILD("Apple","Pear",,"Pecan")
CRT $LISTVALID(w:x)   ! concatenated
CRT $LISTVALID(y)     ! nested
CRT $LISTVALID(z)     ! omitted element
```

The following examples all return 1. **$LISTVALID** considers all of the following “empty” lists as valid lists:

```
CRT $LISTVALID("")
CRT $LISTVALID($LB())
CRT $LISTVALID($LB(NULL))
CRT $LISTVALID($LB(""))
CRT $LISTVALID($LB(CHAR(0)))
CRT $LISTVALID($LB(,))
```

See Also

- $LIST function
- $LISTBUILD function

- $LISTDATA function
- $LISTFIND function
- $LISTFROMSTRING function
- $LISTGET function
- $LISTLENGTH function
- $LISTNEXT function
- $LISTSAME function
- $LISTTOSTRING function

LN

Returns the natural logarithm of a number.

```
LN(number)
```

Arguments

number	An expression that resolves to a positive number.

Description

The **LN** function returns the natural logarithm of *number*. The *number* value must be a non-zero positive number; all other values generate an <ILLEGAL VALUE> error.

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

dynarray	An expression that resolves to a 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. A *dynarray* value that does not contain dynamic array delimiters 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 123ý456ý10ý11
numlower = LOWER(numvm)
PRINT numlower;       ! Returns 123ü456ü10ü11
numraise = RAISE(numlower)
PRINT numraise;       ! Returns 123ý456ý10ý11
```

See Also

- RAISE function
- Dynamic Arrays

LTS

Performs a less than comparison on elements of two dynamic arrays.

```
LTS(dynarray1,dynarray2)
```

Arguments

dynarray	An expression that resolves to a 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 missing or has a null string or non-numeric value, **LTS** assigns it a value of 0 for the purpose of this comparison.

If the two dynamic arrays have different numbers of elements, the returned dynamic array has the number of elements of the longer dynamic array. By default, the shorter dynamic array is padded with 0 value elements for the purpose of comparison. You can also use the REUSE function to define behavior when specifying two dynamic arrays with different numbers of elements.

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 0ý0ý1ý0
```

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

dynarray	An expression that resolves to a 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 missing or has a null string or 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

dynarray	An expression that resolves to a 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 missing or has a null string or 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

numstr1	The dividend. An expression that resolves to a number or numeric string.
numstr2	The divisor. An expression that resolves to a non-zero number 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 the null string or a non-numeric value, **MOD** parses its value as 0 (zero).

Attempting to divide by zero generates a <DIVIDE> error, ending execution of the function and invoking an error trap handler, if available.

The **REM** function is functionally identical to the **MOD** function. To perform integer division, use the **DIV** function. To perform exact division with a fractional quotient, use the division operator (/).

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 corresponding elements in two dynamic arrays (zero divide not allowed).

```
MODS(dynarray1,dynarray2)
```

Arguments

dynarray	An expression that resolves to a 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 is missing or has a null string or a non-numeric value, **MODS** parses its value as 0 (zero).

Attempting to divide by zero generates a <DIVIDE> error, ending execution of the function and invoking an error trap handler, if available. This is the same 0 divisor behavior as the **DIVS** function. In contrast, when the corresponding MODSZ or **DIVSZ** functions encounter a 0 divisor, they return 0 for that element.

If the two dynamic arrays have different numbers of elements, the returned dynamic array has the number of elements of the longer dynamic array. By default, the shorter dynamic array is padded with 0 value elements for the purpose of the arithmetic operation. You can also use the REUSE function to define behavior when specifying two dynamic arrays with different numbers of elements.

You can use the **MOD** function or the **REM** function to perform modulo division on single values.

You can use the **NUMS** function to determine if the elements in a dynamic array are numeric. You can use the ADDS (addition), SUBS (subtraction), MULS (multiplication), DIVS or DIVSZ (division), and PWRS (exponentiation) functions to perform other arithmetic operations on the corresponding elements of two dynamic arrays.

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 1ý0ý0ý-7
```

See Also

- ADDS function
- DIVS function
- DIVSZ function
- MOD function
- MODSZ function
- MULS function
- PWRS function
- REM function
- SUBS function
- Dynamic Arrays

MODSZ

Modulo division of corresponding elements in two dynamic arrays (zero divide allowed).

```
MODSZ(dynarray1,dynarray2)
```

Arguments

dynarray	An expression that resolves to a dynamic array of numeric values.

Description

The **MODSZ** 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 is missing or has a null string or a non-numeric value, **MODSZ** parses its value as 0 (zero).

When **MODSZ** encounters a 0 divisor, it returns 0 for that element. This is the same 0 divisor behavior as the **DIVSZ** function. The corresponding MODS function does not allow division by zero; attempting to divide by zero generates a <DIVIDE> error, ending execution of the function and invoking an error trap handler, if available.

If the two dynamic arrays have different numbers of elements, the returned dynamic array has the number of elements of the longer dynamic array. By default, the shorter dynamic array is padded with 0 value elements for the purpose of the arithmetic operation. You can also use the REUSE function to define behavior when specifying two dynamic arrays with different numbers of elements.

You can use the **MOD** function or the **REM** function to perform modulo division on single values.

You can use the **NUMS** function to determine if the elements in a dynamic array are numeric. You can use the ADDS (addition), SUBS (subtraction), MULS (multiplication), DIVS or DIVSZ (division), and PWRS (exponentiation) functions to perform other arithmetic operations on the corresponding elements of two dynamic arrays.

Examples

The following example uses the **MODSZ** 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 MODSZ(a,b);  ! returns 1ý0ý0ý-7
```

See Also

- ADDS function
- DIVS function
- DIVSZ function
- MOD function
- MODS function
- MULS function
- PWRS function
- REM function
- SUBS function
- Dynamic Arrays

MULS

Multiplies the values of corresponding elements in two dynamic arrays.

```
MULS(dynarray1,dynarray2)
```

Arguments

dynarray	An expression that resolves to a 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 products of these multiplications. If a *dynarray* element value is missing, or has a null string or non-numeric value, **MULS** parses its value as 0 (zero).

If the two dynamic arrays have different numbers of elements, the returned dynamic array has the number of elements of the longer dynamic array. By default, the shorter dynamic array is padded with 0 value elements for the purpose of the arithmetic operation. You can also use the REUSE function to define behavior when specifying two dynamic arrays with different numbers of elements.

You can use the **NUMS** function to determine if the elements in a dynamic array are numeric. You can use the ADDS (addition), SUBS (subtraction), DIVS or DIVSZ (division), MODS (modulo division), and PWRS (exponentiation) functions to perform other arithmetic operations on the corresponding elements of two dynamic arrays.

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 30ý11ý0ý-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 30ý11ý0ý0
```

See Also

- SMUL function
- ADDS function
- DIVS function
- DIVSZ function
- MODS function
- PWRS function
- SUBS function
- Dynamic Arrays

NEG

Returns the inverse sign of a number.

```
NEG(number)
```

Arguments

number	An expression that resolves to 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 the empty string 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

NEGS

Returns the inverse sign of each number in a dynamic array.

```
NEGS(dynarray)
```

Arguments

dynarray	An expression that resolves to a dynamic array of numeric values. Its elements can contain values specified as a number or a numeric string.

Description

The **NEGS** function inverts the sign of each numeric element of *dynarray*. These output values are returned as a dynamic array. A value of 1 is returned as -1, and a value of -1 is returned as 1. **NEGS** returns numbers (both numerics and numeric strings) in canonical form, removing multiple signs and leading and trailing zeros from each value. A string is parsed as a number until a non-numeric character is encountered. Thus “7dwarves” is parsed as 7, and returns -7. If an element value is missing or has an empty string or non-numeric value, **NEGS** returns 0 (zero).

The **NEGS** function inverts the sign of each number in a dynamic array: negative numbers become positive and positive numbers become negative. The **ABSS** function gives the absolute value of each number in a dynamic array: all numbers become positive.

You can use the **NEG** function to invert the sign of a single number.

Examples

The following example uses the **NEGS** function to invert the sign of each number in a dynamic array:

```
nums=123:@VM:-12.300:@VM:++0123.00:@VM:"-123.00":@VM:""
PRINT NEGS(nums)  ! Returns -123ý12.3ý-123ý123ý0
```

See Also

- ABSS function
- NEG function
- Dynamic Arrays

NES

Performs an inequality comparison on elements of two dynamic arrays.

```
NES(dynarray1,dynarray2)
```

Arguments

dynarray	An expression that resolves to a 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 missing, or has a null string or 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 the two dynamic arrays have different numbers of elements, the returned dynamic array has the number of elements of the longer dynamic array. By default, unmatched elements return 1 (not equal). That is, the **NES** comparison of each element in the longer dynamic array that has no corresponding element in the shorter dynamic array always returns 1 (not equal), even when the value of the longer array element is 0 or the null string, or is a missing element within the dynamic array. You can also use the REUSE function to define behavior when specifying two dynamic arrays with different numbers of elements.

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 0ý1ý1ý0
```

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

expression	An expression that resolves to a boolean value.

Description

The **NOT** function returns the logical complement (inverse) of a boolean expression. Thus all expressions that evaluate to 0 become 1, and all expressions that evaluate to 1 (or any 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 (boolean 1), and thus **NOT** returns 0. If *expression* is the null string or a non-numeric value, **NOT** parses it as boolean 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 inverse of a boolean expression:

```
PRINT NOT(1);          ! Returns 0
PRINT NOT(0);          ! Returns 1
PRINT NOT(7);          ! Returns 0
PRINT NOT(-7);         ! Returns 0
PRINT NOT("7dwarves"); ! Returns 0
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

dynarray	An expression that resolves to a dynamic array of boolean values.

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 is missing or contains 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 0ý0ý0ý1ý1ý1
```

See Also

- ANDS function
- ORS function
- NOT function
- Dynamic Arrays
- Operators

NUM

Returns whether a value is numeric.

```
NUM(string)
```

Arguments

string	An expression that resolves to a 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.

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

dynarray	An expression that resolves to a 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 a missing element value.

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.

Examples

The following example uses the **NUMS** function to determine the numeric status of the elements of a dynamic array:

```
a=3:@VM:007:@VM:"+33.00":@VM:"":@VM:0
PRINT NUMS(a);  ! returns 1ý1ý1ý1ý1
```

The following example show how **NUMS** handles non-numeric strings, arithmetic expressions, and missing elements:

```
b="7dwarves":@VM:"dwarves7":@VM:@VM:(4+3):@VM:"hello world"
PRINT NUMS(b);  ! returns 0ý0ý1ý1ý0
```

See Also

- NUM function
- SCMP function
- Dynamic Arrays

OCONV

Converts a value from internal format to external format.

```
OCONV(istring,code)
```

Arguments

istring	An expression that resolves to a string or integer. It specifies a value represented in internal (storage) format.
code	An expression that resolves to a conversion code string. This conversion code specifies the type of conversion to perform. Conversion is from internal format to external format. Conversion codes are not case-sensitive. For a complete list of conversion codes, refer to the Conversion Codes table in the *MultiValue Basic Quick Reference*.

Description

The **OCONV** function is a general-purpose conversion function used to convert from internal (storage) format to external (output) format. The type of conversion is specified by a *code* string that is specific to the type of data to be converted.

The following types of conversions are supported:

- Character conversions: character-to-hexcode, hexcode-to-character.
- Numeric conversions: decimal-to-hex, hex-to-decimal, masked decimal conversion, range extraction, zero and non-zero substitution.
- String Conversions: case conversion, character type extraction, Soundex conversion, string length conversion, uniform string length adjustment, substring extraction by length, delimited substring extraction, pattern match extraction, extraction of strings of a specified length.
- Time and Date Conversions: time internal-to-display format, date internal-to-display format, date display-to-internal format, date element extraction, day-of-week, day-of-year, and quarter calculation.
- Arithmetic and Logical Operations: arithmetic, equality comparisons, collation sequence comparisons, date and time arithmetic.
- String Concatenation: concatenation, concatenation with inserted single-character delimiters.
- Dynamic Array Element Extraction: extracting an element from a dynamic array by position.

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. Note that the MCDX/MCXD, MCAX/MCXA, and MCWX/MCXW *code* pairs have the opposite meanings in **ICONV**, reversing the **OCONV** operation.

You can use the **OCONVS** function to convert the elements of a dynamic array from internal format to external format.

Invalid Values

In most cases, if you specify an *istring* value that cannot be converted, **OCONV** returns the *istring* value unchanged.

- If you supply a non-numeric value to a numeric operation: date, time, and masked decimal conversions return the *istring* value unchanged. However, "DI" returns -46384. "MCX" and the other decimal/hex conversion return 0 (unless the first character(s) of the non-numeric string are viewed as the hexadecimal digits A through F.)

- If you supply a non-alphabetic value to an alphabetic operation: case conversions return the *istring* value unchanged. Soundex conversion returns 0000.

If you specify a *code* value that is valid but is not implemented, **OCONV** generates an [806] error. If you specify a nonexistent *code* value, **OCONV** generates an [850] error.

Character Conversion

The following are the conversion codes for character conversions:

Character-to-code conversion: all characters in the *istring* input string are converted to their corresponding hexadecimal integer codes. Use "MCAX", "MX", or “MX0C” to output 8-bit code values. Use "MCWX" to output 16-bit (wide) code values.	"MCAX", "MX", “MX0C "MCWX"
Code-to-character conversion: one or more hexadecimal codes in the *istring* input string are converted to their corresponding characters. You can specify the hexadecimal letters A-F in uppercase or lowercase. Use "MCXA" or "MY" for two-byte code input values. Use "MCXW" for four-byte (wide) code input values.	"MCXA", "MY" "MCXW"

Numeric Conversion

The following are the conversion codes for numeric conversions:

Decimal-to-hexadecimal conversion.	"MCD" / "MCDX"
Hexadecimal-to-decimal conversion.	"MCX" / "MCXD"
Masked decimal conversion: Shifts the decimal point of a numeric, sets the number of fractional digits, optionally limits the number of characters of the result to return, and optionally performs other number formatting, such as converting a minus sign or appending a currency symbol. The *m* integer value specifies the shift of the decimal point, with a positive value shifting the decimal point to the left, and with 0 indicating no shift; left side zero padding is added as needed. The *n* integer value specifies the number of fractional digits; rounding (by default) and zero padding are performed as needed. If *m* is omitted, it defaults to the same value as *n*. The *k* integer specifies how many characters of the result to return. The *x* modifiers are non-numeric character codes for formatting results; you can specify multiple *x* codes. For *x* code values, see below. **OCONV** masked decimal conversion is the reverse of the **ICONV** masked decimal conversion.	"MD[n[m[k]]][x]" "ML[n[m[k]]][x]" "MR[n[m[k]]][x]"
Range extraction: returns *istring* if it falls within the numeric range *n* and *m* (inclusive). Otherwise, returns an empty string. *istring* can be any numeric value; *n* and *m* must be positive integers. You can specify multiple range pairs, separating them with a semicolon (;) or slash (/). For further details, see below.	"Rn,m[;n,m]"
Zero and non-zero substitution: If *istring* is a non-zero value (either numeric or string), returns the *nval* literal. If *istring* is a zero value (0 or the empty string), returns the *zval* literal. *nval* and *zval* must both be specified (separated by semicolons) as either a quoted string or as an asterisk. You can substitute an asterisk (*) for either *nval* or *zval*; when * is specified, the *istring* literal is returned for that condition, rather than a substitute value.	"S;nval;zval"

Hexadecimal / Decimal Conversion

“MCD” and “MCDX” convert a positive decimal integer to a hexadecimal number. Hexadecimal numbers A through F are returned in upper case. A fractional number is truncated before conversion to hexadecimal. A mixed numeric string is parsed as an integer until the first non-numeric value is encountered, at which point it is truncated and the resulting integer converted to hexadecimal. A negative number is converted to a high-order hexadecimal value; for example -3 is converted to FFFFFFFFFFFFFFFD. A non-numeric string returns 0.

“MCX” and “MCXD” convert a hexadecimal number to a positive decimal integer. The hexadecimal letter A-F are not case-sensitive. A fractional number is truncated before conversion to a decimal integer. A mixed numeric string is parsed as a hexadecimal number until the first non-hexadecimal value is encountered, at which point it is truncated and the

resulting hexadecimal string is converted to a decimal integer. A negative number returns 0. A non-numeric string returns 0.

Masked Decimal Conversion / Currency Conversion

“MD”, “ML”, and “MR” convert an integer by moving the decimal point, specifying the number of fractional digits, and (optionally) inserting a currency symbol or other numeric format characters. For example, to convert the stored integer 123456 to a displayed dollar currency value, you could use the conversion code `"MD22$,"`, which would result in the value $1,234.56.

“MD” removes leading zeros. “ML” (left justification) and “MR” (right justification) retain leading zeros.

The *istring* is commonly an integer. If *istring* is a fractional number, it is rounded to an integer before applying masked decimal conversion. *istring* can contain the numbers 0–9, a leading minus sign, a decimal point, and numeric group separators (commas). All other characters are considered non-numeric, and cause **OCONV** to return *istring* unchanged. Note that a leading plus sign (+) or a dollar sign ($) is considered a non-numeric character and prevents conversion.

The *istring* value is rounded to an integer, and then the *n* and *m* integer arguments are applied:

- *n* specifies the number of fractional digits in the result. A positive integer in the range 0 through 9 (inclusive) sets the number of fractional digits. The *n* argument is optional; the default is 0. If *n* is omitted, *m* is also omitted and defaults to 0.
- *m* specifies how many places (power of ten) to the left to move the decimal point. If *m* is omitted, it is assigned the same value as *n*.

Zero padding is added where needed.

This use of *n* and *m* is shown in the following examples:

```
PRINT OCONV("0123.57","MD")     ! returns 124: n defaults to 0, m defaults to n
PRINT OCONV("0123.57","MD00")   ! returns 124: same as above
PRINT OCONV("0123.57","MD2")    ! returns 1.24: n=2 fractional digits, m defaults to n (2 places left)
PRINT OCONV("0123.57","MD22")   ! returns 1.24: same as above
PRINT OCONV("0123.57","MD21")   ! returns 12.36: n=2 fractional digits, m moves decimal 1 place left
PRINT OCONV("0123.57","MD02")   ! returns 1: n=no fractional digits, m moves decimal 2 places left
```

If *istring* is the null string (""), “MD” conversions always returns the null string. “ML” and “MR” conversions treats the null string as zero and apply the specified numeric conversions. This null string behavior is emulation-dependent:

- Caché, IN2, INFORMATION, PICK, PIOpen, Reality, Universe: treat the null string ("") as zero for numeric conversions. Numeric conversions are applied.
- D3, jBASE, MVBase, R83, POWER95, Ultimate, UniData: treat the null string ("") as null, and return the null string for numeric conversions.

Formatting Codes: x

The following optional *x* codes can be specified following the optional *n* and *m* codes:

- T: truncate the results of applying the *n* and *m* codes. The default is to round the results. Thus by applying `"MD23T"` "123456" becomes "123.45"; by applying `"MD23"` "123456" becomes "123.46".
- P: if *istring* already contains a decimal point, the decimal point is retained and the *m* value is not applied. Thus by applying `"MD22P"` "123456" becomes "1234.56", but "12.3456" becomes "12.35". If "P" is not specified, `"MD22"` causes "12.3456" to become "0.12".
- - (minus sign) or M: moves the minus sign from leading to trailing. Thus by applying `"MD22-"` "-123456" becomes "1234.56-". Positive numbers are unaffected by this code character.
- N: removes the minus sign from a negative number. Has no effect on a positive number.

- C (credit) or D (debit): When *x*=C positive numbers are unaffected, negative numbers lose their minus sign and take a CR suffix. When *x*=D positive numbers take a DB suffix, negative numbers lose their minus sign. Thus by applying `"MD22D"` "123456" becomes "1234.56DB" and "-123456" becomes "1234.56". Specifying lowercase "c" or "d" results in a corresponding lowercase cr or db suffix.
- < (left angle bracket) or E: delimits a negative value with angle brackets. Thus by applying `"MD22<"` "-123456" becomes "<1234.56>". Positive numbers are unaffected by this code character.
- $ (dollar sign) : appends a dollar sign to the conversion result. Thus by applying `"MD22$"` "123456" becomes "$1234.56" and "-123456" becomes "$-1234.56". If supported by the locale, the F (franc), I (international), and Y (yen) currency symbols are applied; if not supported by the locale, these are synonyms for $.
- , (comma): inserts numeric group separators. Thus by applying `"MD22,"` "123456" becomes "1,234.56" and "-123456" becomes "-1,234.56".

  If *istring* already contains one or more commas, these conversion codes handle existing commas as follows: a comma in *istring* is treated as a digit when applying *n* and *m* codes. After applying *n* and *m*, the conversion removes all commas to the left of the decimal point. Then, if *x* is a comma, the conversion adds commas as numeric group separators where needed.
- Z: zero converted to empty string. If *istring* has a zero value, the Z conversion code returns the empty string rather than zero. Thus by applying `"MD22Z"` the *istring* values "0", "0.00", or "-000" all become the empty string. However, note that Z is applied before *n* and *m* processing; thus applying `"MD22Z"` to "0.1" results in "0.00". Appending a “Z” character code causes “ML” and “MR” conversions to return an empty string when *istring* is the empty string.

The E, M, N, P, T, and Z letter codes are not case-sensitive.

Multiple *x* code suffixes can be combined in any order. For example, `"MD22$-,"` appends a dollar sign to the decimal fraction, moves any existing minus sign to the trailing position, and inserts numeric group separators where needed. Thus `OCONV(-123456,"MD22$-,")` returns "$1,234.56-". The *x* codes are applied in left-to-right order. Therefore, if multiple *x* code suffixes conflict (for example, C, <, and –), the last (rightmost) *x* code is the one applied.

Numbers of Characters to Return: k

You can optionally specify how many characters of the masked decimal conversion result to return. For “MD” and “MR” these are the rightmost characters of the conversion result. For “ML” these are the leftmost characters of the conversion result.

The number of characters to return (*k*) can be specified in several ways:

- As a third positive integer value following *n* and *m*. For example, “MD224” specifies three integers: *n* the number of fractional digits; *m* the number of places to move the decimal point to the left; and *k* the number of characters of the result to display. Unlike *n* and *m*, *k* cannot be 0, but can be an integer larger than 9. Thus by applying `"MD22"` "123456" becomes "1234.56" and by applying `"MD224"` "123456" becomes "4.56". Note that the decimal point is counted as a character.

  In this syntax, *n*, *m*, and *k* must be explicitly specified. *x* codes should not be specified.

  For “MD” and “MR”, the *k* characters are counted from right to left. For “ML” the *k* characters are counted from left to right. If *k* exceeds the number of characters in the result, the result is padded with blank spaces.
- As a multi-character *x* code, the first character of which is a #, %, or * character, followed by the *k* integer value. This code can be specified with or without delimiting parentheses. Thus `"MD224"`, `"MD22#4"`, and `"MD22(#4)"` return the same results.

  This syntax does not require explicit *n* and/or *m* values. Thus `"MD#4"`, `"MD(#4)"`, and `"MD004"` return the same results.

  This syntax allows you to apply *x* codes before or after selecting the character subset. Code characters are applied in left-to-right order. Therefore, any *x* codes specified to the left of the *k* code are included in the *k* count. *x* codes to the

right of the *k* code are applied after the *k* count. Thus by applying `"MD22-#3"` "-123456" returns the three characters "56–"; the trailing minus sign is applied before the *k* count. Applying `"MD22#3-"` to "–123456" returns the four characters ".56–"; the trailing minus sign is applied after the *k* count.

You can add a string suffix to the returned numeric string. Use the following syntax, with or without the enclosing parentheses: MD*nm* (*#kstring*), where *string* is one or more non-numeric characters. For example, to add the string suffix “salary” separated from the numeric by a single space: `"MD22$(#8 salary)"` or `"MD22$#8 salary"`.

You can add a fill character suffix to the returned numeric string. Use the following syntax, with or without the enclosing parentheses: MD*nm* (*#kcharr*), where *char* is a single non-numeric character and *r* is an integer repetition count. For example, to add the fill character suffix “^^^^^” to the numeric: `"MD22$(#8^5)"` or `"MD22$#8^5"`.

You can add multiple string suffixes and fill character suffixes. For example, to add the suffix `^^^^URGENT^^^^`, you would specify `"MD22$(#8^4URGENT^4)"`.

Range Extraction

The "Rn,m" code extracts positive numeric values within the specified range (inclusive). The *n* and *m* values specify the bounds of the range. They must be positive integers; 0 is a permitted value. A range may be in ascending or descending order (ascending is preferable when specifying multiple ranges). The *n* and *m* values may be the same value. If a number is within the range (inclusive of the *n* and *m* values) it is returned. If a number is not within the range, the empty string is returned. Signed numbers are returned with their sign; however, the only negative number that can be returned is –0. Fractional numbers are evaluated as being larger than their integer (they are neither rounded nor truncated); thus 1.9 is not within the range 2,4 or in the range 0,1 but is within the range 1,2. An *istring* containing non-numeric characters is parsed as 0.

You can specify multiple ranges by separating each range pair with either a semicolon or a slash. Thus `"R2,4;8,10"` is a valid range which will return the integers 2, 3, 4, 8, 9, or 10. When specifying multiple ranges, each successive range must start at a number equal to or greater than the low value of the previous range. Thus `"R2,4;6,10"`, `"R2,4;3,6"`, or `"R2,4;2,6"` are valid range codes. `"R2,4;1,6"` is not a valid range code; only the first range is parsed as a range; the lower number of the second range is parsed, extending the first range downward. Thus `"R2,4;2,6"` returns 2, 3, 4, 5, or 6; `"R2,4;1,6"` returns 1, 2, 3, or 4.

Masked String Conversion

The following are the codes for string and numeric conversions:

Case conversion: converts the case of alphabetic characters in *istring*; has no effect on non-alphabetic characters. "MCL" converts uppercase letters to lowercase. "MCU" converts lowercase letters to uppercase.	"MCL" "MCU"
Title case conversion: converts the initial letter of each word to uppercase, other letters converted to lowercase. The first letter following an apostrophe is also converted to uppercase, unless that letter is followed by a blank or other non-letter character (thus “O’Brian’s Account”, “Three O’Clock”). "MCT" has no effect on non-letter characters.	"MCT"
Mask Character Alphabetic: converts *istring* by removing all non-alphabetic characters, returning only the alphabetic characters. The inverse is “MC/A” which removes all alphabetic characters, returning only the non-alphabetic characters.	“MCA”

Mask Character Both Alphabetic and Numeric: converts *istring* by removing all punctuation characters, returning only the alphabetic and numeric characters. The inverse is “MC/B” which removes all alphabetic and numeric characters, returning only the punctuation characters.	“MCB”
Mask Character Numeric: converts *istring* by removing all non-numeric characters, returning only the number characters 0 through 9. (Note that plus and minus signs and the decimal point are removed.) The inverse is “MC/N”, which removes all number characters (0 through 9), returning only non-numeric characters.	“MCN”
Non-printable character conversion: converts *istring* by replacing each non-printable character with a period (.). It returns the resulting string of printable characters and periods.	“MCP”
Soundex conversion: represents the *istring* alphabetic string with a four-character Soundex representation. For further details, refer to the SOUNDEX function.	"S"
Length conversion: "L" or "L0": returns the number of characters in *istring*. "Ln": returns the value of *istring* if *n* is exactly the number of characters in *istring*. Otherwise, returns the empty string. *n* must be a positive non-zero integer. "Ln-m" or "Ln,m": returns the value of *istring* if the number of characters in *istring* is in the range *n* through *m* (inclusive). Otherwise, returns the empty string. *n* can be specified as zero in this syntax.	"L" or "L0" "Ln" "Ln-m" or "Ln,m"
Uniform string length adjustment: returns *istring* with trailing padding characters and/or text mark (@TM) insertions. *n* is an integer specifying the desired uniform string length and *x* is a single non-numeric padding character (for example, "7#" converts *istring* to one or more strings each string being 7 characters long). If *n* is larger than the length of *istring*, **OCONV** pads *istring* with *x* characters to a total length of *n* characters. If *n* is smaller than the length of *istring*, **OCONV** inserts @TM delimiters every *n* characters, optionally padding with *x* so that all delimited substrings are the same length. If *n* is the same as the length of *istring*, *istring* is returned unchanged.	"nx"

Text substring extraction: a substring is extracted from *istring* based on a *start* position integer and a *length* integer. The *start* position integer is optional: if *start* is specified, *length* is counted left-to-right from that position; if *start* is omitted, *length* is counted right-to-left from the end of the string. If specified, *start* must be positive integer 1 or greater. *length* must be positive integer 0 or greater. If *start* exceeds the length of *istring*, the empty string is returned. If *start* is not specified and *length* equals or exceeds the length of *istring* the whole string is returned. If *start* is specified and *length* equals or exceeds the length of the *istring* counting from the *start* point, the substring from *start* to the end of the string is returned.	"T[*start*,]*length*"
Group (delimited substring) extraction: a substring is extracted from *istring*, based on a specified delimiter character (*d*) found in *istring* that indicates the stopping point. The optional *s* integer specifies the number of delimiters to skip from the beginning of the string before starting the extract. The default is to start at the beginning of the string. The *n* integer specifies the number of delimiters to count in performing the extract. If *n* is larger than the number of *d* delimiters, the extract continues to the end of the string.	"G[*s*]*dn*"
Pattern match extraction: returns an *istring* if it matches the pattern code; otherwise returns the empty string. A pattern code consists of a series of integer/letter pairs. The integer specifies the number of sequential characters to match with a specified character type; a 0 means to match any number (including 0) of characters. The letter specifies the character type: A=alphabetic, N=numeric, X=alphanumeric. These codes are not case-sensitive. You can also specify literal characters in the pattern code string. Some literals, including numbers and parentheses, must be specified enclosed in single quotes. (The MATCH operator provide similar pattern matching support.) You can specify multiple patterns, separated by a semicolon (;) or a slash (/). Each pattern is enclosed in parentheses. **OCONV** returns the *istring* if it matches any of the specified patterns. For example, the following returns dates with either a one-digit or a two-digit month: `OCONV('06/11/2010',"P(2N/2N/4N);(1N/2N/4N)")`	"P(pattern)[;(pattern2)]"

Group (Delimited Substring) Extraction

You can extract a substring from *istring* based on a specified non-numeric delimiter character (*d*) found in *istring*. This delimiter is the stopping point for the extract operation. This delimiter cannot be a number or a dynamic array level

delimiter character (@VM, @FM, etc.). The G conversion code extracts a substring until it encounters the specified delimiter character. The delimiter is not included in the extracted string. If the specified delimiter is not found in *istring*, the entire string is returned, unless you have specified a non-zero value for the optional *s* (skip) argument.

The optional *s* (skip) integer specifies the number of *d* delimiters to skip from the beginning of the string before starting the extract. The default is 0 (zero) which means to start extraction at the beginning of the string. If the specified delimiter is not found in *istring* and the optional *s* (skip) argument is specified as a non-zero integer value, the empty string is returned. If *s* is larger than the number of delimiters in *istring* the empty string is returned.

The *n* integer specifies the number of delimiters to count when extracting the substring. Substring extraction begins at the starting point established by *s* and continues until the specified number of delimiters is reached. If *n* is 1, the extract stops when the first delimiter is encountered. If *n* is 2, the extract stops when the second delimiter is encountered. Intermediate delimiters are included in the substring, but the final delimiter is not. If *n* is larger than the number of delimiters, the extract continues to the end of the string. If *n* is 0, the empty string is returned.

Time and Date Conversion

The following are the conversion codes for time and date conversion. If you specify optional code characters for date or time formatting, these characters must be specified in the order described below.

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. Internal times are specified as the number of seconds elapsed since midnight. Permitted values are 0 (00:00:00) through 86399 (23:59:59); higher numeric values result in an <ILLEGAL VALUE> error. **OCONV** accepts, but truncates, fractional seconds.

Time conversion from internal format to display format.	"MT" returns the local time in 24–hour format `hh:mm`. You can append one or more of the following optional codes (in the following order): H=12 hour clock with AM or PM suffix. P=12 hour clock with AM or PM prefix. S=include seconds. Z=suppress leading zero from hour integer. A single character to be used as the time separator character, replacing the default colons. AM and PM letters are always displayed uppercase.
Date conversion from internal format to display format.	"D" returns the local date in the format `dd MMM yyyy`, where MMM is the three-letter abbreviation for the month name. You can append one or more optional codes in the following sequence: The integers 0, 2, or 4 to specify the number of year digits (the default is 4). A character (such as /, –, or a space) to specify the date separator character; this cause the date to be displayed in an all-numeric format: `mm/dd/yyyy`. The Z=zero suppress code for leading zeros. The E=European format code for all-numeric format returns dates in European format `dd/mm/yyyy`, regardless of your locale (the default is to use the all-numeric date format for the locale). The letter L, to specify that abbreviated month names are displayed in mixed case (the default is all uppercase).
Date conversion from internal format to ODBC display format.	"DS" returns the date in the format `yyyy-mm-dd`. "DMI" returns the date in the format `yyyy mm dd`. You can optionally specify the integers 2 or 4 to specify the number of year digits (the default is 4).
Date conversion from external format to internal format.	"DI" converts a date in display format to internal format. The input date can be in any American all-numeric display format (with optional 2–digit or 4–digit years) or ODBC date format with 4–digit years. If you omit the year portion of the date, "DI" conversion assumes the current year. Refer to the ICONV function for further details on external to internal conversions.
Day extraction: The day of the month is extracted from a date specified in internal format.	"DD" returns the day of the month as a two-digit integer (08). "DDM" returns the day of the month and the month as two integers, separated by a space (August 6 = 06 08). Appending a Z to the code ("DDZ" or "DDMZ") suppresses the leading zero for the day integer.

Month extraction: The month is extracted from a date specified in internal format.	"DM" returns the month as a two-digit integer (08). "DMA" returns the full name of the month in uppercase letters (AUGUST). "DMB" returns the 3–letter abbreviation for the month in uppercase letters (AUG). Appending an L to the code ("DMAL" or "DMBL") returns the month name in mixed case letters. However, letter case of month names and abbreviations is emulation-dependent (see below). "DMR" returns the month as a Roman numeral. Appending a Z to a code that returns the month as an integer suppresses the leading zero for the month integer.
Day of week extraction: The day of the week is extracted from a date specified in internal format.	"DW" returns the day of the week as an integer, counting Monday as day 1. "DWA" returns the full name of the day in uppercase letters (SUNDAY). "DWB" returns the 3–letter abbreviation for the day in uppercase letters (SUN). Appending an L to the code ("DWAL" or "DWBL") returns the day name in mixed case letters. However, letter case of day of week names and abbreviations is emulation-dependent (see below).
Year extraction: The year is extracted from a date specified in internal format.	"DY" (or "DN") returns the year portion of the date (2004). "DY2" returns the year portion as a 2-digit date (04). "DYA" and "DYAL" return the Chinese zodiac animal corresponding to the year.
Days of year extraction: The number of elapsed day in the year is extracted from a date specified in internal format.	"DJ" returns the elapsed days in the year as a 3-digit integer. "DJZ" suppresses the leading zero(s) for the elapsed days integer.
Quarter / Season extraction: The quarter of the year is extracted from a date specified in internal format.	"DQ" returns the quarter of the year as an integer (Jan.-Mar. = 1). "DQA" returns the name of the season in upper case, with Q1 = WINTER. "DQAL" returns the name of the season in mixed case (Winter).

Date Extraction

Date extraction codes can be combined, with the date components displayed in the order specified, separated by a space, and the L or Z appended codes affecting the first specified component. For example, "DMD", "DMAD", "DMBD", "DMADL", and "DMBDL" return the month followed by the day integer. "DMY" returns the month and the year as two integers. "DMAY", "DMAYL", "DMBY", and "DMBYL" return the name of the month and the year. "DMJ" returns the month and the elapsed days in the year as two integers. "DMW" returns the month and the day of the week as two integers. "DWD" returns the day of the week and the day of the month. "DWJ" returns the day of the week and the elapsed days in the year as two integers.

Caché MVBasic displays month names, day of week names, and their abbreviations as either all uppercase ("DMA", "DWA") or lowercase with the first letter capitalized ("DMAL", "DWAL"). In D3, MVBase, R83, POWER95, Ultimate, and UniData emulations, these names and abbreviations are always displayed as lowercase with the first letter capitalized.

Date Conversion

You can display a date in any of the following formats:

- Abbreviated month format, as shown in the following examples:

```
OCONV('16000','D')   returns 21 OCT 2011
OCONV('16000','D2')  returns 21 OCT 11
OCONV('16000','DL')  returns 21 Oct 2011
OCONV('16000','D2L') returns 21 Oct 11
```

- Numeric format. The default day/month order is determined by the current Caché locale. The following examples use American format as the locale default:

```
OCONV('16000','D/')   returns 10/21/2011
OCONV('16000','D2/')  returns 10/21/11
OCONV('16000','D/E')  returns 21/10/2011
OCONV('16000','D2/E') returns 21/10/11
```

- ODBC format, as shown in the following examples:

```
OCONV('16000','DS')   returns 2011-10-21
OCONV('16000','DS2')  returns 11-10-21
OCONV('16000','DMI')  returns 2011 10 21
OCONV('16000','DMI2') returns 11 10 21
```

Note: You can specify the default date format using Caché NLS. Because of operational differences between MV and Caché NLS in the handling of month names, your NLS default date format must represent months as integers.

You can use the DATE function to supply the current date in internal format. The **DATE** and **TIME** functions return internal format values. The **TIMEDATE** function returns external format values.

An internal date is an integer count of days, with 0 representing December 31, 1967. If you specify a fractional number for an internal date, it is truncated to an integer. If you specify a non-numeric value for an internal date, is returned unchanged.

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 CENTURY.PIVOT verb, described in *Operational Differences Between MultiValue and Caché*.

Arithmetic and Logical Operations

The 'A' conversion code can be used for a variety of operations, including arithmetic, equality comparison of numbers or strings, collation order comparison, current date and time arithmetic or equality comparison, and string concatenation. The 'A' conversion code is followed by its operands; 'A' is separated from its operands by either a blank space or a semicolon.

All 'A' operations ignore the *istring* value, and operate on the values following the 'A' code. For example, an addition operation is performed as follows:

```
PRINT OCONV(123,"A '5'+'6'");   ! returns 11
```

Note that the *istring* value must be present but is not used; it could just as easily be an alphabetic string or an empty string placeholder. Also note that numeric literals must be enclosed by delimiters; you may use double quotes ("), single quotes ('), or backslashes (\) to enclose the *code* string. Within the *code* string you may use double quotes (") or single quotes (') as literal delimiters.

'A' conversion code operations only process one multivalue at a time. When using 'A' with a multivalue, you must set up the following @ variables: @RECORD = item to get data from; @ID = item Id to use for attribute 0; @NV = specific multivalue to use; @NS = specific subvalue to use.

The following are supported arithmetic and logical operators:

+	Addition: "A 'n'+'m'"
-	Subtraction: "A 'n'-'m'"
*	Multiplication: "A 'n'*'m'"
/	Division: "A 'n'/'m'". Division is integer division only; neither fractions nor a remainder are returned. Dividing a number by 0 returns 0.
R()	Remainder: "A R('n','m')". Returns the remainder of dividing *n* by *m* as an integer or fractional number. Attempting to divide by zero results in a <DIVIDE> error.
= EQ	Equal to: "A 'n'='m'". Returns either 0 (not equal) or 1 (equal). Can compare numeric or non-numeric strings for equality. Numbers are compared in canonical form; plus signs and leading and trailing zeros are ignored.
# NE	Not Equal to: "A 'n'#'m'". Returns either 0 (equal) or 1 (not equal). Can compare numeric or non-numeric strings for equality. Numbers are compared in canonical form; plus signs and leading and trailing zeros are ignored.
< LT	Less Than: "A 'n'<'m'". Returns either 0 or 1. Compares collation sequence of non-numeric strings.
> GT	Greater Than: "A 'n'>'m'". Returns either 0 or 1. Compares collation sequence of non-numeric strings.
<= LE	Less Than or Equal to: "A 'n'<='m'". Returns either 0 or 1. Compares collation sequence of non-numeric strings.
>= GE	Greater Than or Equal to: "A 'n'>='m'". Returns either 0 or 1. Compares collation sequence of non-numeric strings.

By default, Caché MVBasic order of operations is to perform division, then multiplication, then subtraction, then addition, then equality/inequality tests. You can change this order of operations by using parentheses to nest operations. Note that ObjectScript uses a different order of operations; it uses strict left-to-right evaluation of operators.

You can use the AND and OR logical operators to group multiple equality/inequality operations.

```
PRINT OCONV(123,"A '5'='6' OR '3'<'7'");   ! returns 1
```

You can use IF, THEN, and ELSE functions with an equality/inequality operation to return a user-specified result. You must specify a THEN function (boolean 1). The ELSE function is optional; if ELSE is omitted, boolean 0 returns the empty string.

```
PRINT OCONV(123,"A IF('5'='6') THEN('equal') ELSE('not equal')");   ! returns 'not equal'

PRINT OCONV(123,"A IF('5'='6') THEN('equal')");   ! returns empty string
```

You can use the D and T variables to specify the current date or time in arithmetic or logical operations. Dates and times are represented in internal format as integers:

```
PRINT OCONV(123,"A D+'7'");   ! returns the current date plus 7 days
```

This example returns a value such as 15622.

Caché MVBasic provides additional operators, as described in the Operators page of this manual.

String Concatenation and Delimiter Insertion

You can use either the 'A' code or the 'C' code for string concatenation. These codes can concatenate multiple substrings into a single string. In both cases, the concatenation operands follow the letter code.

'A' Code Concatenation

'A' is used for simple concatenation, using the colon character. For example, the following returns ‘quickbrownfox’:

```
PRINT OCONV(123,"A;'quick':'brown':'fox'")
```

Although the *istring* is required, it is always ignored by the 'A' conversion code. The *istring* can be any value, including an empty string.

'C' Code Concatenation with Insertion

'C' is used for concatenation that inserts a single-character delimiter. For example, the following returns ‘quick^brown^fox’:

```
PRINT OCONV(123,"C;'quick'^'brown'^'fox'")
```

'C' is also used for concatenation that inserts the *istring* value as specified by the asterisk wildcard.

The character following the 'C' code must be a semicolon; you cannot separate 'C' from its operands with a blank space.

Although the *istring* is required, the 'C' code ignores the *istring* value unless you specify an asterisk wildcard (as described below). The *istring* can be any value, *except* an empty string.

The 'C' code can specify the same single-character delimiter, or different single-character delimiters between items. For example, the following returns ‘quick^brown*fox’:

```
PRINT OCONV(123,"C;'quick'^'brown'*'fox'")
```

You can specify almost any single character, including the blank space, as a delimiter when concatenating substrings into a string. You can specify a string delimiter (single quote, double quote, or backslash) if that delimiter is not already in use. For example, the following returns ‘quick"brown"fox’:

```
PRINT OCONV(123,\C;'quick'"'brown'"'fox'\)
```

The semicolon is a special case. The semicolon is used to specify concatenation with no inserted delimiter. For example, the following returns ‘quick^brownfox’:

```
PRINT OCONV(123,"C;'quick'^'brown';'fox'")
```

The colon, which elsewhere in MultiValue is a concatenation operator, is here simply a literal character. For example, the following returns ‘quick:brownfox’:

```
PRINT OCONV(123,"C;'quick':'brown';'fox'")
```

The 'C' code can insert the *istring* value in to the returned string by using an asterisk as a substring wildcard. An asterisk can be used as either an inserted delimiter or a wildcard.

The following example uses the asterisk as an inserted delimiter. It returns ‘quick*fox*dog’:

```
PRINT OCONV('red',"C;'quick'*'fox'*'dog'")
```

The following example uses blank spaces as the inserted delimiters, and uses the asterisk as a wildcard for the *istring* value. It returns ‘quick red fox red dog’:

```
PRINT OCONV('red',"C;'quick' * 'fox' * 'dog'")
```

You can use an asterisk wildcard to specify the substring delimiter in *istring*. This wildcard delimiter can be a single character or a multiple character string. It can be a dynamic array delimiter variable, such as @FM. It can even be a semicolon.

The following example uses asterisks to specify the substring delimiter as a wildcard, with its value supplied by *istring*. It returns ‘quick^^brown^^fox’:

```
PRINT OCONV('^^',"C;'quick';*;'brown';*;'fox'")
```

The following example uses asterisks to specify the substring delimiter as a wildcard, with a semicolon as the *istring* value. It returns ‘quick;brown;fox’:

```
PRINT OCONV(';',"C;'quick';*;'brown';*;'fox'")
```

The following example uses asterisks to specify the substring delimiter as a wildcard, with @FM (the field mark delimiter) as the *istring* value:

```
PRINT OCONV(@FM,"C;'quick';*;'brown';*;'fox'")
```

Dynamic Array Element Extraction

You can use the 'ZVn' code to extract a single value mark element from a dynamic array. Here *n* is a positive integer specifying the position of the element in the dynamic array, counting from 1. Specifying a *n* value of 0 returns the entire dynamic array. Specifying a *n* value larger than the number of elements in the dynamic array returns the null string.

The following program returns the string “Barney”:

```
x="Fred":@VM:"Barney":@VM:"Wilma"
CRT OCONV(x,"ZV2")
```

Implicit Formatting

Many **OCONV** conversion codes are identical to FMT formatting codes. You can, therefore, perform many of the **OCONV** conversions using implicit formatting. Implicit formatting simply specifies the *code* value as the second argument in a CRT, DISPLAY, or PRINT statement. It is functionally identical to explicit formatting using the **FMT** function. For example, the following are equivalent date conversions:

```
PRINT OCONV(14100,"D");    ! "08 AUG 2006"
PRINT 14100 "D";           ! "08 AUG 2006"
PRINT FMT(14100,"D");      ! "08 AUG 2006"
```

When there is a difference between **OCONV** conversion and **FMT** formatting, implicit formatting behaves like **FMT**:

- Because the letters "L" and "R" are used as FMT formatting codes, the **OCONV** length ("L", "Ln", "Ln-m" or "Ln,m") and range ("Rn,m") conversion codes cannot be used for implicit formatting.
- The "MLn" and "MRn" conversion codes are not the same as the corresponding format codes. In **OCONV** these conversion codes move the decimal point the specified number of digits. In **FMT** (and implicit formatting) these codes append the specified number of fractional digits. Thus:

  ```
  PRINT OCONV(12345,"ML2");    ! "123.45"
  PRINT 12345 "ML2";           ! "12345.00"
  PRINT FMT(12345,"ML2");      ! "12345.00"
  ```

  The "MDn" code is the same for both **OCONV** and **FMT**. Thus:

  ```
  PRINT OCONV(12345,"MD2");    ! "123.45"
  PRINT 12345 "MD2";           ! "123.45"
  PRINT FMT(12345,"MD2");      ! "123.45"
  ```

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, for example "20 APR 2011"
  PRINT OCONV(@DATE,"D");   ! current date, for example "20 APR 2011"
  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")         ! "00:00"
  PRINT OCONV(TIME(),"MT")    ! current time, for example "21:02"
  PRINT OCONV(TIME(),"MTH")   ! current time, for example "09:02PM"
  PRINT OCONV(TIME(),"MTS")   ! current time, for example "21:02:41"
  PRINT OCONV(TIME(),"MTS.")  ! current time, for example "21.02.41"
  PRINT OCONV(TIME(),"MTHS*") ! current time, for example "09*02*41PM"
```

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
```

The following example shows decimal-to-hex and hex-to-decimal conversions. It shows both the **OCONV** conversions and the inverse **ICONV** conversions:

```
HexConversions:
  PRINT OCONV(10,"MCXD");     ! Returns 16
  PRINT OCONV(10,"MCDX");     ! Returns A

  PRINT ICONV(10,"MCXD");     ! Returns A
  PRINT ICONV(10,"MCDX");     ! Returns 16
```

The following example shows character-to-code and code-to-character conversions. It shows both the **OCONV** conversions and the inverse **ICONV** conversions:

```
CharConversions:
  PRINT OCONV("mnop","MCAX");     ! Returns 6D6E6F70
  PRINT OCONV("6D6E6F70","MCXA"); ! Returns mnop

  PRINT ICONV("mnop","MCXA");     ! Returns 6D6E6F70
  PRINT ICONV("6D6E6F70","MCAX"); ! Returns mnop
```

The following example shows masked decimal conversions with moving of the decimal point, rounding or truncation to the specified number of fractional digits, and zero padding when needed. The first integer (n) is number of fractional digits, the second integer (m) is the leftward shift of the decimal point, the third integer (k or #k) is a returned character count mask.

```
MaskedDecimalConversions:
  PRINT OCONV("123456","MD2");      ! Returns 1234.56  n=2 m=n by default
  PRINT OCONV("123456","MD12");     ! Returns 1234.6   n=1 m=2 round by default
  PRINT OCONV("123456","MD12T");    ! Returns 1234.5   n=1 m=2 T=truncate
  PRINT OCONV("123456","MD12T$");   ! Returns $1234.5  n=1 m=2 $=append dollar sign
  PRINT OCONV("123456","MD12T$,");  ! Returns $1,234.5 n=1 m=2 ,=add numeric group separator(s)
  PRINT OCONV("123456","MD12T$#3"); ! Returns 4.5      n=1 m=2 #k=3 char mask
  PRINT OCONV("123456","MD37");     ! Returns 0.012    n=3 m=7 zero padding
  PRINT OCONV("123456","MD374");    ! Returns .012     n=3 m=7 k=4 char mask
```

The following example shows length conversions:

```
LengthConversions:
  PRINT OCONV("abcd","L");     ! Returns 4
  PRINT OCONV("abcd","L0");    ! Returns 4
  PRINT OCONV("abcd","L4");    ! Returns "abcd"
  PRINT OCONV("abcd","L3");    ! Returns empty string
  PRINT OCONV("abcd","L5");    ! Returns empty string
  PRINT OCONV("abcd","L1-4");  ! Returns "abcd"
  PRINT OCONV("abcd","L0-5");  ! Returns "abcd"
  PRINT OCONV("abcd","L4-6");  ! Returns "abcd"
  PRINT OCONV("abcd","L4-4");  ! Returns "abcd"
  PRINT OCONV("abcd","L5-9");  ! Returns empty string
```

The following example shows pattern match extraction. It returns the input string if it matches the specified pattern. These examples perform pattern matches on 7 and 10 digit telephone numbers specified in the following commonly used formats: “123–4567”, "617-123-4567", and "(617) 123-4567":

```
PatternMatch:
  PRINT OCONV(telnum,"P(3n-3n-4n)")
    ! matches and returns 10-digit numbers
    ! in hyphen format
  PRINT OCONV(telnum,"P(3n-4n);(3n-3n-4n)")
    ! matches and returns either 7-digit or 10-digit numbers
    ! in hyphen format.
  PRINT OCONV(telnum,"P(3n-3n-4n);('('3n')' 3n-4n)")
    ! matches and returns 10-digit numbers
    ! in either hyphen or parentheses formats.
  PRINT OCONV(telnum,"P('617'-3n-4n);('(617)' 3n-4n)")
    ! matches and returns 10-digit numbers
    ! in either hyphen or parentheses formats
    ! that begin with the 617 area code.
```

See Also

- ICONV function
- OCONVS function
- FMT function
- STATUS function
- DATE function
- TIME function
- TIMEDATE function
- DOWNCASE function
- UPCASE function
- Strings
- MATCH Pattern Match operator

OCONVS

Converts dynamic array element values from internal format to external format.

```
OCONVS(idynarray,code)
```

Arguments

idynarray	An expression that resolves to a dynamic array, each element of which specifies a value in internal (storage) format.
code	An expression that resolves to a conversion code string. This conversion code specifies the type of conversion to perform. Conversion is from internal format to external format. Conversion codes are not case-sensitive. For descriptions of these *code* values, refer to the OCONV function. For a complete list of conversion codes, refer to the Conversion Codes table in the *MultiValue Basic Quick Reference*.

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. The type of conversion is specified by a *code* string that is specific to the type of data to be converted.

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. Note that the MCDX/MCXD, MCAX/MCXA, and MCWX/MCXW *code* pairs have the opposite meanings in **ICONVS**, reversing the **OCONVS** operation.

You can use the **OCONV** function to perform conversions for a single value. For available *code* values, further details on conversions, and additional program examples, refer to the OCONV function.

Examples

The following example shows date conversions:

```
DateConversions:
  x=14143:@VM:14144:@VM:14145
  PRINT OCONVS(x,"D")
  ! Returns 20 SEP 2006ý21 SEP 2006ý22 SEP 2006
```

The following example shows character-to-hexcode and hexcode-to-character conversions. It shows both the **OCONVS** conversions and the inverse **ICONVS** conversions:

```
CharConversions:
  odyn="m":@VM:"n":@VM:"o":@VM:"p"
  idyn=OCONVS(odyn,"MCAX")
  PRINT idyn;                ! Returns 6Dý6Eý6Fý70
  PRINT OCONVS(idyn,"MCXA"); ! Returns mýnýoýp

  PRINT ICONVS(odyn,"MCXA");    ! Returns 6Dý6Eý6Fý70
  PRINT ICONVS(idyn,"MCAX");    ! Returns mýnýoýp
```

See Also

- OCONV function
- ICONVS function
- STATUS function
- DATE function

- TIME function
- TIMEDATE function
- DOWNCASE function
- UPCASE function
- Dynamic Arrays
- MATCH Pattern Match operator

$ORDER ($O)

Returns the next local variable or the subscript of a local or global variable.

```
$ORDER(variable,direction,target)
$O(variable,direction,target)
```

Parameters

variable	Either a local variable or a subscripted local, global, or process-private global variable. If an array, the subscript is required. You cannot specify just the array name. *variable* may be specified as a variable or an object property with the syntax obj->property.
direction	*Optional* — The subscript order in which to traverse the target array. An expression that resolves to either: 1 = ascending subscript order (the default) or -1 = descending subscript order. For unsubscripted local variables, 1 (the default) is the only permitted value.
target	*Optional* — Returns the current data value of the next or previous node of *variable*. Whether it is the next or previous depends on the setting of *direction*. You must specify a *direction* value to specify a *target*. For unsubscripted local variables, *direction* must be set to 1. If *variable* is undefined, the *target* value remains unchanged. *target* may be specified as a variable or an object property with the syntax obj->property. The *target* parameter cannot be used with structured system variables (SSVNs) such as **^$ROUTINE**.

Description

The value **$ORDER** returns depends on the parameters used.

- **$ORDER**(*variable*) returns the number of the next defined subscript if *variable* is a subscripted variable. The returned subscript is at the same level as that specified for the variable. For example, $ORDER(^client(4,1,2)) returns the next subscript (3), assuming that ^client(4,1,3) exists.

  $ORDER(*variable*) returns the name of the next defined local variable in alphabetic collating sequence, if *variable* is an unsubscripted local variable. For example, **$ORDER** would return the following defined local variables in the following sequence: a, a0a, a1, a1a, aa, b, bb, c.

- **$ORDER**(*variable*,*direction*) returns either the next or the previous subscript for the variable. You can specify *direction* as 1 (next, the default) or –1 (previous).

  For unsubscripted local variables, **$ORDER** returns variables in *direction* 1 (next) order only; you cannot specify a *direction* of –1 (previous).

- **$ORDER**(*variable*,*direction*,*target*) returns the subscript for the variable, and sets *target* to its current data value. This can be either the next or the previous subscript for a subscripted variable, depending on the *direction* setting. For an unsubscripted local variable, *direction* must be set to 1 to return the current data value to *target*. The *target* parameter cannot be used with structured system variables (SSVNs) such as ^**$ROUTINE**.

Examples

The following example lists the name and value of the next defined global variable subscript after variable ^fruit(1):

```
^fruit(1)="apple"
^fruit(2)="orange"
^fruit(3)="pear"
^fruit(4)="banana"
PRINT $ORDER(^fruit(1),1,target)
PRINT target
```

The following example returns 1, the first subscript in ^X. It sets the naked indicator to the first level.

```
^x(1,2,3)="1"
^x(2)="2"
PRINT $ORDER(^x(-1))
```

The following example returns 2, the next subscript on the single subscripted level. (The node you specify in the argument need not exist.) The naked indicator is still set to the first level.

```
^x(1,2,3)="1"
^x(2)="2"
PRINT $ORDER(^x(1))
```

The following example returns 2, the first subscript on the two-subscript level. The naked indicator is now set at the second level.

```
^x(1,2,3)="1"
^x(2)="2"
PRINT $ORDER(^x(1,-1))
```

The following example uses **$ORDER** to list all of the primary subscripts in the ^data global:

```
^data(1)="a"
^data(3)="c"
^data(7)="g"
   ! Get first subscript
key=$ORDER(^data(""))
   WHILE (key'="") {
   PRINT key,!
     ! Get next subscript
   key=$ORDER(^data(key))
   }
```

In the following example, a multidimensional property is used as the *variable* value. This example returns the names of defined namespaces to the *target* parameter:

```
obj = "%ResultSet"->%New("%SYS.Namespace:List")
obj->Execute()
obj->Next()
rtn = $ORDER(obj->Data(x),1,val)
crt rtn        ! returns level "Nsp"
crt val        ! returns namespace name
obj->Next()
rtn = $ORDER(obj->Data(x),1,val)
crt val        ! returns next namespace name
obj->Next()
rtn = $ORDER(obj->Data(x),1,val)
crt val        ! returns next namespace name
```

Similar programs return the same information using the $GET and $DATA functions.

Notes

Uses for $ORDER

$ORDER is typically used with loop processing to traverse the nodes in a sparse array. A sparse array is an array that may contain undefined nodes on any given level. Unlike the **$DATA** function, **$ORDER** simply skips over undefined nodes to return the subscript of the next existing node. For example:

```
struct=""
   FOR  {
     struct=$ORDER(^client(struct))
     QUIT:struct=""
  PRINT ^client(struct)
   }
```

The above routine writes the values for all the top-level nodes in the ^client global array.

$ORDER skips over undefined nodes, but not nodes that contain no data. Such nodes include both pointer nodes and terminal nodes. If you use **$ORDER** in a loop to feed a command (such as **PRINT**) that expects data, you must include a **$DATA** check for dataless nodes.

Start and End for a Search

To start a search from the beginning of the current level, specify a null string ("") for the subscript. This technique is required if the level may contain negative as well as positive subscripts. The following example returns the first subscript on the array level:

```
s=$ORDER(^client(""))
PRINT s
```

When **$ORDER** reaches the end of the subscripts for the given level, it returns a null string (""). If you use **$ORDER** in a loop, your code should always include a test for this value.

$ORDER Uses Naked Global Reference

Like the **$NAME** and **$QUERY** functions, **$ORDER** can be used with a naked global reference, which is specified without the array name and designates the most recently executed global reference. For example:

```
var1=^client(4,5)
var2=$ORDER(^(""))
PRINT "var1=",var1,"var2=",var2
```

The first **SET** command establishes the current global reference, including the subscript level for the reference. The **$ORDER** function uses a naked global reference to return the first subscript for this level. For example, it might return the value 1, indicating ^client(4,1).

For more details, see Naked Global Reference in *Using Caché Globals*.

$ORDER and $NEXT

$ORDER is similar to **$NEXT**. Both functions return the subscripts of the next sibling in collating order to the specified node. However, **$ORDER** and **$NEXT** have different start and failure codes, as follows:

	$NEXT	$ORDER
Starting point	-1	Null string
Failure code	-1	Null String

Because **$ORDER** starts and fails on the null string, it correctly returns nodes having both negative and positive subscripts.

See Also

- $DATA function
- $GET function
- Global Structure chapter in *Using Caché Globals*

ORS

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

```
ORS(dynarray1,dynarray2)
```

Arguments

dynarray	An expression that resolves to a dynamic array of boolean values.

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 missing, a null string, or a string containing any non-numeric value, **ORS** parses its value as 0.

If the two dynamic arrays have different numbers of elements, the returned dynamic array has the number of elements of the longer dynamic array. By default, the shorter dynamic array is padded with 0 value elements for the purpose of the logical comparison. You can also use the REUSE function to define behavior when specifying two dynamic arrays with different numbers of elements.

Caché MVBasic also supports the logical operators ! and OR.

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 1ý1ý1ý0
```

The following example performs an OR test on two dynamic arrays of different lengths:

```
a=1:@VM:0:@VM:1:@VM:0
b=1:@VM:1:@VM:1:@VM:1:@VM:1:@VM:0
PRINT ANDS(a,b)
      ! returns 1ý1ý1ý1ý1ý0
```

See Also

- ANDS function
- NOTS function
- Dynamic Arrays
- Operators

PWR

Returns a number raised to a power.

```
PWR(num,exponent)
```

Arguments

num	The base number. An expression that resolves to a numeric, specified either as a number or as a numeric string. If *num* is 0, *exponent* must be non-negative. If *num* is negative, *exponent* must be an integer.
exponent	The exponent. An expression that resolves to a numeric, specified either as a number or as a numeric 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 non-zero *num* raised to an *exponent* of 0 returns 1. If *num* and *exponent* are both 0, **PWR** returns 0. If *num* is 0 and *exponent* is a negative number, **PWR** generates an <ILLEGAL VALUE> error. If *num* is a negative number and *exponent* is a fractional number, **PWR** generates an <ILLEGAL VALUE> error.

Very large positive *exponent* values (such as **PWR(9,153)**) or very small *num* values with a negative *exponent* (such as **PWR(.00005,-30)**) may result in an overflow, generating a <MAXNUMBER> error. Very large negative *exponent* values (such as **PWR(9,-135)**) or very small *num* values with a positive *exponent* (such as **PWR(.00005,30)**) may result in an underflow, returning 0.

The same operation can be performed using the exponentiation operator: **. To perform exponentiation on the elements of a dynamic array, use the **PWRS** function.

See Also

- PWRS function
- SQRT function
- FMUL function
- SMUL function
- Operators

PWRS

Returns the elements of a dynamic array raised to a power.

```
PWRS(dynarray,exponents)
```

Arguments

dynarray	An expression that resolves to a dynamic array of numeric values. Each element in *dynarray* is raised to the exponent in the corresponding element of the *exponents* dynamic array. If an element value is 0, the corresponding *exponents* element must be non-negative. If an element is negative, the corresponding *exponents* element must be an integer.
exponents	The exponents to apply to the *dynarray* elements. An expression that resolves to a dynamic array of numeric values.

Description

The **PWRS** function raises each element of *dynarray* to the power specified by the corresponding element of the *exponents* dynamic array. 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.

If the two dynamic arrays have different numbers of elements, by default the shorter dynamic array) is padded so that the returned dynamic array has the number of elements of the longer dynamic array. If the shorter dynamic array is the value to be raised, it is padded with the required number of elements with the value of 0. If the shorter dynamic array is the exponents, it is padded with the required number of elements with the value of 1. You can also use the REUSE function to define behavior when specifying two dynamic arrays with different numbers of elements.

Note: In the *exponents* dynamic array, missing elements and elements with a null string or non-numeric string value are parsed as 1. This parsing differs from the **PWR** function and other MultiValue numeric functions and operators, which parse non-numerics as 0.

Any non-zero numeric element value raised to an exponent of 0 returns 1. If the corresponding *dynarray* and *exponents* elements are both 0, **PWRS** returns 0 for that element. If the *dynarray* element is 0 and the corresponding *exponents* element is a negative number, **PWRS** generates an <ILLEGAL VALUE> error. If the *dynarray* element is a negative number and the corresponding *exponents* element is a fractional number, **PWRS** generates an <ILLEGAL VALUE> error.

A very large positive *exponents* element value (such as 135), or a very small *dynarray* element value (such as .00005) with a corresponding negative *exponents* element (such as -30) may result in an overflow, generating a <MAXNUMBER> error. A very large negative *exponents* element value (such as -135), or a very small *dynarray* element value (such as .00005) with a corresponding positive *exponents* element (such as 30) may result in an underflow, returning 0.

To return the exponent of a single value raised to a power, use the PWR function or the ** operator.

Example

The following example returns the value of each element in *mynums* raised to the corresponding element in *myexps*:

```
mynums=1:@VM:2:@VM:3:@VM:4:@VM:5
myexps=2:@VM:3:@VM:2:@VM:3:@VM:2
crt PWRS(mynums,myexps)
   ! returns: 1ý8ý9ý64ý25
```

See Also

- PWR function
- MULS function
- SQRT function
- Operators

QUOTE

Encloses a value in double quotation marks.

```
QUOTE(string)
```

Arguments

string	An expression that resolves to a string or a number. *string* may be a dynamic array.

Description

The **QUOTE** function returns *string* enclosed in double quotation marks. The quotation marks are part of the resulting string. Therefore, using **QUOTE** increases the length of *string* by 2 characters. If *string* is the null string ("") **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
```

The following example uses the **QUOTE** function to enclose a string in double quotation marks:

```
str1 = "Hello"
str2 = 'Hello'
str3 = \Hello\
PRINT str1:str2:str3;  ! Returns HelloHelloHello
PRINT LEN(str1),LEN(str2),LEN(str3);  ! Returns 5   5   5
q1 = QUOTE(str1)
q2 = QUOTE(str2)
q3 = QUOTE(str3)
PRINT q1:q2:q3;       ! Returns "Hello""Hello""Hello"
PRINT LEN(q1),LEN(q2),LEN(q3);    ! Returns 7    7    7
```

Note that the quote marks are not simply string delimiters, but are part of the returned string.

See Also

- DQUOTE function
- SQUOTE function
- LEN function
- PRINT statement

RAISE

Raises dynamic array delimiters to next level.

```
RAISE(dynarray)
```

Arguments

dynarray	An expression that resolves to a 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. **RAISE** returns a non-dynamic array value 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 123ü456ü10ü11
numraise = RAISE(numsm)
PRINT numraise;      ! Returns 123ý456ý10ý11
numlower = LOWER(numraise)
PRINT numlower;      ! Returns 123ü456ü10ü11
```

See Also

- LOWER function
- Dynamic Arrays

RECORDLOCKED

Returns the lock status for a record or file.

```
RECORDLOCKED(filevar,recID)
```

Arguments

filevar	A file variable name used to refer to a MultiValue file. This *filevar* is supplied by the **OPEN** statement. *filevar* must be specified as a literal.
recID	The record ID of the record to be checked for lock status. An expression that resolves to an integer.

Description

The **RECORDLOCKED** function returns an integer code specifying the lock status of the specified record or file. The following are the return codes:

3	This user has a file lock (shared or exclusive).
2	This user has an update record lock.
1	This user has a shared record lock.
0	This record is not locked.
-1	Another user has a shared record lock.
-2	Another user has an update record lock.
-3	Another user has a file lock (shared or exclusive).

If the specified file has both a file lock and a record lock, **RECORDLOCKED** returns the record lock code.

Invoking the **RECORDLOCKED** function sets the **STATUS** function to the job number of the job that holds the lock. If the lock is a shared lock, **STATUS** returns the job number of the first job found. If the current user shares the lock with another user, **STATUS** returns the other user's job.

File and Record Locking and Unlocking

To lock a file, use the FILELOCK statement. To unlock a file, use the FILEUNLOCK statement.

To lock a record, use the RECORDLOCKU (update lock) or RECORDLOCKL (shared lock) statement. To unlock a record, use the RELEASE statement.

See Also

- OPEN statement
- STATUS statement
- STATUS function

REM

Remainder after integer division of two values.

```
REM(numstr1,numstr2)
```

Arguments

numstr1	The dividend. An expression that resolves to a number or numeric string.
numstr2	The divisor. An expression that resolves to a non-zero number 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 the null string or a non-numeric value, **REM** parses its value as 0 (zero).

You cannot divide a number by 0. Attempting to do so results in a <DIVIDE> error.

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

dynarray	An expression that resolves to a dynamic array from which successive data values are to be extracted.
delimcode	A local variable used to receive an integer code for the dynamic array delimiter type. *delimcode* cannot be a global variable or a subscripted variable.

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":@FM:"Wilma":@VM:"Betty"
FOR x=1 TO 5
   PRINT REMOVE(names,lvl)
   PRINT lvl
   ! Returns:
   !   Fred
   !   3
   !   Barney
   !   2
   !   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

dynarray	An expression that resolves to a dynamic array.
f	An expression that resolves to an integer specifying the Field level of the dynamic array from which to access the data. Fields are counted from 1.
v	*Optional* — An expression that resolves to an integer specifying the Value level of the dynamic array from which to access the data. Values are counted from 1 within a Field.
s	*Optional* — An expression that resolves to an integer specifying the Subvalue level of the dynamic array from which to access the data. Subvalues are counted from 1 within a Value.
replacement	An expression that resolves to a data value used to replace the element data value specified by *f*, *v*, and *s*. Note the semicolon (;) that precedes *replacement*; if *f*, *v*, and *s* are all specified, you can precede *replacement* with a comma, otherwise you must use a semicolon.

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.

Note that a semicolon (;) is used before *replacement* as an argument separator. This is because the *v* and *s* arguments are optional and can be omitted.

If *replacement* is the null string, **REPLACE** removes the current data value (replaces it with a null string), but does not remove the level delimiter character.

If lower level delimiters exist in *dynarray*, setting an upper level to 0, the null string, or a non-numeric value is equivalent to setting it to 1.

If lower level delimiters do not exist in *dynarray*, setting this nonexistent lower level to 1, 0, the null string, or a non-numeric value 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 replaces 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")
```

The following example replaces the second value with an empty string:

```
cities="New York":@VM:"London":@VM:
"Chicago":@VM:"Boston":@VM:"Los Angeles"
PRINT REPLACE(cities,1,2,0,"")
```

Emulation

UniData systems differ in how they handle *f*, *v*, and *s* arguments set to 0. The $OPTIONS ATTR.0IS1 (“zero is one”) provides support for this UniData feature. UniData systems ignore *v* and *s* arguments that are set to a negative number.

See Also

- REMOVE statement
- EXTRACT function
- Dynamic Arrays

REUSE

Reuses a value when comparing two dynamic arrays of different lengths.

```
REUSE(dynarray)
```

Arguments

dynarray	An expression that resolves to a 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.

REUSE can be used with the following MVBasic functions: ADDS (addition), SUBS (subtraction), MULS (multiplication), DIVS (division), MODS (modulo division), PWRS (exponentiation), EQS (equal to), NES (not equal to), GTS (greater than), GES (greater than or equal to), LTS (less than), LES (less than or equal to), CATS (concatenate), SPLICE (concatenate with delimiter), ANDS (logical AND), and ORS (logical OR).

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.

Emulation

INFORMATION, jBASE, PIOpen, Prime, and UniData set **$OPTIONS VEC.MATH**. When the $OPTIONS VEC.MATH is set, **REUSE** can use operator symbols to perform the five basic arithmetic operations on dynamic arrays. The + operator is equivalent to the ADDS function. The – operator is equivalent to the SUBS function. The * operator is equivalent to the MULS function. The / operator is equivalent to the DIVS function. The ** operator is equivalent to the PWRS function. These operators perform vector arithmetic when supplied dynamic array arguments, and perform simple arithmetic operations when supplied numeric arguments.

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

- Dynamic Arrays

RIGHT

Returns a specified number of characters from the right end of a string.

```
RIGHT(string,length)
```

Arguments

string	An expression that resolves to a string from which the rightmost characters are returned.
length	An expression that resolves 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

number	An expression that resolves to an integer, 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 fractional 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 or the empty string (""), it is parsed as 0, and thus **RND** always returns 0.

Examples

The following example generates twenty random numbers in the range 0 through 99 (inclusive):

```
FOR x=1 TO 20
  PRINT RND(100)
NEXT
```

See Also

- RANDOMIZE statement

ROUND

Rounds a number.

```
ROUND(num[,precision])
```

Arguments

num	An expression that resolves to a number or numeric string.
precision	*Optional* — An expression that resolves to an integer specifying the number of decimal digits to round to. If omitted, rounds to an integer.

Description

The **ROUND** function returns a number rounded to the specified number of decimal digits. The number 5 is always rounded up. If *precision* is not specified, or is specified as 0, a negative number, or an non-numeric string, **ROUND** rounds *num* to an integer. If *precision* is specified, only those digits that were present in *num* can be returned. The *precision* argument specifies the maximum number of fractional digits to be returned; **ROUND** does not perform zero-padding of fractional digits.

For numerics, prior to rounding MVBasic performs all arithmetic operations and converts numbers to canonical form, removing leading and trailing zeroes, a trailing decimal point, and all signs except a single minus sign. For this reason, input trailing zeros are not returned, but decimal digits rounded to trailing zeros are returned.

Examples

The following examples use the **ROUND** function to return a number rounded to an integer:

```
PRINT ROUND(123.4);         ! Returns 123
PRINT ROUND(123.5);         ! Returns 124
PRINT ROUND(123.4,0);       ! Returns 123
PRINT ROUND(123.999,0);     ! Returns 124
PRINT ROUND(123,-1);        ! Returns 123
```

The following examples use the **ROUND** function to return a number rounded to the specified number of decimal digits. Note that trailing zeros are only returned when they are the result of the rounding operation:

```
PRINT ROUND(1.234,2);       ! Returns 1.23
PRINT ROUND(1.235,2);       ! Returns 1.24
PRINT ROUND(1.000,2);       ! Returns 1
PRINT ROUND(1.100,2);       ! Returns 1.1
PRINT ROUND(1.999,2);       ! Returns 2.00
PRINT ROUND(1.999,3);       ! Returns 1.999
```

See Also

- FMT function
- FMTS function
- LEN function

SADD

Adds two numeric strings.

```
SADD(numstr1,numstr2)
```

Arguments

numstr	An expression that resolves to a number 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 the null string 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

num1	An expression that resolves to a number or a numeric string.
num2	An expression that resolves to 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 the null string 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

numstr1	The dividend. An expression that resolves to a number or numeric string.
numstr2	The divisor. An expression that resolves to a non-zero numeric or numeric string.

Description

The **SDIV** function divides *numstr1* by *numstr2* and returns the quotient. 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.

Attempting to divide by zero generates a <DIVIDE> error, ending execution of the function and invoking an error trap handler, if available.

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 <DIVIDE> error:

```
PRINT SDIV(7,0)
PRINT SDIV("7","")
PRINT SDIV("7","three")
```

See Also

- FDIV function
- DIVS function
- DIV function
- MOD function
- MODS
- Operators

SELECTINFO

Returns information about a select list.

```
SELECTINFO(listnum,key)
```

Arguments

listnum	An expression that resolves to an integer between 1 and 10 (inclusive) that identifies a select list. This value is defined in the **SELECT** statement. *listnum* 0 is not valid for Caché MVBasic.
key	An expression that resolves to an integer code indicating what select list information to return. The available values are 1 (active), and 3 (count).

Description

The **SELECTINFO** function returns different values depending on the value of *key*:

- If *key* is 1, **SELECTINFO** returns a boolean value, indicating whether the specified select list is active. 1=select list is active; 0=select list is inactive. A **SELECT** command activates a select list. When a **READNEXT** attempts to read past the last item of the select list, the list is inactivated.
- If *key* is 3, **SELECTINFO** returns an integer value, specifying the total number of items in the select list. It returns 0 if the select list is not active, or if an active select list does not contain any items.

Examples

The following example uses the **SELECTINFO** function to return information about select list 4:

```
slist=4
PRINT SELECTINFO(slist,1)
PRINT SELECTINFO(slist,3)
```

See Also

- SELECT command

SENTENCE

Returns the command line that invoked the current process.

```
SENTENCE([n])
```

Arguments

n	*Optional* — An expression that resolves to an integer used to specify what portion of the command line to return.

Description

The **SENTENCE** function returns the most recently issued command line. It returns all portions of the command line exactly as specified, including any user-specified comments.

SENTENCE is commonly invoked with no arguments to return the entire command line. The parentheses are mandatory. **SENTENCE** with no arguments is functionally identical to the @SENTENCE system variable.

You can specify an optional integer argument to limit the value returned to only one portion of the command line. An *n* value of 0 returns the initial command. For example, `PRINT SENTENCE(0)` returns “PRINT”. An *n* value of 1 returns the first command argument. For example, `PRINT SENTENCE(1)` returns “SENTENCE(1)”. Higher values of *n* return subsequent arguments and commands on the command line. A value of *n* larger than the number of commands and arguments on the command line returns the empty string. This optional argument is provided for compatibility with jBASE.

Examples

The following examples use the argumentless form of **SENTENCE** to return the full command line:

```
USER:;PRINT SENTENCE()
  ! Returns:
  ! ;PRINT SENTENCE()

USER:;PRINT ABS(4-7),SENTENCE(); ! comment
  ! Returns:
  ! 3       ;PRINT ABS(4-7),SENTENCE(); ! comment

USER:;PRINT SENTENCE(),"hello world"; ! comment
  ! Returns:
  ! ;PRINT SENTENCE(),"hello world"; ! comment      hello world
```

The following examples use the **SENTENCE** *n* argument to return a single portion of the command line:

```
USER:;PRINT ABS(4-7):"cheers",SENTENCE(0); ! comment
  ! Returns:
  ! 3cheers    ;PRINT
USER:;PRINT ABS(4-7):"cheers",SENTENCE(1); ! comment
  ! Returns:
  ! 3cheers   ABS(4-7):
USER:;PRINT ABS(4-7):"cheers",SENTENCE(2); ! comment
  ! Returns:
  ! 3cheers   cheers
USER:;PRINT ABS(4-7):"cheers",SENTENCE(3); ! comment
  ! Returns:
  ! 3cheers ,SENTENCE(3);
USER:;PRINT ABS(4-7):"cheers",SENTENCE(4); ! comment
  ! Returns:
  ! 3cheers !
USER:;PRINT ABS(4-7),"cheers",SENTENCE(5); ! comment
  ! Returns:
  ! 3cheers comment
USER:;PRINT ABS(4-7),"cheers",SENTENCE(6); ! comment
  ! Returns:
  ! 3cheers
USER:
```

See Also

- @SENTENCE system variable

SEQ

Returns the character code corresponding to a specified character.

```
SEQ(char)
```

Arguments

char	An expression that resolves to a character or string. If *char* is a string, **SEQ** returns the value of the first character.

Description

The **SEQ** function takes a character and returns the corresponding character code, a base-10 integer value. Its inverse, the **CHAR** function takes a numeric code and returns the corresponding character.

If *char* is the null string, **SEQ** returns -1. If *char* is a string the first character of which is either a space or a tab, **SEQ** returns 32.

The Caché MVBasic **SEQ** function returns the numeric value for a single character. The corresponding 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

dynarray	An expression that resolves to a 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 is missing or contains the null string, **SEQS** returns -1 for that element. If an element is a string the first character of which is either a space or a tab, **SEQS** returns 32.

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 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 65ý66ý67ý68
```

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

number	An expression that resolves to a number 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, a value in the range -1 to 1 (inclusive).

To return results in radians, set **$OPTIONS 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 **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

number	An expression that resolves to a number 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.

By default, Caché MVBasic trig functions return results in degrees. To return results in radians, set **$OPTIONS RADIANS**. The result, in radians, is 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 hyperbolic 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

numstr	An expression that resolves to a number or numeric string.

Description

The **SMUL** function multiplies the value of two numeric strings and returns a numeric value. If a *numstr* value is the 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

SORT

Sorts the elements of a dynamic array.

```
SORT(dynarray)
```

Arguments

dynarray	An expression that resolves to a dynamic array.

Description

The **SORT** function takes a dynamic array and returns a dynamic array with its elements sorted in ascending ASCII order. The null string and missing elements are sorted first. Numbers are sorted in ASCII order (for example: 12, 12.3, 123, 13) not numeric order. Numbers are converted to canonical form (removing leading signs and zeros) before sorting; numeric strings are not converted to canonical form.

SORT is provided for compatibility with D3 systems.

Examples

The following example uses the **SORT** function to sort a dynamic array:

```
cities="New York":@VM:"London":@VM:
"Chicago":@VM:"Boston":@VM:"Los Angeles"
PRINT SORT(cities)
```

returns `BostonýChicagoýLondonýLos AngelesýNew York`

See Also

- Dynamic Arrays

SOUNDEX

Returns the Soundex code for an alphabetic string.

```
SOUNDEX(string)
```

Arguments

string	An expression that resolves to an alphabetic string.

Description

The **SOUNDEX** function is used to group and sort near-equivalents of alphabetic strings, such as variant spellings of a name. The Soundex algorithm takes an alphabetic string of any length, such as a name or an English word or phrase, and returns a four-character equivalence code. This code consists of the first recognized letter of the string (which may not be the first character), followed by three integers between 0 and 6 (inclusive) for the remaining 3 code characters. The three numbers assigned by the Soundex algorithm represent up to three distinct consonant sounds (syllables) that follow the initial letter. Repeating letters (such as “mm” or “mn”) have no effect on assigning a Soundex number.

For example, “Fred” is represented as F630, because F is the first character, 6 is assigned to the letter sound “R”, 3 is assigned to the letter sounds “D” or “T”, and 0 indicates that there are no more consonant sounds in the string. Note that vowels and unvoiced letters (A, E, I, O, U, H, W, Y) are not assigned a number. Ann, Anne, Anna, Ana, and Annie are all represented by A500. Anita, Anida, Annette, and Ann T. are all represented by A530. Anton, Anthony, Anoinette are all represented by A535.

Caché MVBasic uses the Soundex algorithm used by the United States Census Bureau; this is not the same algorithm used by other MultiValue implementations. Therefore, all files using Soundex should be regenerated when moving them to Caché MultiValue. The MVBasic Soundex numeric codes for English consonants are as follows: 1=B,F,P,V; 2=C,G,J,K,Q,S,X,Z; 3=D,T; 4=L; 5=M,N, 6=R.

The Soundex algorithm is not case-sensitive; all Soundex codes return the first recognized letter as an uppercase letter, regardless of its case in the input string. All non-alphabetic characters are ignored, including numbers, punctuation characters, and blank spaces. Soundex does not recognize accented letters or non-Latin letters. For example, “Ü-boat” returns B300, exactly the same as “Boat”. If **SOUNDEX** cannot recognize at least one letter in *string*, it returns 0000 (four zeros). If *string* is the null string, **SOUNDEX** returns the null string.

Examples

The following examples use the **SOUNDEX** function to return equivalence codes. Note how the Soundex code is established by the initial letter and the next three significant consonants:

```
PRINT SOUNDEX("M");           ! Returns M000
PRINT SOUNDEX("MMMM");        ! Returns M000
PRINT SOUNDEX("Mc");          ! Returns M200
PRINT SOUNDEX("Mac");         ! Returns M200
PRINT SOUNDEX("McD");         ! Returns M230
PRINT SOUNDEX("McT");         ! Returns M230
PRINT SOUNDEX("McDuff");      ! Returns M231
PRINT SOUNDEX("McDufflebag"); ! Returns M231
```

See Also

- OCONV function
- OCONVS function

SPACE

Returns a string consisting of the specified number of spaces.

```
SPACE(number)
```

Arguments

number	The number of spaces you want in the string. An expression that resolves to an integer.

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 the integer portion.

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

dynarray	An expression that resolves to a dynamic array of positive integers, specifying the number of spaces you want in each corresponding element of the output dynamic 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 missing, or is 0, a negative number, the 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

dynarray	An expression that resolves to a dynamic array.
separator	An expression that resolves to a character or string of characters. **SPLICE** inserts *separator* 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*, with a *separator* between the components of each element.

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.

Caché MVBasic converts numbers to canonical form (resolving signs, removing leading and trailing zeros, removing a leading plus sign, removing a trailing decimal point) before concatenating. Caché MVBasic *does not* convert numeric strings to canonical form before concatenating.

If the two dynamic arrays have different numbers of elements, the returned dynamic array has the number of elements of the longer dynamic array. By default, the shorter dynamic array is padded with null string ("") value elements for the purpose of the concatenation operation. **SPLICE** inserts the *separator* in every element of the returned dynamic array, even when a source element is missing or is the null string. You can use the REUSE function to concatenate a default value (instead of the null string) when the dynamic arrays differ in length.

You can use the REUSE function with **SPLICE** to concatenate the same value to all of the elements of a dynamic array. You can use the **CATS** function to concatenate the elements of two dynamic arrays with no separator between the element components.

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[,ident])
```

Arguments

n	An expression that resolves to an integer code specifying what category of information to return. Available values are 1 through 5, inclusive.
ident	*Optional* — Limits information returned to jobs created by the specified *ident*. An expression that resolves to either a positive integer port number or an account name string. Applicable to *n*=2 and *n*=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 *ident*. 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, 2=form queue type (the SP-CREATE *device-type*), 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=username (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=username (same as 3), 15=Caché username in a security-enabled locked-down system, otherwise "UnknownUser", 17=MV account name, 18=page size (in lines), 19=options (in long format), 20=current position of the despool process (in `lines,copies` 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=username (OS login name), 8=username (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 `^%MV.SPOOL`.

SPOOLER(1) can return information on a form queue, or on a form queue group. If it is a form queue group (element 2=GROUP) then element 3 consists of a subvalue mark delimited list of the form queues in the group.

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:

```
STANDARDý1ýFredý5948ý14213ý54958ýCLOSEDýý1ý1ýýýýFredýFredýýUSERý4ýý
```

which contains the following elements:

```
1 form queue name=STANDARD
2 print job number=1
3 username (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 username (same as 3)=Fred
15 username (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:

```
^%MV.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

number	An expression that resolves to a positive number or 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 an <ILLEGAL VALUE> 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

string	An expression that resolves to a string or a number. *string* may be a dynamic array.

Description

The **SQUOTE** function returns *string* enclosed in single quotation marks. The quotation marks are part of the resulting string. Therefore, using **SQUOTE** increases the length of *string* by 2 characters. If *string* is the null string, **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.

Note: Some MultiValue Basic implementations (D3, for example) use **SQUOTE** and **DQUOTE** to extract quoted substrings from within a string. The Caché MVBasic quote functions do not support this functionality. Use the **FIELD** function or the [ ] operator to extract quoted substrings.

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
```

The following example uses the **SQUOTE** function to enclose a string in single quotation marks:

```
str1 = "Hello"
str2 = 'Hello'
str3 = \Hello\
PRINT str1:str2:str3;  ! Returns HelloHelloHello
PRINT LEN(str1),LEN(str2),LEN(str3);  ! Returns 5   5   5
q1 = SQUOTE(str1)
q2 = SQUOTE(str2)
q3 = SQUOTE(str3)
PRINT q1:q2:q3;       ! Returns 'Hello''Hello''Hello'
PRINT LEN(q1),LEN(q2),LEN(q3);    ! Returns 7    7    7
```

See Also

- QUOTE function
- DQUOTE function
- LEN function
- PRINT statement

SSUB

Subtracts two numeric strings.

```
SSUB(numstr1,numstr2)
```

Arguments

numstr1	The minuend. An expression that resolves to a number or numeric string.
numstr2	The subtrahend. An expression that resolves to a number 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 that indicates the status of the most recently completed MVBasic operation for the current process. Many, but not all, MVBasic operations change the **STATUS** function value. MVBasic sets **STATUS** upon successful completion of an operation (completion without issuing an error).

STATUS returns 0 when an operation performed the intended task, and a non-zero integer (positive or negative) to indicate a situation that prevented the intended completion of the task. For example, a command could have completed without error but performed no operation because a specified operand was of the wrong type, or the specified operand was already in the state that the command was supposed to establish.

MVBasic initializes the **STATUS** function value to 0. The **STATUS** function value persists until changed by the successful completion of an operation. Exiting and reentering the MV Shell resets the **STATUS** function value to 0. You can set the **STATUS** function value to a user-defined positive or negative integer value using the ASSIGN statement.

The **STATUS** function value is set by the following commands. Please see the individual command for the applicable status code values: BSCAN, CLEARFILE, CLOSE, CLOSESEQ, CREATE, DELETE, DELETESEQ, EXECUTE, FILELOCK, FILEUNLOCK, INPUT, OPEN, OPENPATH, OPENSEQ, READ, READBLK, READL, READSEQ, READU, READVL, READVU, SEEK, WEOFSEQ, WRITE, WRITEBLK, WRITESEQ, WRITESEQF, WRITEU, WRITEV, and WRITEVU.

The **STATUS** function value is set by the following functions. Please see the individual function for the applicable status code values: ACCESS(), FILEINFO(), FMT(), ICONV(), ICONVS(), OCONV(), OCONVS(), RECORDLOCKED().

See Also

- ASSIGN statement

STR

Repeats a string value.

```
STR(string,repeats)
```

Arguments

string	An expression that resolves to a string or number.
repeats	An expression that resolves to 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

dynarray	An expression that resolves to a dynamic array.
repeats	An expression that resolves to 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: AAAýBBBýýDDDýEEE

See Also

- LENS function
- STR function
- Dynamic Arrays

SUBR

Returns a value from an external subroutine.

```
SUBR(routine[,arg1[,arg2...]])
```

Arguments

routine	The name of an existing subroutine, specified as a quoted string. If an external (user-defined) subroutine, just specify the subroutine name. If a system-supplied subroutine (a system function), prefix the function name with a minus sign. A list of supported system functions is provided below. This syntax for calling system functions is provided for compatibility with UniData systems.
arg	*Optional* — 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.

Commonly, **SUBR** is used to call user-defined subroutines. You can create a subroutine using the **SUBROUTINE** statement.

If the *routine* name begins with an asterisk (*), **SUBR** first looks it up as a local routine. If not found, **SUBR** looks it up as a global routine. If still not found, **SUBR** generates an error. Note that **routine* processing is different in UniData emulation, as described below.

SUBR can also be used to call certain system-defined functions, using the following syntax:

```
SUBR('-funcname',arg1[,arg2])
```

Note the hyphen appended to *funcname*. The *–funcname* must be quoted. This syntax can be used in I-types; Caché MVBasic converts it to the corresponding standard MVBasic function during I-type compilation. The following MVBasic system functions are supported: ADDS, ANDS, CATS, CHARS, COUNTS, DIVS, EQS, FIELDS, FMTS, GES, GTS, ICONVS, IFS, INDEXS, LENS, LES, LTS, MODS, MULS, NES, NOTS, NUMS, OCONVS, ORS, SEQS, SPACES, SPLICE, STRS, SUBS, and SUBSTRINGS. This syntactical form is compatible with UniData.

For example, the following two calls of the MVBasic LENS function are equivalent:

```
mydyn="Apple":@FM:"Orange":@FM:"Banana"
PRINT LENS(mydyn)
PRINT SUBR('-LENS',mydyn)
```

The following two CMQL queries are equivalent:

```
LIST VOC EVAL "SUBR('-OCONV',12345,'D')"
LIST VOC EVAL "OCONV(12345,'D')"
```

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
```

Emulation

In UniData and UDPICK emulations, a *routine* name with an initial character of * is handled as a global routine name. **SUBR** removes the leading * and then looks up the resulting routine name as a global routine. If the runtime environment is not a UniData emulation, a normal lookup is done on a *routine* name with a leading * character.

The use of `$OPTIONS UNIDATA` in the MVBasic source file *does not* activate this behavior. The handling of names with leading * is determined by the user setting in the command language at runtime. Therefore, to activate this behavior, the CEMU command must set UniData emulation before running a program that calls a *routine* name with a leading *.

See Also

- CALL statement
- GOSUB statement
- SUBROUTINE statement

SUBS

Subtracts the values of corresponding elements in two dynamic arrays.

```
SUBS(dynarray1,dynarray2)
```

Arguments

dynarray1	The minuend. An expression that resolves to a dynamic array of numeric values. If a dynamic array element contains a non-numeric value, **SUBS** treats this value as 0 (zero).
dynarray2	The subtrahend. An expression that resolves to a 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 missing, or is the null string or a non-numeric value, **SUBS** parses its value as 0 (zero).

If the two dynamic arrays have different numbers of elements, the returned dynamic array has the number of elements of the longer dynamic array. By default, the shorter dynamic array is padded with 0 value elements for the purpose of the arithmetic operation. You can also use the REUSE function to define behavior when specifying two dynamic arrays with different numbers of elements.

You can use the ADDS (addition), MULS (multiplication), DIVS or DIVSZ (division), MODS (modulo division), and PWRS (exponentiation) functions to perform other arithmetic operations on the corresponding elements of two dynamic arrays.

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 1ý13ý25ý37
```

The following example subtracts elements of dynamic arrays of differing lengths:

```
a=11:@VM:22:@VM:33:@VM:44
b=2:@VM:2:@VM:2:@VM:2:@VM:2
PRINT SUBS(a,b)
      ! returns 9ý20ý31ý42ý-2ý-2
```

See Also

- ADDS function
- DIVS function
- DIVSZ function
- MODS function
- MULS function
- PWRS function
- Dynamic Arrays

SUBSTRINGS

Returns a substring for each element of a dynamic array.

```
SUBSTRINGS(dynarray,start,length)
```

Arguments

dynarray	An expression that resolves to a dynamic array of elements from which a dynamic array of substrings is to be extracted.
start	An expression that resolves to a positive integer specifying the start position (counting from 1) within each element to begin extracting a substring.
length	An expression that resolves to a positive integer specifying the number of characters to extract from each element, beginning with the *start* 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, or a non-numeric string, **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, or a non-numeric string, the *length* is parsed as 0; 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: "NewýLonýChiýBosýLos"
```

See Also

- Dynamic Arrays
- Strings

SUM

Adds the values of the elements of a dynamic array.

```
SUM(dynarray)
```

Arguments

dynarray	An expression that resolves to a dynamic array of numeric values.

Description

The **SUM** function adds the values the elements in a dynamic array and returns the sum as a dynamic array. If an element is missing or has a null string or a non-numeric value, **SUM** parses its value as 0 (zero).

- If all of the elements in *dynarray* are on the same dynamic array level, **SUM** returns a dynamic array consisting of a single value, the sum of the elements. This is shown in the following example, in which each **SUM** returns 34:

```
a=10:@FM:9:@FM:8:@FM:7
b=10:@VM:9:@VM:8:@VM:7
c=10:@SM:9:@SM:8:@SM:7
PRINT SUM(a);    ! returns 34
PRINT SUM(b);    ! returns 34
PRINT SUM(c);    ! returns 34
```

- If elements in *dynarray* are on different dynamic array levels, **SUM** returns a dynamic array in which elements at the lowest array level are summed. Values at higher levels are returned as array elements. This is shown in the following example:

```
a=9:@VM:10:@VM:11:@FM:8:@FM:7
PRINT SUM(a);    ! returns 30^8^7
b=9:@VM:10:@VM:11:@SM:8:@SM:7
PRINT SUM(b);    ! returns 9^10^26
```

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.

See Also

- ADDS function
- SUMMATION function
- Dynamic Arrays

SUMMATION

Adds the values of the elements of a multi-level dynamic array.

```
SUMMATION(dynarray)
```

Arguments

dynarray	An expression that resolves to a 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 is missing, or has 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

SYSTEM

Returns various system parameter values.

```
SYSTEM(code)
```

Arguments

code	An expression that resolves to an integer code specifying which information to return.

Description

The **SYSTEM** function returns a variety of system parameters. The following table lists the *code* parameters. Those *code* numbers that are not listed are either not implemented or perform no operation and always return 0 or the empty string.

0	Always returns 0 for all emulations except D3, see below.
1	A boolean value that returns 1 if the printer is on. Otherwise, returns 0.
2	Returns the current page width, in characters, as defined by the terminal settings. If output is directed to the printer, it returns the width of the printer (for channel 0).
3	Returns the current page length (depth), in lines, as defined by the terminal settings. If output is directed to the printer, it returns the page length of the printer (for channel 0).
4	Returns the number of lines remaining on the current page, with the last line being line 1. If output is directed to the printer, it returns the printer/spooler values, dependent on the HEADING and FOOTING settings. The PRINTER RESET command resets this value.
5	Returns the current page number, as calculated for the HEADING or FOOTING statement. Note that the SYSTEM(5) value reflects when the function is invoked, not when its return value is printed; the page number value may be the previous page when this return value is printed as the first line of the next page. This parameter may be set using the ASSIGN command from the Caché MultiValue Shell.
6	Returns the current line number.
7	Returns the terminal type code. This parameter may be set using the ASSIGN command from the Caché MultiValue Shell. Refer to the Caché MultiValue CHOOSE.TERM command for further details on terminal types.
9	CPU msec used.
10	A boolean value that indicates whether the data stack is active. Returns 1 if input data is pending from a DATA statement. Otherwise, returns 0.
11	An integer value that indicates the number of items in the default select list. If no select list is active, returns 0. (For UniVerse, PICK, INFORMATION, PIOpen, and IN2 emulation, see below.) See @SELECTED system variable.
12	Returns the current system time in elapsed milliseconds since midnight (local time). This is the same local time returned by the TIME function, which returns the time in elapsed seconds and fractional seconds, rather than in milliseconds. Refer to TIME for details on how local time is determined.
13	Release timeslice.

14	A boolean value that returns 1 when there are unread characters in the input buffer. Returns 0 when there are no characters in the input buffer.
15	Returns the option codes from the end of a command line. For example, if you invoke a cataloged program as `MYPROG (X,P` the value returned by SYSTEM(15) is "XP". If you specify the command `DATE.FORMAT (D)`, SYSTEM(15) returns "D".
16	A boolean value that indicates whether the current process is running from a proc. Returns 1 if running from a proc. Otherwise, returns 0.
17	STOP code for child process.
18	Returns the port number for the terminal. Refer to the LISTME command in *Caché MultiValue Commands Reference*.
19	Returns the current user's login name. (For D3 emulation, see below.)
20	Returns the spooler job number last created by this user. If a job is output to the &HOLD& file (printing mode 3) then the SYSTEM(20) value is not updated. The SYSTEM(20) value is only updated in printing mode 1 (printing to the ^SPOOL global).
22	Returns PERFORM / EXECUTE level. The default is 0. (For D3 emulation, see below.)
23	A boolean value that indicates whether the Break key is enabled. Returns 1 if it is enabled. Otherwise, returns 0. The default is enabled.
24	A boolean value that indicates whether the echoing of input characters is enabled. Returns 1 if it is enabled. Otherwise, returns 0. The default is enabled.
25	A boolean value that indicates whether the current process is running from a phantom process. Returns 1 if running from a phantom process. Otherwise, returns 0.
26	Returns the current user prompt character. The default is “?”.
27	On UNIX® systems, returns the uid (user ID). On other systems, returns 0.
28	On UNIX® systems, returns the effective user ID. On other systems, returns 0. (For D3 emulation, see below.)
29	On UNIX® systems, returns the gid (group ID). On other systems, returns 0. (Only supported in Caché and UniVerse.)
30	On UNIX® systems, returns the effective gid (group ID). On other systems, returns 0. (Only supported in Caché and UniVerse.)
31	Returns the Caché license order number for Caché and for IN2, INFORMATION, Pick, PIOpen, Prime, and UniVerse emulations. (For D3 emulation, see below.)
32	Returns the Caché system manager directory pathname. For example, c:\intersystems\cache\mgr\.
33	Returns the contents of the command stack (Caché, UniVerse, PICK emulations only). See @COMMAND.STACK system variable for further details. (For D3 and UniData emulations, see below.)
34	Returns data pending on the input stack. (Only supported in Caché and UniVerse.)
35	[Only supported in IN2, INFORMATION, PICK, PIOpen, Prime, and UniVerse emulations. See below.]
36	Returns the current licensed user count. Refer to the **%SYSTEM.License.GetUserLimit()** method in the *InterSystems Class Reference*.

40	For an MVBasic program, returns the current program name as `file~program`. For a terminal, returns the current MV shell identifier. For example `MVBASIC420.mvi`, where 420 is the process ID for the current terminal. (Supported for all emulations except D3; in D3 emulation use SYSTEM(157).)
41	Returns the Caché version number. This is the same as the value returned by the ObjectScript $ZVERSION special variable. (Supported for all emulations.)
43	Returns the port number of the process holding a lock requested by the current process. After a failed lock request, the STATUS function returns the process ID (pid) of the holder of the lock.
44	Returns 1 for a Caché MultiValue system, in any emulation except D3.
49	Returns the current call stack as a dynamic array. The top level is the current routine. The array format is: array<n,1> = the stack level, counting from 1; array<n,2> = the program name specified as `file~program`; array<n,3> = the source program line number that called the next level. array<n,3> returns 0 if the line cannot be determined; usually this occurs if the routine has been recompiled.
52	[Only supported in jBASE emulation. See below.]
91	A boolean value that indicates whether the operating system is Windows. Returns 1 if running on an operating system in the Windows NT family. Otherwise, returns 0.
99	Returns the POSIX-compliant current universal system time, specified as an integer number of elapsed seconds since midnight of January 1, 1970, in Greenwich Mean Time (GMT). Refer to the ObjectScript $ZDATETIME and $ZDATETIMEH functions in *Caché ObjectScript Reference*.
100	Returns Caché version information. The format is platform-dependent. *Windows:* system-name;config-file-name:name;release;version;hardware-type;release-date;unused;unused;config-file-name;os-name;hardware/serialnum *UNIX®:* system-name;os-name;config-file-name:name;release;version;hardware;monitor-version;boot-monitor-release-date;config-file-name;os-name;hardware/serialnum For further details, see SYSTEM(100) in *Operational Differences between MultiValue and Caché*.
104	Returns the number of active MultiValue users, as reported by the **LISTU** command.
157	Same as SYSTEM(40). Supported in all emulations.
166	Same as SYSTEM(49). Supported in all emulations.
169	Returns the name of the computer.
255	Returns the operating system type. For example, “Windows NT”.
1001	Returns an emulation table number that specifies the emulation type that the program was compiled with (see below).
1002	Returns current Caché namespace
1005	Returns a boolean value for screen paging control.
1017	Returns the operating system name. For example, “Windows”.
1051	Returns an emulation table number that specifies the emulation type that the program was compiled with (see below).
1052	Returns the current Caché namespace.
1053	Returns the client IP address. For example, “127.0.0.1”

1054	Returns the Caché system manager directory pathname. For example, c:\intersystems\cache\mgr\. See SYSTEM(32).
1055	Returns the current namespace pathname. For example, c:\intersystems\cache\mgr\samples\.
1056	Returns a string of four emulation attributes, as follows: An integer code for the emulation in effect when the program was compiled; the name of the emulation type in effect when the program was compiled; an integer code for the current emulation for the MultiValue account; the name of the current emulation type for the MultiValue account.

You can use the **ASSIGN** statement to modify these system settings.

Emulation

SYSTEM(1001) and SYSTEM(1051) return the current emulation as an integer code, as follows: 0=Caché, 1=jBASE, 2=Reality, 3=UniVerse, 4=UniData, 5=PICK, 6=Ultimate, 7=Prime or INFORMATION, 8=PIOpen, 9=POWER95, 10=MVBase, 11=D3, 12=IN2, 13=R83, 14=UDPICK.

SYSTEM(16) returns the current stack level for all MultiValue emulations *except* Caché, and UniVerse emulation.

- D3 emulation:
  - SYSTEM(0) returns 0 or a process ID indicating lock status. For example, following a failed lock request, it returns the process ID (pid) of the process holding a lock requested by the current process.
  - SYSTEM(11) after a **READNEXT** always returns 0, due to the PICK.SELECT behavior described in READNEXT.
  - SYSTEM(19) returns a unique ID value, composed of the date and time in Caché internal format (DDDDDSSSSS) and (if necessary) an incrementing two-character alpha suffix to ensure uniqueness.
  - SYSTEM(22) returns the port number of the current process.
  - SYSTEM(24) defaults to 0.
  - SYSTEM(30) returns the port number of the process holding a lock requested by the current process.
  - SYSTEM(31) returns the last form queue number (a number from 0 upwards, not the form queue name) that was last assigned with the **SP.ASSIGN** command.
  - SYSTEM(33) returns the call stack (a @VM-delimited stack of routine names) if the current routine is a subroutine; returns nothing if the current routine is a program. D3 also returns the standard SYSTEM(49) call stack.
  - SYSTEM(23), SYSTEM(25), SYSTEM(26), SYSTEM(27), SYSTEM(28), SYSTEM(29), SYSTEM(32), SYSTEM(34), SYSTEM(35), SYSTEM(40), SYSTEM(43), SYSTEM(44), and SYSTEM(52) are not supported in D3 emulation. Use SYSTEM(157) for SYSTEM(40) functionality.
- jBASE emulation:
  - SYSTEM(14) in native jBASE checks for pending input from any source; Caché MultiValue users must change this to check both SYSTEM(14) and SYSTEM(10).
  - SYSTEM(52) returns the computer name.
  - SYSTEM(1001) returns the command line as an attribute-delimited string.
  - SYSTEM(29), SYSTEM(31), SYSTEM(32), SYSTEM(33), SYSTEM(34), and SYSTEM(35) are not supported in jBASE emulation.
- UniData emulation:
  - SYSTEM(11) returns the **SELECT** count (the same as the @SELECTED value) when using Select List 0. For any other select list, only @SELECTED is set. Each invocation of **READNEXT** decrements this SYSTEM(11) count (but not the @SELECTED count).

  - SYSTEM(33) returns the current system platform.
  - SYSTEM(48) when called by a program being run by the PHANTOM command, SYSTEM(48) returns the name of the item in the &PH& file that is receiving program output.
  - SYSTEM(29), SYSTEM(31), SYSTEM(32), SYSTEM(34), SYSTEM(35), and SYSTEM(52) are not supported in UniData emulation.

- UniVerse emulation. Unless otherwise specified, references to UniVerse emulation also apply to PICK, Prime, INFORMATION, PIOpen, and IN2:
  - SYSTEM(11) returns a boolean value that indicates whether the default select list is active.
  - SYSTEM(29) is supported in UniVerse emulation, but is not supported in other emulation modes.
  - SYSTEM(34) is supported in UniVerse emulation, but is not supported in other emulation modes.
  - SYSTEM(35) returns the number of active MultiValue users, as reported by the **LISTU** command line command.
  - SYSTEM(40) returns the current namespace pathname.
  - SYSTEM(52) is not supported in UniVerse emulation.

See Also

- ASSIGN statement

TAN

Returns the tangent of an angle.

```
TAN(number)
```

Arguments

number	An expression that resolves to a number that specifies 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.

By default, Caché MVBasic trig functions return results in degrees. To return results in radians, set **$OPTIONS 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

number	An expression that resolves to a number that specifies 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.

By default, Caché MVBasic trig functions return results in degrees. To return results in radians, set **$OPTIONS 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 local time in a format such as the following:

```
29848.349
```

This represents the elapsed number of seconds since midnight, with fractional seconds. This is the same local time returned by SYSTEM(12). The only difference is that **TIME** displays the count in elapsed seconds and fractional milliseconds; **SYSTEM(12)** displays the count in elapsed milliseconds.

The ICONV function can convert a time value with fractional seconds from display format to an internal count of elapsed seconds since midnight with fractional seconds. All other time and date functions use whole seconds as the smallest unit of time.

TIME, **TIMEDATE**, **SYSTEM(12)**, and **@TIME** all return a local time value. **SYSTEM(99)** returns a Coordinated Universal Time (UTC) time value.

Caché MultiValue determines local time (and date) as follows:

- It determines the current Coordinated Universal Time (UTC) from the system clock.
- It adjusts UTC to the local time zone by using the value of the Caché special variable $ZTIMEZONE.
- It applies local time variant settings (such as Daylight Saving Time) for that time zone from the host operating system.

Note: The local time returned by the **TIME** function is *not* the same as the local time returned by the @TIME system variable. Both return time in elapsed seconds since midnight. However, **TIME** returns the current time. **@TIME** returns the time of invocation of the current routine; the **@TIME** value does not change during the execution of the current routine. When issued from the MultiValue Shell, **@TIME** contains the time that the last (prior) command line was invoked. For further details, see the System 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 **OCONV** function to convert time from internal format to display format. Note that **OCONV** conversion truncates fractional seconds.

```
now=TIME()
PRINT now
PRINT OCONV(now,"MTS")
```

The following example shows the difference between the **TIME** function and the **@TIME** system variable:

```
SLEEP 2
PRINT @TIME,TIME()
```

The **TIME** function returns the current time; the **@TIME** variable returns the time that the **SLEEP** command was invoked.

See Also

- TIMEDATE function
- OCONV function
- 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.

Note: You can specify the default date format using Caché NLS. Because of operational differences between MV and Caché NLS in the handling of month names, your NLS default date format must represent months as integers.

TIMEDATE does not return fractional seconds. It truncates fractional seconds. To return the local time with fractional seconds, use the TIME function or the SYSTEM(12) function.

Caché MultiValue determines local time (and date) as follows:

- It determines the current Coordinated Universal Time (UTC) from the system clock.
- It adjusts UTC to the local time zone by using the value of the Caché special variable $ZTIMEZONE.
- It applies local time variant settings (such as Daylight Saving Time) for that time zone from the host operating system.

See Also

- TIME function
- OCONV function
- ObjectScript: $HOROLOG special variable
- SQL: NOW function

TRANS

Reads field data from a MultiValue file.

```
TRANS(mvfile,recID,fieldno,code)
```

Arguments

mvfile	The name of a MultiValue file defined in the VOC. An expression that resolves to a string. If there are multiple defined data sections (data files), you can specify *mvfile* as either `"filename,datasection"` or simply `"datasection"`.
recID	The record ID of the desired record. An expression that resolves to a string (commonly a positive integer). This argument can be multivalued in which case **TRANS** returns multivalues.
fieldno	An expression that resolves to either an integer or a non-numeric string. If an integer, *fieldno* specifies the field number of the field to be read, or, if -1, returns the contents of the entire record. This usage is compatible with most MultiValue systems. If a non-numeric string, *fieldno* specifies an attribute to look up in the file's dictionary. If that attribute is a “D” data defining entry, **TRANS** looks up the data using the corresponding field number recorded in the dictionary. This usage is compatible with UniData systems.
code	A letter code that specifies what to do if the requested record does not exist. An expression that resolves to a quoted string.

Description

The **TRANS** function returns a field value from a MultiValue file. Unlike a **READ** statement, you do not have to use the **OPEN** statement to open the MultiValue file before issuing a **TRANS**.

The *code* argument determines how **TRANS** should respond when you request an invalid field. The following are valid letter codes:

X	Return an empty string if the specified record or field cannot be located.
V	Return an empty string and generate an error message if the specified record or field cannot be located.
C	Return the *recID* value if the specified record or field cannot be located.
N	Return the *recID* value if the specified field value is null.

The **TRANS** and **XLATE** functions are functionally identical.

Examples

The following example illustrates the use of the **TRANS** function:

```
mydyn = TRANS("TEST.FILE",1,1,"X")
PRINT "the field value:",mydyn
```

See Also

- XLATE function
- READ statement
- STATUS statement
- Dynamic Arrays

TRIM

Removes leading and trailing characters from a string.

```
TRIM(string[,char[,code]])
```

Arguments

string	The string to trim. An expression that resolves to a string.
char	*Optional* — The character to remove from *string*. An expression that resolves to a single-character string. The default is to trim blank spaces and tabs.
code	*Optional* — The type of trimming to perform, An expression that resolves to a single-character code 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 tabs, and replaces multiple (redundant) spaces (including tabs) 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. If *char* is set to a multi-character string, only the first character is used. If *char* is omitted or set to a single-character string containing a blank space (" "), blank spaces are trimmed. If *char* is set to the string empty string (""), *string* is returned unchanged.

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). Use **TRIMS** to remove leading, trailing, and multiple embedded blank spaces from all of the elements of a dynamic array.

Trim Codes

By default, **TRIM** removes leading, trailing, and redundant space characters, and tabs. **TRIM** also removes leading, trailing and redundant space characters if you specify a *code* of "R" or the empty string ("").

You can perform other types of trim operation by specifying a single-character *code* string. The following are the available *code* characters:

A	All occurrences of *char* removed from *string*.
B	Both leading and trailing occurrences of *char* removed.
D	Duplicate blank spaces and leading and trailing blank spaces removed. *char* must be specified, but its value is ignored.
E	Remove trailing spaces. *char* must be specified, but its value is ignored.
F	Remove leading spaces. *char* must be specified, but its value is ignored.
L	Leading occurrences of *char* removed.
R	Redundant, leading, and trailing occurrences of *char* removed. This is the default.
T	Trailing occurrences of *char* 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
- TRIMS function
- Strings

TRIMB

Removes trailing blanks from a string.

```
TRIMB(string)
```

Arguments

string	An expression that resolves to a 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

dynarray	An expression that resolves to a 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 **TRIMS** to remove leading, trailing, and multiple embedded blank spaces from all of the elements 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
- TRIMS function
- TRIM function
- TRIMB function
- TRIMF function
- Dynamic Arrays

TRIMF

Removes leading blanks from a string.

```
TRIMF(string)
```

Arguments

string	An expression that resolves to a 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

dynarray	An expression that resolves to a 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 **TRIMS** to remove leading, trailing, and multiple embedded blank spaces from all of the elements 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
- TRIMS function
- TRIM function
- TRIMB function
- TRIMF function
- Dynamic Arrays

TRIMS

Removes leading and trailing spaces from each element of a dynamic array.

```
TRIMS(dynarray)
```

Arguments

dynarray	An expression that resolves to a dynamic array.

Description

The **TRIMS** function trims leading and trailing blank spaces from each element of a dynamic array. It also replaces multiple (redundant) spaces within an element with a single space. It returns the resulting trimmed dynamic array.

To trim only trailing blank spaces from each element, use **TRIMBS**. To trim only leading blank spaces from each element, use **TRIMFS**.

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 **TRIMS** function to trim leading and trailing spaces:

```
RawDyn="North   ":@VM:"South   ":@VM:"  East":@VM:"West..."
PRINT LENS(RawDyn)
TrimDyn = TRIMS(RawDyn)
PRINT LENS(TrimDyn)
```

The following example uses the **TRIMS** function to trim redundant spaces within elements:

```
RawDyn="North    America":@VM:"Central     America":@VM:"South America"
PRINT LENS(RawDyn)
TrimDyn = TRIMS(RawDyn)
PRINT LENS(TrimDyn)
```

See Also

- LENS function
- TRIMBS function
- TRIMFS function
- TRIM function
- TRIMB function
- TRIMF function
- Dynamic Arrays

UNASSIGNED

Determines if a variable is unassigned.

```
UNASSIGNED(var)
```

Arguments

var	A user variable. If *var* 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, a dynamic array value, or the null string.

The input *var* can be a local variable, a global variable, or a process-private global variable. It can be with or without subscripts.

Note: **UNASSIGNED** should not be used on system variables (@ variables). It always returns 1 for all @ variables, whether or not the @ variable currently has a value.

The **COMMON** statement initializes variables as unassigned in Caché MVBasic. Array variable initialization varies with different MultiValue emulations. You can use the **$KILL** statement to unassign user variables.

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 all of these variables:

```
a=123
b="fred"
c=1:@VM:2:@VM:3
d=""
^a="fruit"
^a(3)="banana"
^||a="ppv"
PRINT UNASSIGNED(a)
PRINT UNASSIGNED(b)
PRINT UNASSIGNED(c)
PRINT UNASSIGNED(d)
PRINT UNASSIGNED(^a)
PRINT UNASSIGNED(^a(3))
PRINT UNASSIGNED(^||a)
```

See Also

- COMMON statement
- ASSIGNED function

UNICHAR

Returns the character corresponding to the specified character code.

```
UNICHAR(charcode)
```

Arguments

charcode	An expression that resolves to a base-10 integer that identifies a character. For 8-bit characters, *charcode* must be a positive integer in the range 0 through 255. For 16-bit characters, *charcode* must be a positive integer 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**, **CHAR**, and **BYTE** 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 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

- BYTE function
- 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

dynarray	An expression that resolves to a dynamic array of base-10 integers that identify characters. For 8-bit characters, each element value must be a positive integer in the range 0 through 255. For 16-bit characters, each element value must be a positive integer in the range 256 through 65534.

Description

The **UNICHARS** function takes a dynamic array of character codes and returns a dynamic array containing the corresponding character for each element.

This is the inverse of the **UNISEQS** function, which 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, **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 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 AýBýCýD
```

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

char	An expression that resolves to a single character. If *char* 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 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

dynarray	An expression that resolves to a dynamic array.

Description

The **UNISEQS** function takes a dynamic array of characters and returns a dynamic array containing the corresponding numeric code for the first character in each element. 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 is missing or contains the 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 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 65ý66ý67ý68
```

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

string	An expression that resolves to a string.

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 **OCONV** 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 "CACHÉ FROM INTERSYSTEMS"
```

See Also

- DOWNCASE function
- OCONV function

XLATE

Reads field data from a MultiValue file.

```
XLATE(mvfile,recID,fieldno,code)
```

Arguments

mvfile	The name of a MultiValue file defined in the VOC. An expression that resolves to a string. If there are multiple data files, you can specify *mvfile* as either `"dirname,datafile"` or simply `"datafile"`.
recID	The record ID of the desired record. An expression that resolves to a string (commonly a positive integer). This argument can be multivalued in which case **XLATE** returns multivalues.
fieldno	An expression that resolves to either an integer or a non-numeric string. If an integer, *fieldno* specifies the field number of the field to be read, or, if -1, returns the contents of the entire record. This usage is compatible with most MultiValue systems. If a non-numeric string, *fieldno* specifies an attribute to look up in the file's dictionary. If that attribute is a “D” data defining entry, **XLATE** looks up the data using the corresponding field number recorded in the dictionary. This usage is compatible with UniData systems.
code	A letter code that specifies what to do if the requested record does not exist. An expression that resolves to a code string.

Description

The **XLATE** function returns a field value from a MultiValue file. Unlike a **READ** statement, you do not have to use the **OPEN** statement to open the MultiValue file before issuing an **XLATE**.

The *code* argument determines how **XLATE** should respond when you request an invalid field. The following are valid letter codes:

X	Return an empty string if the specified record or field cannot be located.
V	Return an empty string and generate an error message if the specified record or field cannot be located.
C	Return the *recID* value if the specified record or field cannot be located or has an empty string value ("").
N	Return an empty string if the specified record or field cannot be located.

The **XLATE** and **TRANS** functions are functionally identical.

Examples

The following example illustrates the use of the **XLATE** function:

```
mydyn = XLATE("TEST.FILE",1,1,X)
PRINT "the field value:",mydyn
```

See Also

- TRANS function
- READ statement
- STATUS statement
- Dynamic Arrays

XTD

Converts a number from hexadecimal to decimal.

```
XTD(hexnum)
```

Arguments

hexnum	An expression that resolves to a positive hexadecimal integer. If *hexnum* contains only the characters 0 – 9, it may be specified as a number; if it includes the hexadecimal characters A – F, it must be specified as a string. The hexadecimal characters A – F are not case-sensitive.

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 integer value.

If *hexnum* is zero, a negative number, or a non-numeric string, **XTD** returns 0. If *hexnum* is a fractional number, it is truncated to its integer portion. If *hexnum* is a mixed numeric string, the hexadecimal part is parsed until a non-hexadecimal character is encountered. Thus “7Dwarves” is parsed as 7D. If *hexnum* is the null string, a <FUNCTION> 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 28
PRINT XTD("-1C");     ! Returns 0
PRINT XTD("red");     ! Returns 0
```

See Also

- DTX function

Caché MultiValue Basic General Concepts

Whitespace and Comments

Whitespace and comment indicators.

Whitespace

A whitespace character can be a blank space or a tab. Between command arguments, or between an operator and its operands you can specify no whitespace, one whitespace character, or multiple whitespace characters. For example, the following pairs of statements are functionally identical:

```
PRINT"fred";!a comment
PRINT   "fred"   ;  ! a comment
PRINT"butter":"fly"
PRINT   "butter"  :   "fly"
PRINT(4=3)+2
PRINT  ( 4 = 3)  +  2
```

However, if the first command argument is a number, it must either be quoted, enclosed in parentheses, or separated from the command name by one or more whitespace characters. If the first command argument is a variable, it must either be enclosed in parentheses, or separated from the command name by one or more whitespace characters.

An MVBasic statement can begin in column 1, or can be indented with any number of whitespace characters.

Vertical spacing (line breaks) within a command are only permitted following a comma or following a line continuation character. Refer to the Line Continuation page of this manual for further details.

Comments

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. A comment can appear after a comma in a command containing a line break. There are three ways to indicate a comment on the same line as executable code: the **REM** statement, a semicolon asterisk (;*), or a semicolon exclamation mark (;!). When indicating a comment on a separate program line, you can use a dollar sign asterisk ($*), an asterisk (*), or an exclamation point (!), in addition to the **REM**, ;* and ;! forms. Whitespace is permitted (but optional) before a comment indicator, or between the semicolon and the asterisk or exclamation mark.

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 "Hello",   ;* comment text
;* further comments
      "World!"
PRINT "Hello",  REM comment text
      " World!"
REM  this is a comment
$* comment text
```

See Also

- REM statement
- Line Continuation

Compiler Directives

Preprocessor statements issued to the MVBasic compiler.

Caché MVBasic supports most, but not all of the compiler directives provided by other flavors of MultiValue. The *MultiValue Basic Quick Reference* contains a Compiler Directives table showing which of the UniVerse compiler directives are supported by Caché MVBasic. The following are the Caché MVBasic compiler directives:

#INCLUDE

```
#INCLUDE program
#INCLUDE filename program
#INCLUDE filename,section program
#INCLUDE account,filename, program
#INCLUDE account,filename,section program
```

Inserts MVBasic source code from a specified shared library routine into the program being compiled. The inserted code is compiled along with the program in which it is included. The *program* shared library routine has a `.h` suffix. If *filename* is omitted, defaults to the filename of the current program. The *account* name (namespace name) and the *section* name are optional.

#PRAGMA

```
#PRAGMA ROUTINENAME=rrr
```

When you compile an MVBasic routine, the system generates a default routine name, with the format `MVB.xxx`. This name applies to the intermediate MVI source code (xxx.mvi) and to the object code (xxx.obj).

Use **#PRAGMA** to override this default and specify this routine name. You do this by adding the following statement to the source code file: `#PRAGMA ROUTINENAME=rrr`, where "rrr" is the routine name. This routine name must satisfy the naming convention syntax for Cache routines: the first character must be % or a letter, and subsequent characters must be letter, a number, or a period.

CAUTION: Use care in selecting a routine name, or when copying program code that contains `#PRAGMA ROUTINENAME=rrr`. **#PRAGMA** creates a routine with the specified name even if that name was already assigned to an existing routine. It overwrites the object code for an existing routine with object code for the current routine.

Once a routine name has been associated with a source file, that name will continue to be used, even if the #PRAGMA statement is removed. To stop using this associated routine name and have the system generate a new default routine name, specify an empty routine name:`#PRAGMA ROUTINENAME=""`.

This empty routine name should be specified only for the first compile. This removes the associated routine name. The #PRAGMA statement should then be removed before subsequent compiles of the source code file. Otherwise the system will continue to generate a new name each time the routine is compiled, instead of each compile replacing the prior existing routine.

$COPYRIGHT

```
$COPYRIGHT text
```

The **$COPYRIGHT** statement inserts the specified copyright text into the generated object code. This text is a non-executable comment. *text* is a string enclosed in double or single quotes. The *text* string can include Unicode characters.

If there is more than one **$COPYRIGHT** statement in a routine, only the text from the final **$COPYRIGHT** is inserted into the generated object code.

$DEFINE

```
$DEFINE symbol value
```

Assigns a value to a symbol used at compile time.

Do not specify a comment on the same line as a **$DEFINE** statement. **$DEFINE** uses the entire line as is, including any leading or trailing blank spaces and any comments.

The preprocessor symbol INTERSYSTEMS is automatically defined for all Caché MVBasic compilations.

$IFDEF / $IFNDEF

Performs a logical branch based on whether a symbol is defined (**$IFDEF**) or not defined (**$IFNDEF**). Specify a block of dependent code after this statement. You can use or omit a **$ELSE** clause. Terminate the block of dependent code with a **$ENDIF** keyword.

The preprocessor symbol INTERSYSTEMS is automatically defined for all Caché MVBasic compilations.

$INCLUDE

A synonym for **#INCLUDE**.

$INSERT

A synonym for **#INCLUDE**.

$UNDEFINE

Removes an assigned meaning from a symbol used at compile time. **$UNDEFINE** reverses the action of **$DEFINE**.

MV Data Types

Data types supported for MultiValue compatibility.

%MV.Date

Data type for the MultiValue internal representation of dates. It represents the elapsed number of days since December 31, 1967. This data type extends Caché %Date, which represents the elapsed number of days since December 31, 1840. For further details, refer to the MVBasic DATE function and %MV.Date class in the *InterSystems Class Reference*.

%MV.Numeric

Data type for the MultiValue internal representation of numeric values. This data type corresponds to %Numeric, but has an additional DESCALE parameter.

DESCALE only exists as a parameter for %MV.Numeric, and controls descaling — moving the decimal point for numbers stored with implicit decimals. If you expect numbers to be entered accurately with decimals, then you need to descale on input to move the decimal to the internal format (in most cases, remove the decimal).

SCALE is inherited from %Numeric, and controls the number of decimal places that the number is rounded to, usually reducing precision. There's no sense in scaling on input because you can't add precision that isn't there. DESCALE is reversable, because it just moves the decimal. SCALE is not reversible, because it reduces the precision.

A null value is stored in %MV.Numeric as NULL. Descaling is ignored for null values.

For further details, refer to the %MV.Numeric class in the *InterSystems Class Reference*.

See Also

- Caché SQL Data Types

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.

The level of a dynamic array element can be specified either by delimiter character variables (for example, @FM), or by numeric operators (for example, <1,2>). These two ways of specifying the level of dynamic array elements are described below.

You can use the RAISE and LOWER functions to change the level of dynamic array elements.

Dynamic Array Level Delimiter Characters

Dynamic arrays can be 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 (see Subvalue Considerations, below)	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.

The Text Mark (@TM) character is inserted in a string by the FMT and FMTS functions.

The character values specified in the above table are the English locale default values. The dynamic array level code variables can be mapped to other character values, as specified in your NLS locale definition.

Dynamic Array Level Numeric Operators

You use the <> operators to assigned element values to a dynamic array, extract an element value from a dynamic array, or to replace an element value in a dynamic array. The <> operators use the following formats:

```
<f>
<f,v>
<f,v,s>
```

Where f (field), v (value), and s (subvalue) are comma-separated integers representing the position of the desired element (counting from 1). For example, <2,3> indicates the 3rd Value Mark value within the 2nd Field Mark field.

The first dynamic array element is specified by <1>. Specifying <0> means to insert the specified value to the beginning of the dynamic array; this value then becomes the first dynamic array element. Specifying <-1> (or any negative integer) means to insert the specified value at the end of the dynamic array. The use of these integers is shown in the following example:

```
alphabet<0>="B"   ; "B"
alphabet<0>="Z"   ; "Z^B"
alphabet<1>="A"   ; "A^B"
alphabet<3>="C"   , "A^B^C"
alphabet<-1>="D"  , "A^B^C^D"
```

The following example assigns the ^fruit global a dynamic array of field elements:

```
^fruit<1>="Apple"
^fruit<2>="Orange"
^fruit<3>="Banana"
```

This is exactly equivalent to:

```
^fruit="Apple":@FM:"Orange":@FM:"Banana"
```

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.

The subscript `dynarray<0,v>` is parsed as `dynarray<1,v>`. The subscript `dynarray<0,0,s>` is parsed as `dynarray<1,1,s>`. This is true for Caché and all emulations except UniData.

Subvalue Considerations

Caché MultiValue and all emulations allow a simple selection by subvalues on multivalued fields. UniData emulation also can select by subvalues in single-valued fields. Caché MultiValue supports selection on subvalued fields if the field is defined as a property in an associated class with the MVSVASSOCIATION parameter set.

When exploding, UniVerse and UniData explode subvalues, but D3, Reality, and jBASE only explode values. Reality provides a BY-EXP-SUB keyword to explode by subvalues which gives the same result as a simple BY-EXP on UniVerse. For further details, refer to the BY clause in *Caché MultiValue Query Language (CMQL) Reference*.

When using print limiting, D3 and Reality limit to values where the desired subvalue is the first or only subvalue. UniVerse limits to only the subvalues that have the limiting 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 a label must be a letter or a number. If the first character of the label is a letter, the label can contain the underscore (_) character. The underscore character cannot be the last character in a label.
- 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 31 characters are significant. The label must be unique within the first 31 characters.

Labels are used by the GOTO statement. This statement may be abbreviated as the **GO** statement.

A label cannot be used on a program line containing a **SUBROUTINE** or **FUNCTION** command.

Line Continuation

Allows a program statement to occupy multiple lines.

Description

By default, a MVBasic program statement can only continue to a second line if it contains a comma. You can specify a line break following a comma. For example:

```
CRT "Hello",
    "World!"
```

However, by specifying a line continuation character at the end of a line, you can place a line break almost anywhere in a program statement and continue the program statement on the next line.

You must enable a line continuation option in order to use line continuation characters. You can enable either (or both) of the two supported line continuation characters: backslash (\) or vertical bar (|). The following $OPTIONS statements enable these line continuation characters:

```
$OPTIONS LINE.CONT.BSLASH

$OPTIONS LINE.CONT.VBAR
```

You can use either or both of these **$OPTIONS** statements. Like most option statements, you can disable an enabled option by prefacing it with a minus sign. For example `$OPTIONS -LINE.CONT.BSLASH`.

Once enabled, the specified line continuation character may be used in that MVBasic program. The line continuation character must be the last character in a line. A space or comment is not permitted following a line continuation character. It is recommended that you separate the line continuation character from the preceding characters by a space, though this is not required. You can continue a program statement over any number of lines by specifying a line continuation character at the end of each line.

You can use the backslash character (\) as both a line continuation character and a string quotation character in the same program. For example, the following is valid MVBasic code:

```
$OPTIONS LINE.CONT.BSLASH
CRT \Hello World! \: \
  \It's an "awesome" day\
```

For clarity of code, it would be preferable to use the vertical bar (|) line continuation character when the backslash is being used as a quotation character. Regardless of which string quotation characters are used, you cannot use a line continuation character within a string.

Emulation

In Caché and most emulations line continuation is disabled by default.

In jBASE emulation, the default is LINE.CONT.BSLASH enabled.

In UniData emulation, the default is LINE.CONT.VBAR enabled.

See Also

- Whitespace and Comments

MATCH Pattern Matching

A pattern match operator.

```
string MATCH code
string MATCHES code

string MATCH string
string MATCHES string
```

Arguments

string	Any valid string expression.
code	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 OCONV and OCONVS functions provide similar pattern matching support.

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 (""). (This code is not supported for **OCONV** / **OCONVS** pattern matching.)
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 *n* 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 *n* number of alphabetic characters.
0N	Matches any number of number characters, defined as the numbers 0 through 9. Number characters *do not* include the plus sign, minus sign, or decimal separator. Also matches the null string ("").
nN	Matches exactly *n* 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.

The same *code* values can be used in an <<...>> inline prompt, to pattern match test an interactive input value. Inline prompts can be used in MVBasic statements or MultiValue command line commands. They are described in the *Caché MultiValue Commands Reference*.

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.

Multiple Patterns

Caché MVBasic does not support the use of the "}" (right curly brace) character to delimit multiple match patterns. Code from other MultiValue implementations (such as Pick and UniVerse) that uses this syntax must be changed to replace the right curly brace with the @VM (value mark) character.

The following example shows a match to multiple patterns. If a match occurs with either pattern, the THEN clause is taken:

```
IF F1 MATCHES "'K'...":@VM:"'V'..." THEN GOSUB 100
```

See Also

- Strings
- Dynamic Arrays
- The "Pattern Match Operators" section of the WITH clause in the *Caché MultiValue Query Language (CMQL) Reference*

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.

Caché Objects

Accessing class methods from MVBasic.

$SYSTEM

You can access methods belonging to $SYSTEM classes from Caché MVBasic by using the –> syntax. The arrow (–>) indicates that what precedes it is the name of a class, and what follows it is something belonging to the class, such as a method.

A simple example of this usage in shown in the following:

```
CRT $SYSTEM.OBJ->Version()
```

This returns the current Caché Objects version number. This is similar to the information returned by the MVBasic **SYSTEM(41)** function.

In ObjectScript, the same class method would be invoked using dot syntax, as follows:

```
  WRITE $SYSTEM.OBJ.Version()
```

For further details on $SYSTEM classes, refer to the *InterSystems Class Reference*.

See Also

- SYSTEM function

Operators

Arithmetic, logical, and string operators.

Overview

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.

The following types of operators are supported:

- Arithmetic Operators
- Logical Operators
- String Operators

Arithmetic Operators

The following are the arithmetic operators supported by Caché MVBasic:

=	Numeric equality (assignment) operator. `a=5+3`
+	Addition operator. `a=5+3`
++	Increment operator. `a++`
+=	Increment (addition assignment) operator. `a+=3`
–	Subtraction operator. `a=5-3`
—	Decrement operator. `a--`
–=	Decrement (subtraction assignment) operator. `a-=3`
*	Multiplication operator. `a=5*3`
*=	Multiplication assignment operator. `a*=3`
/	Division operator. `a=5/3`
/=	Division assignment operator. `a/=3`
**	Exponentiation operator. `a=5**3`
^	Exponentiation operator. `a=5^3`
( )	Grouping (nesting) operator. `a=((5+3)*2)+1`

The division operator (/) performs exact division, returning a fractional quotient. To perform integer division, truncating the fractional portion of the quotient, use the DIV function. To return the modulo of integer division, use the MOD function. You cannot divide a number by 0. Attempting to do so results in a <DIVIDE> error.

The exponentiation operators (** or ^) can perform exponentiation by raising any base number (*num*) to any power (*exponent*), subject to the following: If *num* is non-zero and *exponent* is 0, exponentiation returns 1. If *num* and *exponent* are both 0, exponentiation returns 0. If *num* is 0 and *exponent* is a non-zero negative number, exponentiation generates an <ILLEGAL VALUE> error. If *num* is a non-zero negative number and *exponent* is a fractional number, exponentiation generates an <ILLEGAL VALUE> error. Very large positive *exponent* values (such as 9**153) or very small *num* values with a negative *exponent* (such as .00005**-30) may result in an overflow, generating a <MAXNUMBER> error. Very

large negative *exponent* values (such as 9**-135) or very small *num* values with a positive *exponent* (such as .00005**30) may result in an underflow, returning 0. You can also perform exponentiation using the PWR function.

By default, Caché MVBasic arithmetic operators do not perform vector arithmetic on the elements of dynamic arrays. To perform vector arithmetic, use the ADDS, SUBS, MULS, DIVS, MODS, and PWRS functions. In some MultiValue emulations the arithmetic operators (+, -, *, /, **) do perform vector arithmetic on dynamic arrays, as described below.

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 ObjectScript uses a different order of operations: it uses strict left-to-right evaluation of arithmetic operators.

Emulation

INFORMATION, jBASE, PIOpen, Prime, and UniData set **$OPTIONS VEC.MATH**. This causes the five basic arithmetic operators to perform vector arithmetic on dynamic arrays. Thus the + operator is equivalent to the ADDS function. The – operator is equivalent to the SUBS function. The * operator is equivalent to the MULS function. The / operator is equivalent to the DIVS function. The ** operator is equivalent to the PWRS function. These operators perform vector arithmetic when supplied dynamic array arguments, and perform simple arithmetic operations when supplied numeric arguments.

Logical Operators

Logical operators result in a boolean result, either 1 (True) or 0 (False). Caché MVBasic supports comparison logical operators (equal to, greater than) and logic operators that associate multiple comparison logical operators (AND, OR).

Comparison Logical Operators

Logical operators can compare numbers, strings, etc. String comparisons are case-sensitive. Strings are compared character-by-character. A string is logically “greater than” when a character is higher in collation sequence than its corresponding character. For example, "fred">"Fred"=1 (True), because “f” is higher in the ASCII sequence than “F”. "fred">"fre"=1 (True), because “d” is higher in the ASCII sequence than null.

= 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.

The following example demonstrates the 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)
```

AND / OR Logical Operators

The following logical operators are used to specify multiple equality operations:

& AND	Logical AND operator.
! OR	Logical OR operator.

Left and Right Side Evaluation

MultiValue emulations differ in how to handle evaluation of AND and OR logical operations. Some MultiValue systems support full logical evaluation: both sides of the logical operator are evaluated, regardless of the logical value of the left side operation. Other MultiValue systems support partial logical evaluation (short circuit evaluation):

- AND operation: evaluate the left side operation, and if it evaluates to False, do not evaluate the right side operation.
- OR operation: evaluate the left side operation, and if it evaluates to True, do not evaluate the right side operation.

Caché MultiValue uses partial logical evaluation by default. The following emulations also default to partial logical evaluation: IN2, INFORMATION, jBASE, PICK, PIOpen, Prime, UDPICK, Ultimate, UniData, and UniVerse. The following emulations default to full logical evaluation: D3, MVBase, R83, Power95, and Reality.

You can use `$OPTIONS FULL.LOGICAL.EVALUATION` to enable full logical evaluation. You can use `$OPTIONS -FULL.LOGICAL.EVALUATION` to enable partial logical evaluation.

Order of Evaluation

Caché MultiValue gives equal precedence to the AND and the OR logical operators. This means that multiple AND and OR logical operations are evaluated in strict left-to-right sequence, unless parentheses are provided to specify evaluation sequence. Caché uses the same strict left-to-right order of evaluation.

String Operators

The following are the string operators supported by Caché MVBasic:

: 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.
:=	Concatenation assignment operator. Placed between a variable name and a value to be concatenated to the variable's value. For example, if `a="abcd"` using `a:="efg"` results in `a="abcdefg"`.
[]	Substring extract operator. Placed after a string, the brackets enclose positive integers: `string[start,length]` specifies the start position and length of the substring to be extracted from the start of the string. `string[length]` specifies the length of the substring to be extracted from the end of the string. Emulation: IN2: the default setting IN2.SUBSTR makes `string[n]` equivalent to `string[n,1]`, specifying the start position (*n*) from the beginning of the string and a length of 1. Reality: the default setting REAL.SUBSTR causes `string[start,-end]` to count *start* from the beginning of the string and to count *–end* as the ending point, counting from the end of the string; `string[-start,-n]` counts both the starting and ending points from the end of the string.
<>	Dynamic array element extract or replacement operator. For further details see the Dynamic Arrays page of this manual.

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

By default, Caché MVBasic converts numeric strings to numeric and boolean values using the ObjectScript conventions. To perform these conversions using PICK mode conventions (used by all MultiValue system emulation modes) specify **$OPTIONS PICK.CONVERT**.

An *empty string* in Caché is either converted to a numeric/boolean value of 0, or treated a string of zero length, depending on the function. In **PICK.CONVERT** mode an empty string is always converted to 0.

A *numeric string* is a string that contains only numeric characters (the number 0 through 9, a single leading plus or minus sign, the decimal point delimiter, and the letter E used for scientific notation). It may include leading or trailing zeros. A numeric string is always accepted as a numeric. Thus "123.4" is identical to 123.4. In this conversion, a leading plus sign and leading and trailing zeros are removed, and the decimal point is removed if not followed by a fractional value. In Caché mode, multiple leading plus and minus signs are permitted and evaluated. In PICK mode, only a single leading plus or minus sign is evaluated; multiple signs are treated as a mixed numeric string.

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 very different in Caché mode and PICK mode. In Caché mode, the numeric is parsed until the first non-numeric character is encountered. Thus "12.5 kilometers" is parsed as 12.5. The boolean value of a mixed numeric string simply depends on whether the resulting number is zero or non-zero. In PICK mode, a mixed numeric string is never parsed for its numeric value. When a number is expected, a mixed numeric string is always evaluated as 0, and frequently generates an error. When converted to a boolean, a mixed numeric string always returns True, because any string is a non-zero value.

The following examples demonstrate the numeric conversion differences between Caché mode and PICK mode:

```
;! Cache Mode
PRINT "" + 3           ;! returns 3
PRINT "+007" + 3       ;! returns 10
PRINT "--7" + 3        ;! returns 10
PRINT "7dwarves" + 3   ;! returns 10
```

```
;! PICK Mode
$OPTIONS PICK.CONVERT
PRINT "" + 3           ;! returns 3
PRINT "+007" + 3       ;! returns 10
PRINT "--7" + 3        ;! returns 3
PRINT "7dwarves" + 3   ;! returns 3
```

System Variables

System-defined @ variables used for storing data values.

System-Defined 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. Unless otherwise indicated, they cannot be set by user programs. If no value is set, these variables contain the empty string.

MVBasic provides four @ sign variables that are reserved as user-defined variables. These are @USER1, @USER2, @USER3, and @USER4. By default they contain the empty string.

MVBasic also supports three other types of user-defined variables: local variables, global variables, and process-private global variables. For further details, refer to the Variables page in this manual.

A special case of these @ variables are the dynamic array level delimiter characters. These always contain the same 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.

Caché MVBasic supports most, but not all of the system variables provided by other flavors of MultiValue. The *MultiValue Basic Quick Reference* contains an @-Variables table showing which UniVerse system variables are supported by Caché MVBasic. The following are the Caché MVBasic system variables:

@ACCOUNT	The current account name, which is usually identical to the Caché namespace name. Note that the SYSPROG account corresponds to the %SYS namespace.
@ANS	The answer of the last-executed ITYPE() function. This value is user modifiable.
@AUTHORIZATION	The current Caché username. This is the same value that is returned by *%SYS.ProcessQuery.UserName*. This is the username used by the MultiValue Shell start-up routine to match with a corresponding Proc or Paragraph name.
@COMMAND	The command line that invoked this program. (See Note 2.) This value is user modifiable.
@COMMAND.STACK	The contents of the command stack, a list of commands issued from the MultiValue Shell as a dynamic array (with @FM delimiters) with most recent items first. See also SYSTEM(33) function.
@CONV	Used by the CALCULATE function to hold the *code* string for the OCONV function. This value is user modifiable.
@CRTHIGH	Number of lines displayed in the terminal window.
@CRTWIDE	Number of columns displayed in the terminal window.
@DATA	Returned data, separated by CHAR(13) (carriage return) characters. Provided for compatibility with UniData systems.
@DATA.PENDING	Returned data, separated by @AM (CHAR(254)) delimiter characters.
@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.) This value is user modifiable.

@DAY	The day of the month when the current process started, specified as an integer. (See Note 1.) This value is user modifiable.
@DICT	You must set @DICT to the dictionary of the file you will specify in the CALCULATE function. This value is user modifiable.
@FALSE	The 0 character, representing the boolean value.
@FILE.NAME	Same as @FILENAME. This value is user modifiable.
@FILENAME	The pathname specified in the most recent invocation of **OPENSEQ**. If the sequential file exists, @FILENAME contains *pathname* as a file pathname. If the sequential file does not exist, @FILENAME contains *pathname* 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. This value is user modifiable.
@FORMAT	Used by the CALCULATE function to hold the *format* string for the FMT function. This value is user modifiable.
@HEADER	Used by the CALCULATE function to hold the *header* string for the HEADING command. This value is user modifiable.
@ID	Current record ID. This value is user modifiable.
@IO.ERROR	The error status of the most recent failed I/O operation. Is not modified by a successful I/O operation. This value is user modifiable. Compare with @IO.STATUS.
@IO.STATUS	The status of the most recent I/O operation. Contains 0 if the I/O operation succeeded. Contains an error code if the I/O operation failed. This value is user modifiable. Compare with @IO.ERROR.
@ITYPECACHE	This value is user modifiable.
@LASTLOGONPROG	The name of the last LOGON procedure, paragraph, or program executed. Updated when a process starts the MV Shell or changes accounts using the LOGTO command. Only valid for processes started from the MV Shell.
@LEVEL	Nested level of execution. Starts at 0. Reset by ABORT commands.
@LOGNAME	The operating system user login name. This is the same value that is returned by *%SYS.ProcessQuery.OSUserName*. @LOGNAME and @USER are synonyms. Also see @AUTHORIZATION.
@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.
@ME	A handle to the object reference from within an instance, method, or property. Contains the current class context. Same as the $THIS special variable in ObjectScript. @ME can reference a multidimensional property using arrow syntax, as follows: @ME->property(subscript). @ME itself is not modifiable, but @ME->property can be used wherever a command uses a SETTING clause to set an output variable.

@MONTH	The month of the year when the current process started, specified as an integer. (See Note 1.) This value is user modifiable.
@MV	The current value counter, only used for columnar listing. Used only in I-descriptors. Same as @NV. This value is user modifiable.
@NB	Current BREAK level number. 1 is the lowest-level break. Used only in I-descriptors. This value is user modifiable.
@ND	Number of detail lines since the last BREAK on a break line. Used only in I-descriptors. This value is user modifiable.
@NI	Current item counter (the number of items listed or selected to Select List 0). Used only in I-descriptors. Same as @RECCOUNT. This value is user modifiable.
@NS	Current subvalue counter for columnar listing only. Used only in I-descriptors. This value is user modifiable.
@NV	The current value counter, only used for columnar listing. Used in I-descriptors. Same as @MV. This value is user modifiable.
@PARASENTENCE	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.
@PORTNO	The current port number, specified as an integer.
@QWHO	The current account name when a program is run using the MV Shell. @QWHO retains the name of a Q pointer account if that was the login account. @QWHO does not track account changes caused by changing namespaces using ObjectScript commands. For the current account name for programs run using CSP or class methods, use @WHO.
@RECCOUNT	The current item counter (the number of items listed or selected). Used only in I-descriptors. Same as @NI. This value is user modifiable.
@RECORD	The current record. This value is user modifiable.
@RECURn	A set of variables: @RECUR0, @RECUR1, @RECUR2, @RECUR3, and @RECUR4. These values are user modifiable.
@SELECTED	Number of elements selected from the most recent select list. See the SELECT statement. Defaults to 0. This value is user modifiable. If **$OPTIONS FSELECT** is set, @SELECTED returns the number of elements selected; If **$OPTIONS FSELECT** is not set, @SELECTED always returns 1. See also SYSTEM(11) function.
@SENTENCE	The command line that invoked this program. (See Note 2)
@STDFIL	Standard file. The file opened to the default file variable.
@SYS.BELL	The ASCII bell character (CHAR(7)). Printing this variable rings the bell.
@SYSTEM.RETURN.CODE	Status code for system processes. Returns 0 for success, -1 for error. This variable is not set by I/O operations; I/O operations set @IO.STATUS and @IO.ERROR. This value is user modifiable.
@SYSTEM.SET	Status code for system processes. This value is user modifiable.
@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). (When issued from the MV terminal shell, this is the time that the last command line was invoked.) @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. This value is user modifiable.
@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`. This value is user modifiable.
@UDTNO	Terminal Number. A unique integer assigned to a terminal job. Corresponds to the port number returned by the **LISTME** MultiValue command. Exiting and re-entering the MV shell does not change this integer value. Provided for compatibility with UniData systems.
@UID	User ID (uid) on a UNIX® system. Returns 0 on non-UNIX® systems. Provided for compatibility with UniData systems.
@USER	The operating system user login name. This is the same value that is returned by *%SYS.ProcessQuery.OSUserName*. @USER and @LOGNAME are synonyms. Provided for D3 compatibility.
@USER.NO	Same as @USERNO.
@USERNO	The port number of the current process. (See Note 3)
@USER.RETURN.CODE	Status code for user processes. This value is user modifiable.
@USER.TYPE	Returns 0 if the Caché process is an interactive terminal. Returns 1 if the Caché process is a MultiValue phantom process. Returns 2 if neither of the above, for example a process started via the JOB statement. Provided for compatibility with UniData systems.
@WHO	Name of the current account. @WHO tracks account changes caused by changing namespaces using ObjectScript commands. See also @QWHO.
@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 *Operational Differences Between MultiValue and Caché*. (See Note 1.) This value is user modifiable.
@YEAR4	The year when the current process started, specified as four digits. (See Note 1.) This value is user modifiable.

Note 1

This variable is computed when a program is started and does not change during execution. Time and date variables represent local time and date. Caché MultiValue determines local time and date as follows:

- It determines the current Coordinated Universal Time (UTC) from the system clock.
- It adjusts UTC to the local time zone by using the value of the Caché special variable $ZTIMEZONE.
- It applies local time variant settings (such as Daylight Saving Time) for that time zone from the host operating system.

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.

See Also

- $MVname special variables in the *Caché ObjectScript Reference*
- $MVV(n) special variables in the *Caché ObjectScript Reference*

User Variables

User-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.
- Four user-defined @ variables: @USER1, @USER2, @USER3, and @USER4. The scope of these variables is the current process. By default they contain the empty string.
- System variables, identified by an @ character as the first character. These variables are generally not user-modifiable. For further details, refer to the System Variables page in this manual.

Naming Conventions

The following are the naming conventions for local variables:

- MVBasic does not, strictly speaking, have reserved words. Therefore, a local variable may have the same name as a function or a command. For clarity of code, it is strongly suggested that you avoid the names listed in the Reserved Words table in the *MultiValue Basic Quick Reference*.
- The first character of a local variable name must be a letter, dollar ($), or percent (%) character. Names beginning with $SYSTEM. (in any letter case) are reserved as system elements. Certain names beginning with $ are used as command or function names in MVBasic (for example, $DATA, $GET, $KILL, $LIST, $MERGE, $ORDER) and should be avoided. Names beginning with a % character (except those beginning with %Z or %z) are reserved as system elements. For further details, refer to "Rules and Guidelines for Identifiers" in the *Caché Programming Orientation Guide*.
- The second and subsequent characters a local variable name may be letters, numbers, the period (.), dollar ($), underscore (_), and percent (%) characters. The last character cannot be an underscore (_) character.
- Letters in local variable names are case-sensitive in Caché and in all MultiValue emulations *except* D3. D3 only uses all-uppercase variable names. A local variable name defined with lowercase letters in Caché mode is considered undefined in D3 mode. A local variable name defined with lowercase letters in D3 mode is converted to all capital letters. Case sensitivity can be configured using **$OPTIONS CASE** (to turn on case sensitivity) and **$OPTIONS -CASE** (to turn off case sensitivity).
- 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.

Be aware valid local variable names in MVBasic may not be valid in ObjectScript. For example, in ObjectScript all variable names that begin with a dollar ($) character are system-supplied special variables. For this reason, MVBasic local variable names containing punctuation characters should be avoided whenever possible.

You can use the ObjectScript $ZNAME function to validate an MVBasic local variable name. If you specify `$ZNAME(string,0,11)`, **$ZNAME** validates *string* using the MultiValue Basic naming conventions for local variables.

A global variable begins with the caret (^) character, indicating that it is a global variable. A global variable follows the 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 ObjectScript naming conventions, not the MultiValue variable naming conventions. For further details, see the Variables chapter of *Using Caché ObjectScript*.

Note that in D3 emulation, global variable names and process-private global variable names are case-sensitive, but local variable names are not case-sensitive.

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.

Common Storage Areas

Caché MVBasic allows you to group user variables into common storage areas. This permits you to set, limit access to, or clear multiple variables with a single command. For further details, refer to the COMMON, CLEARCOMMON, and CLEAR commands.

Undefined Variables

By default, if an MVBasic routine references an undefined variable, the system generates an <UNDEFINED> error. That is, MVBasic undefined variables are handled the same as Caché undefined variables. They are subject to the current settings of the Management Portal **UndefVarBehavior** configuration setting, and the *Undefined* property of the Config.Miscellaneous class and **Undefined()** method of the %SYSTEM.Process class settings. All of these default to generating an <UNDEFINED> error. Go to the Management Portal, select **[System] > [Configuration] > [Compatibility Settings]**. View and edit the current setting of **Undefined**. The default is 0 (always throw an error).

You can change this MVBasic default behavior to substitute an empty string for an undefined variable, without signalling an error. You can do this in either of two ways:

- You can use the Caché *MVDefined* property of the Config.Miscellaneous class to set this behavior on a system wide basis, or the **MVUndefined()** method of the %SYSTEM.Process class to set this behavior for the current process. Setting these values affects only the handling of MVBasic undefined functions; the handling of Caché undefined functions is unchanged.
- You can use the ObjectScript **Undefined()** method of the %SYSTEM.Process class to change undefined variable behavior for the current process. This causes all variables (both Caché variables and MVBasic variables) to substitute an empty string.

To set these ObjectScript functions, use the MVBasic $XECUTE statement, as shown in the following example:

```
$XECUTE "DO $ZUTIL(68,72,1)"
```

Maximum Number of Variables

A Caché MVBasic routine can contain a maximum of 32,759 private variables, and a maximum of 65,280 public variables. Exceeding these limits results in a compile error.

VOC Format

The format for VOC entries.

Description

The VOC plays a vital part in the MV command processor and the MV file I/O system. Every word found on a command line and every filename opened is looked up in the VOC to determine its meaning or location.

For new and imported accounts, the VOC is created in a specific global, MV.VOC, and is populated with the contents of the NEWACC file stored in sysprog. All accesses to the VOC by system code will go directly to the global ^MV.VOC whereas any accesses by user code will open the file "VOC" or "MD" and access the global specified in the F pointer (so don't change them from pointing to ^MV.VOC).

VOC Entry Format

The first attribute of a VOC item indicates the type of the item:

"F"	File. You can use the LISTF command to list VOC file entries.
"K"	Keyword. Keywords used in CMQL are listed in the *Caché MultiValue Query Language (CMQL) Reference*.
"PA"	Paragraph. You can use the LISTPA command to list VOC paragraph entries.
"PH"	Phrase. You can use the LISTPH command to list VOC phrase entries.
"PQ"	PROC command
"PQN"	New PROC command
"Q"	Q pointer
"R"	Remote
"S"	Sentence. You can use the LISTS command to list VOC sentence entries.
"V"	Verb

File Definition Format

File Definition

```
001 F
002 data section
003 dict section
004 M if this is a multi data section file
005 primary class name
006 file options
007 list of data section names
008 list of locations of data sections
009 list of class names for data sections
```

The dict and data sections may contain either global references or directory paths.

The file options are any combination of these chars:

- B — Basic program file. Used to control display in Studio.

- U — Untranslated. Item names in a directory file are not passed through the standard translation algorithm to create the filename.

Keyword Definition Format

Keyword Definition

```
001 "K"
002 Canonical form of the keyword. This is the text that is passed to the parser.
003 Alternate VOC entry to use if this word is used as a verb.
```

Verb Definition Format

Verb Definition

```
001 V
002 Function name
003 Location code
004 Processing Options
005 Parsing Options
006
007 BasicObjectPointer Location
008 ItemID of Basic Program
009 Filename of Basic Program
010 Account of Basic Program
```

The Location code directs the shell to the correct routine.

The Processing options are a set of single character codes passed to CMQL:

- "A" - output processor uses default output attributes
- "D" - output processor requires a datastream
- "I" - output processor requires entire items
- "J" - output processor requires total,avg and enum (STAT verb)
- "L" - output processor requires a selectlist
- "R" - output processor requires the raw data only, no conversions or formatting
- "S" - output should be sorted by default
- "T" - output processor requires totals only (SUM verb)

The Parsing options control the shell's operation:

- 2 TCL2 format
- A Default to All items if none specified by select list or on the command line
- B Backslashes (\ ) can be used to quote strings
- C Read Item into @Record
- D force read from the Dict of the file.
- E Handling of CRT/PRINT and error messages differs slightly for some verbs, usually CMQL verbs
- F Filename only. Do not look for item IDs.
- J Inhibit TCL1 parse - pass unprocessed line to verb implementation code.
- K basic program is to keep select list 0 available to TCL after execution.
- L Remove parenthesis from the line - for CLEAR-FILE (DATA XXX)
- M Process error Messages before returning the next item.

- N New Item OK
- P Print out item IDs (TCL2)
- Q Query syntax
- R Require Item IDs to be specified. Overrides D3 behavior of assuming '*' if item spec missing.
- U Take lock for Update