Caché ObjectScript Reference

Version 2018.1
2019-09-20
# Table of Contents

**About This Book** ........................................................................................................................................ 1

**Symbols and Abbreviations** ................................................................................................................. 3
  Symbols Used in ObjectScript ................................................................................................................ 4
  Abbreviations Used in ObjectScript ....................................................................................................... 14

**ObjectScript Commands** ......................................................................................................................... 19
  BREAK .................................................................................................................................................. 20
  CATCH .............................................................................................................................................. 25
  CLOSE ............................................................................................................................................... 31
  CONTINUE ....................................................................................................................................... 33
  DO ..................................................................................................................................................... 35
  DO WHILE ...................................................................................................................................... 43
  ELSE .............................................................................................................................................. 46
  ELSEIF .......................................................................................................................................... 47
  FOR ............................................................................................................................................... 48
  GOTO ............................................................................................................................................. 56
  HALT ............................................................................................................................................ 60
  HANG .......................................................................................................................................... 63
  IF .................................................................................................................................................... 66
  JOB ................................................................................................................................................. 70
  KILL ............................................................................................................................................... 82
  LOCK ............................................................................................................................................ 87
  MERGE ......................................................................................................................................... 98
  NEW ............................................................................................................................................. 101
  OPEN ........................................................................................................................................... 106
  QUIT ............................................................................................................................................... 112
  READ ........................................................................................................................................... 118
  RETURN ..................................................................................................................................... 126
  SET ............................................................................................................................................... 131
  TCOMMITE .................................................................................................................................. 141
  THROW ......................................................................................................................................... 143
  TROLLBACK ................................................................................................................................ 148
  TRY .............................................................................................................................................. 152
  TSTART ....................................................................................................................................... 156
  USE ............................................................................................................................................... 159
  VIEW ............................................................................................................................................ 164
  WHILE ......................................................................................................................................... 168
  WRITE .......................................................................................................................................... 173
  XECUTE ....................................................................................................................................... 183
  ZKILL ........................................................................................................................................... 188
  ZNSPACE .................................................................................................................................... 190
  ZTRAP .......................................................................................................................................... 195
  ZWRITE ...................................................................................................................................... 198
  ZZDUMP ..................................................................................................................................... 204
  ZZWRITE .................................................................................................................................. 206

**Routine and Debugging Commands** ......................................................................................................... 209
  PRINT ........................................................................................................................................... 210
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZBREAK</td>
<td>213</td>
</tr>
<tr>
<td>ZINSERT</td>
<td>220</td>
</tr>
<tr>
<td>ZLOAD</td>
<td>223</td>
</tr>
<tr>
<td>ZPRINT</td>
<td>226</td>
</tr>
<tr>
<td>ZREMOVE</td>
<td>229</td>
</tr>
<tr>
<td>ZSAVE</td>
<td>232</td>
</tr>
<tr>
<td>ZZPRINT</td>
<td>235</td>
</tr>
<tr>
<td><strong>ObjectScript Functions</strong></td>
<td></td>
</tr>
<tr>
<td>$ASCII</td>
<td>240</td>
</tr>
<tr>
<td>$BIT</td>
<td>243</td>
</tr>
<tr>
<td>$BITCOUNT</td>
<td>247</td>
</tr>
<tr>
<td>$BITFIND</td>
<td>249</td>
</tr>
<tr>
<td>$BITLOGIC</td>
<td>251</td>
</tr>
<tr>
<td>$CASE</td>
<td>254</td>
</tr>
<tr>
<td>$CHAR</td>
<td>257</td>
</tr>
<tr>
<td>$CLASSMETHOD</td>
<td>261</td>
</tr>
<tr>
<td>$CLASSNAME</td>
<td>263</td>
</tr>
<tr>
<td>$COMPILE</td>
<td>265</td>
</tr>
<tr>
<td>$DATA</td>
<td>269</td>
</tr>
<tr>
<td>$DECIMAL</td>
<td>273</td>
</tr>
<tr>
<td>$DOUBLE</td>
<td>276</td>
</tr>
<tr>
<td>$EXTRACT</td>
<td>281</td>
</tr>
<tr>
<td>$FACTOR</td>
<td>288</td>
</tr>
<tr>
<td>$FIND</td>
<td>290</td>
</tr>
<tr>
<td>$NUMBER</td>
<td>293</td>
</tr>
<tr>
<td>$GET</td>
<td>298</td>
</tr>
<tr>
<td>$INCREMENT</td>
<td>301</td>
</tr>
<tr>
<td>$NUMBER</td>
<td>306</td>
</tr>
<tr>
<td>$OBJECT</td>
<td>309</td>
</tr>
<tr>
<td>$VALIDDOUBBLE</td>
<td>312</td>
</tr>
<tr>
<td>$VALIDNUM</td>
<td>314</td>
</tr>
<tr>
<td>$JUSTIFY</td>
<td>318</td>
</tr>
<tr>
<td>$LENGTH</td>
<td>323</td>
</tr>
<tr>
<td>$LIST</td>
<td>326</td>
</tr>
<tr>
<td>$LISTBUILD</td>
<td>329</td>
</tr>
<tr>
<td>$LISTDATA</td>
<td>338</td>
</tr>
<tr>
<td>$LISTFIND</td>
<td>343</td>
</tr>
<tr>
<td>$LISTFROMSTRING</td>
<td>346</td>
</tr>
<tr>
<td>$LISTGET</td>
<td>349</td>
</tr>
<tr>
<td>$LISTLENGTH</td>
<td>351</td>
</tr>
<tr>
<td>$LISTNEXT</td>
<td>354</td>
</tr>
<tr>
<td>$LISTSAME</td>
<td>356</td>
</tr>
<tr>
<td>$LISTSTRING</td>
<td>358</td>
</tr>
<tr>
<td>$LISTUPDATE</td>
<td>362</td>
</tr>
<tr>
<td>$LISTVALID</td>
<td>365</td>
</tr>
<tr>
<td>$LOCATE</td>
<td>368</td>
</tr>
<tr>
<td>$MATCH</td>
<td>370</td>
</tr>
<tr>
<td>$METHOD</td>
<td>370</td>
</tr>
<tr>
<td>$NAME</td>
<td>377</td>
</tr>
<tr>
<td>$NCONVERT</td>
<td>380</td>
</tr>
<tr>
<td>MultiValue Commands, Functions, and Special Variables</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------------------</td>
<td>--</td>
</tr>
<tr>
<td>MV</td>
<td>744</td>
</tr>
<tr>
<td>MVCALL</td>
<td>746</td>
</tr>
<tr>
<td>MVCRT</td>
<td>747</td>
</tr>
<tr>
<td>MVDIM</td>
<td>748</td>
</tr>
<tr>
<td>MVPRINT</td>
<td>750</td>
</tr>
<tr>
<td>$CHANGE</td>
<td>751</td>
</tr>
<tr>
<td>$MV</td>
<td>752</td>
</tr>
<tr>
<td>$MVT</td>
<td>754</td>
</tr>
<tr>
<td>$MVFMT</td>
<td>755</td>
</tr>
<tr>
<td>$MVFMTS</td>
<td>756</td>
</tr>
<tr>
<td>$MVICONV</td>
<td>757</td>
</tr>
<tr>
<td>Structured System Variables</td>
<td>729</td>
</tr>
<tr>
<td>^$GLOBAL</td>
<td>730</td>
</tr>
<tr>
<td>^$JOB</td>
<td>733</td>
</tr>
<tr>
<td>^$LOCK</td>
<td>735</td>
</tr>
<tr>
<td>^$ROUTE</td>
<td>740</td>
</tr>
<tr>
<td>$ROLES</td>
<td>657</td>
</tr>
<tr>
<td>$STACK</td>
<td>659</td>
</tr>
<tr>
<td>$STORAGE</td>
<td>661</td>
</tr>
<tr>
<td>$SYSTEM</td>
<td>664</td>
</tr>
<tr>
<td>$TEST</td>
<td>673</td>
</tr>
<tr>
<td>$THIS</td>
<td>675</td>
</tr>
<tr>
<td>$THROWOBJ</td>
<td>676</td>
</tr>
<tr>
<td>$TLEVEL</td>
<td>677</td>
</tr>
<tr>
<td>$USERNAME</td>
<td>679</td>
</tr>
<tr>
<td>$X</td>
<td>680</td>
</tr>
<tr>
<td>$Y</td>
<td>682</td>
</tr>
<tr>
<td>$ZA</td>
<td>684</td>
</tr>
<tr>
<td>$SB</td>
<td>687</td>
</tr>
<tr>
<td>$SZCHILD</td>
<td>690</td>
</tr>
<tr>
<td>$SZEOF</td>
<td>692</td>
</tr>
<tr>
<td>$SZEOS</td>
<td>693</td>
</tr>
<tr>
<td>$SZERROR</td>
<td>694</td>
</tr>
<tr>
<td>$ZHOROLOG</td>
<td>700</td>
</tr>
<tr>
<td>$ZIOL</td>
<td>701</td>
</tr>
<tr>
<td>$ZJOB</td>
<td>702</td>
</tr>
<tr>
<td>$ZMODE</td>
<td>704</td>
</tr>
<tr>
<td>$ZNAME</td>
<td>705</td>
</tr>
<tr>
<td>$ZNSPACE</td>
<td>706</td>
</tr>
<tr>
<td>$ZORDER</td>
<td>708</td>
</tr>
<tr>
<td>$ZPARENT</td>
<td>709</td>
</tr>
<tr>
<td>$ZPI</td>
<td>710</td>
</tr>
<tr>
<td>$ZPOS</td>
<td>711</td>
</tr>
<tr>
<td>$ZREFERENCE</td>
<td>712</td>
</tr>
<tr>
<td>$ZSTORAGE</td>
<td>716</td>
</tr>
<tr>
<td>$ZTIMESTAMP</td>
<td>717</td>
</tr>
<tr>
<td>$ZTIMEZONE</td>
<td>720</td>
</tr>
<tr>
<td>$ZTRAP</td>
<td>724</td>
</tr>
<tr>
<td>$ZVERSION</td>
<td>728</td>
</tr>
</tbody>
</table>
List of Figures

Figure B–1: Client/Server Connections in the Non-Concurrent and Concurrent Modes ................. 79
Figure B–2: Initial Structure of ^X and ^Y .................................................................................... 99
Figure B–3: Result on ^X and ^Y of MERGE Command .............................................................. 100
List of Tables

Table E–1: Non-Configuration-Related Functions ................................................................. 619
Table E–2: Process-Related Configuration Functions ........................................................ 623
Table E–3: System-Related Configuration Functions ......................................................... 626
Table F–1: Existing Flags .................................................................................................... 665
Table F–2: Compiler Qualifiers ....................................................................................... 666
Table F–3: Export Qualifiers ............................................................................................. 668
Table F–4: ShowClassAndObject Qualifiers ................................................................... 669
Table F–5: UnitTest ........................................................................................................ 669
Table F–6: Flag Qualifier Mapping .................................................................................. 670
About This Book

This book provides reference material for various elements of the ObjectScript language: commands, functions, and special variables, and tables of abbreviations of their names, and symbols used in the language.

This book contains the following sections:

- Symbols and Abbreviations
- Commands
- Functions
- Special Variables
- Structured System Variables

There is also a detailed Table of Contents, and sections dedicated to MultiValue-specific and Legacy commands, functions, and special variables.

Other related topics in the Caché documentation set are:

- Using Caché ObjectScript
- Using Caché Objects, particularly the chapter “Working with Registered Objects” and the appendix “Object-Specific ObjectScript Features”.
- Caché I/O Device Guide

For general information, see Using InterSystems Documentation.
Symbols and Abbreviations
Symbols Used in ObjectScript

A table of characters used in ObjectScript as operators, prefixes, and so on.

Table of Symbols

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

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name and Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>[space] or [tab]</td>
<td>White space (Tab (9) or Space (32)): Leading white space (space or tab) is required before every line of code, with the exceptions of labels, and some comment lines. Within commands, one (and only one) space is required between the command name and the first argument. Trailing white space (space or tab) is required between the last command argument and any following command or comment on the same line. Trailing whitespace is also required between a label and a following command or comment on the same line.</td>
</tr>
<tr>
<td>[two spaces, two tabs, or a space and a tab]</td>
<td>Double white space: Trailing double white space required between an argumentless command and the next command on the same line.</td>
</tr>
<tr>
<td>!</td>
<td>Exclamation mark (33): OR logical operator (full evaluation). In READ and WRITE commands, specifies a new line. As first character at terminal prompt, load interactive subshell.</td>
</tr>
<tr>
<td>&quot;</td>
<td>Quotes (34): Used to enclose string literals. In Dynamic SQL used to enclose the SQL code as a string argument of the %Prepare() method. For differences between straight quotes (&quot;) and directional quotes (&quot; &quot;). see Pattern Matching in Using Caché ObjectScript.</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quotes: Used to specify the null string (&quot;&quot;), which is a zero-length string. Used to specify a literal quote character within a quoted string.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Name and Usage</td>
</tr>
<tr>
<td>--------</td>
<td>----------------</td>
</tr>
<tr>
<td>#</td>
<td><strong>Pound sign (35):</strong> Modulo division operator. Can be used to determine a bit value. For example, $ZA#2 returns the 1’s bit value (0 or 1); with the integer divide () operator $ZJOB\1024#2 returns the 1024’s bit value (0 or 1). In <strong>READ</strong> and <strong>WRITE</strong> commands, form feed. In fixed-length <strong>READ</strong>, number of characters to read. Prefix for referencing the value of a class parameter from within the class: #ParameterName. Prefix for many <strong>macro preprocessor directives</strong> such as: #Define, #Include, and #If. See also ##. In <strong>class syntax</strong>, parameter prefix used to return the <strong>parameter value</strong>. For example, ##class(%Library.Boolean).#XSDTYPE of myinstance.#EXTENTQUERYSPEC. In <strong>ZBREAK</strong> debugging, an iteration counter for disabling a specified breakpoint or watchpoint. For example, the following disables the breakpoint at label^rou for 100 iterations: ZBREAK -label^rou#100. Delimiter for a <strong>runtime expression</strong>: #(expr)#, where expr is an ObjectScript expression. In the callout routine ZFENTRY, an argtype prefix indicating a DOUBLE data type: #D or #F. <strong>Regular expression</strong> end-of-line comment indicator (in (?x) mode only).</td>
</tr>
<tr>
<td>##</td>
<td><strong>Double pound sign:</strong> Object <strong>class invocation prefix</strong>: ##class(classname).methodname() or ##class(classname).#parametername. ##super() syntax is used to invoke an overridden superclass method. Prefix for certain <strong>macro preprocessor directives</strong>, including ##Continue, ##Expression, ##Function, ##Lit, ##SQL, and ##Unique. ##SQL is invoked to execute a line of SQL code from within ObjectScript: ##SQL(SQL command). Delimiter for a <strong>compile-time expression</strong>: ##(expr)##, where expr is an ObjectScript expression.</td>
</tr>
<tr>
<td>##;</td>
<td><strong>Double pound sign semicolon:</strong> <strong>Single-line comment</strong> indicator; can be used in column 1 in either ObjectScript or Embedded SQL.</td>
</tr>
<tr>
<td>#;</td>
<td><strong>Pound sign semicolon:</strong> <strong>Single-line comment</strong> indicator; can be used in column 1.</td>
</tr>
<tr>
<td>$</td>
<td><strong>Dollar sign (36):</strong> system function prefix: $name(parameters). <strong>Special variable</strong> prefix: $name. $Znnn (a name beginning with $Z) can be a user-defined function or special variable defined using %ZLANG language extension library. It can also be an InterSystems-supplied system function or special variable. <strong>Regular expression</strong> end of string anchor; for example, (USA)$ In <strong>ZBREAK</strong> debugging, a single-step breakpoint. As first character at terminal prompt, <strong>load interactive subshell</strong>.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Name and Usage</td>
</tr>
<tr>
<td>--------</td>
<td>----------------</td>
</tr>
<tr>
<td>$$</td>
<td><em>Double dollar sign</em>: user-supplied function call prefix: $$myname(parameters). $$ is returned by $STACK when context was established by a reference to a user-supplied function. Prefix to a routine name to <em>directly invoke</em> that routine.</td>
</tr>
<tr>
<td>$$$</td>
<td><em>Triple dollar sign</em>: Macro invocation prefix.</td>
</tr>
<tr>
<td>%</td>
<td><em>Percent sign</em> (37): Permitted as first character of names: (1) local variable names, indicating a &quot;% variable&quot; with special scoping rules, used for locking. (2) routine names, often indicates a system utility. (3) Package class names, such as %SYSTEM.class and %Library.class, as well as class names within the %Library package, including data types such as %String. (4) %Persistent object property names and method names, such as %Dialect, %New(), and %OpenId(). A %On... method name is a <em>callback method</em>. (5) Labels. Required as first character of a <em>macro argument</em>. Prefix for some embedded SQL variables: %msg, %ROWCOUNT, and for some SQL keywords: %STARTSWITH. See % (instance variable).</td>
</tr>
<tr>
<td>%%%</td>
<td><em>Double percent sign</em>: Prefix for the pseudo-field reference variable keywords %%CLASSNAME, %%CLASSNAMEQ, %%ID, and %%TABLENAME, used in ObjectScript computed field code and <em>trigger code</em>.</td>
</tr>
<tr>
<td>&amp;</td>
<td><em>Ampersand</em> (38): AND logical operator (full evaluation). $BITLOGIC bitstring AND operator. In a formal parameter list, an optional, non-functional <em>variable name prefix</em> that marks a parameter as one that should be passed by reference. The &amp; is a marker and is not part of the variable name. For example Calc(x,&amp;y). Shell invocation prefix for embedded code. For example &amp;sql(<em>SQL commands</em>); &amp;js&lt;JavaScript commands&gt;; &amp;html&lt;HTML commands&gt;. UNIX® batch command.</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td><em>Double ampersand</em>: AND logical operator (partial evaluation). Regular expression AND logical operator.</td>
</tr>
<tr>
<td>'</td>
<td><em>Apostrophe</em> (39): Unary Not operator. Can be combined with: logical operators `&amp; (Not And),</td>
</tr>
<tr>
<td>Symbol</td>
<td>Name and Usage</td>
</tr>
<tr>
<td>--------</td>
<td>----------------</td>
</tr>
<tr>
<td>( )</td>
<td>Parentheses ((40,41)): Used to enclose a procedure or function parameter list. Parentheses are mandatory, even when empty.</td>
</tr>
<tr>
<td></td>
<td>Used to nest expressions; nesting overrides the Caché default of strict left-to-right evaluation of operators, and allows you to give precedence to expressions.</td>
</tr>
<tr>
<td></td>
<td>Used to specify array subscripts for a local variable: (a(1,1)), a global variable: (^a(1,1)), or a process-private global: (^</td>
</tr>
<tr>
<td></td>
<td>Used to enclose an alternating pattern match (following a ?).</td>
</tr>
<tr>
<td></td>
<td>With NEW and KILL commands, exclusive (everything but) indicator.</td>
</tr>
<tr>
<td></td>
<td>For postconditionals, required if postconditional contains a space.</td>
</tr>
<tr>
<td></td>
<td>Used to enclose embedded SQL code, following an &amp;sql shell invocation command: &amp;sql(SQL commands).</td>
</tr>
<tr>
<td></td>
<td>Regular expression match string (Boston) or string list (Boston</td>
</tr>
<tr>
<td></td>
<td>When setting a JSON object or array value, used to enclose a Caché literal or expression.</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk ((42)): Multiplication operator.</td>
</tr>
<tr>
<td></td>
<td>In $ZSEARCH, wild card for zero, one, or more than one characters.</td>
</tr>
<tr>
<td></td>
<td>In $EXTRACT, $LIST, and $PIECE specifies the final item at the end of the string; can be used with a signed integer to specify offset from the end, for example (*-2, +1).</td>
</tr>
<tr>
<td></td>
<td>In WRITE command, specifies an integer code for a character. For example, WRITE (*65) writes the letter “A”.</td>
</tr>
<tr>
<td></td>
<td>As prefix to $ZTRAP string value, specifies that call stack level should be left unchanged.</td>
</tr>
<tr>
<td></td>
<td>In ZBREAK a name prefix denoting a local variable. In certain error codes returned to $ZERROR, a name prefix denoting an undefined local variable, class, method, or property.</td>
</tr>
<tr>
<td></td>
<td>Regular expression 0 or more character quantifier.</td>
</tr>
<tr>
<td>**</td>
<td>Double asterisk: Exponentiation operator. For example, (4^{**3}=64).</td>
</tr>
<tr>
<td>*+</td>
<td>Asterisk plus: In $EXTRACT, $LIST, $PIECE specifies offset beyond the last item of the string; used to append values. For example, (*+1) appends an item to the end of string.</td>
</tr>
<tr>
<td>*-</td>
<td>Asterisk minus: In WRITE command, specifies a device control integer code. For example, WRITE (*-10) clears the terminal input buffer.</td>
</tr>
<tr>
<td></td>
<td>In $EXTRACT, $LIST, $LISTGET, and $PIECE specifies offset backwards from the last item of the string; for example, (*-1) is the next-to-last item.</td>
</tr>
<tr>
<td>*/</td>
<td>Asterisk slash: Multi-line comment ending indicator. Comment begins with /*.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Name and Usage</td>
</tr>
<tr>
<td>--------</td>
<td>----------------</td>
</tr>
<tr>
<td>+</td>
<td><strong>Plus sign (43):</strong> Unary arithmetic positive operator. When appended to a string or a function that returns a string forces numeric evaluation; for example, <code>WRITE +&quot;007.0&quot;</code> or <code>WRITE +$PIECE(str&quot;,&quot;&quot;,2)</code>. Addition operator. Integer line count offset from a label: <code>label+offset</code>. In <code>$ZTRAP</code>, integer line count offset from top of a procedure: <code>+offset^procname</code>. With <code>LOCK</code> and <code>ZBREAK</code> commands, a prefix that enables or applies/increments the item that follows. <strong>Regular expression</strong> 1 or more character quantifier.</td>
</tr>
<tr>
<td>+=</td>
<td><strong>Plus sign, Equal sign</strong> In commands and functions that output to a file, means that output data is appended to the existing file contents. Just an equal sign means that output data overwrites the contents of an existing file. See <code>$ZF(-100)</code>.</td>
</tr>
<tr>
<td>,</td>
<td><strong>Comma (44):</strong> In functions and procedures, multiple parameters separator. In commands, multiple arguments delimiter. In array variables, subscript levels separator. American numeric group separator or European decimal point character (configurable). In <code>$ECODE</code>, surround error code: <code>.M7</code>,</td>
</tr>
<tr>
<td>,,</td>
<td><strong>Two commas:</strong> In functions, a placeholder for an unspecified positional parameter (which takes a default value).</td>
</tr>
<tr>
<td>–</td>
<td><strong>Minus sign (45):</strong> Unary arithmetic negative operator. Subtraction operator. With <code>LOCK</code> and <code>ZBREAK</code> commands, a prefix that disables or decrements/removes the item that follows. <strong>Regular expression</strong> character range operator; for example <code>[A-Z]</code>.</td>
</tr>
<tr>
<td>——</td>
<td><strong>Double minus sign:</strong> With <code>ZBREAK</code> command, a prefix that removes the item that follows. <strong>Regular expression</strong> subtract (except for) logical operator.</td>
</tr>
</tbody>
</table>
### Symbols Used in ObjectScript

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name and Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td><strong>Period (46):</strong> American decimal point character, or European numeric group separator (configurable).&lt;br&gt;<strong>Object dot syntax</strong> used to refer to a method or property of an object instance: myinstance.Name.&lt;br&gt;Windows and UNIX®: As a pathname or part of a pathname, specifies the current directory. Used by $ZSEARCH.&lt;br&gt;May be included within a global name or a routine name.&lt;br&gt;Prefix to a variable or array name in an actual parameter list that specifies passing by reference: SET x=Calc(num,.result).&lt;br&gt;<strong>Pattern match repeat indicator.</strong>&lt;br&gt;<strong>Regular expression</strong> single-character wildcard.&lt;br&gt;Following argumentless DO command, a code block structure line prefix.</td>
</tr>
<tr>
<td>..</td>
<td><strong>Double period:</strong> relative dot syntax: a prefix that specifies a method or property of the current object. For example, WRITE ..foo()&lt;br&gt;Windows and UNIX®: As a pathname or part of a pathname, specifies the parent directory of the current directory. Used by $ZSEARCH.</td>
</tr>
<tr>
<td>..#</td>
<td><strong>Double period, pound sign:</strong> A prefix for references to a class parameter from within a method of the same class. For example, WRITE ..#MyParam</td>
</tr>
<tr>
<td>...</td>
<td><strong>Triple period (ellipsis):</strong> A suffix appended to the last (or only) parameter in a formal parameter list or actual parameter list that is used to specify a variable number of parameters. For example, Calc(x,y, params...). This syntax is commonly used with a dynamic dispatch method, such as Method %DispatchMethod(Method As %String,Params...)&lt;br&gt;In <strong>ZWRITE</strong> output, trailing ellipsis indicates string truncation.&lt;br&gt;In a <strong>Terminal prompt</strong>, leading characters that indicate that a long implied namespace has been truncated to its final 24 characters.&lt;br&gt;(This literal use of ellipsis in code should not be confused with format convention usage in our documentation, where ellipsis indicates that an argument can be repeated multiple times, or that a section of code is intentionally omitted.)</td>
</tr>
<tr>
<td>/</td>
<td><strong>Slash (47):</strong> Division operator (keep remainder).&lt;br&gt;In OPEN, CLOSE, and USE commands, I/O keyword parameter prefix. In READ and WRITE, device control mnemonic prefix.&lt;br&gt;With <strong>ZBREAK</strong> command, a subcommand prefix.</td>
</tr>
<tr>
<td>//</td>
<td><strong>Double slash:</strong> Single-line comment indicator.</td>
</tr>
<tr>
<td>///</td>
<td><strong>Triple slash:</strong> Single-line comment indicator. Can be used in column 1 for macro comments.</td>
</tr>
</tbody>
</table>
| /*     | **Slash asterisk:** Multi-line comment begins indicator. Comment ends with */.
## Symbols and Abbreviations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name and Usage</th>
</tr>
</thead>
</table>
| :     | *Colon (58)*: In commands, **postconditional indicator**, for example, WRITE:x=0 “nothing”.  
In commands such as **OPEN**, **USE**, **CLOSE**, **JOB**, **LOCK**, **READ**, and **ZBREAK** a placeholder separator of arguments and/or separator of parameters within an argument. For example, LOCK var1:10,+var2:15, or OPEN "|TCP|4";(:4200;"PSTE";::32767:32767)  
In **$CASE** and **$SELECT** functions, used to specify test:value paired items.  
In a **JSON object**, used to specify a key:value pair. For example, `SET JSONobj="name":"Sam"`.  
In **$JOB** special variable value, separates process ID (PID) and nodename. For example, 11368:MYCOMPUTER.  
Prefix indicating an ObjectScript **label** within embedded ObjectScript code, such as **SQL trigger code**, where the label cannot be coded in column 1. |
| ;     | *Semicolon (59)*: **Single-line comment** indicator. |
| ;;    | *Double semicolon*: **Retained single-line comment** indicator. |
| <     | *Less than (60)*: Less than operator.  
Used to enclose **Embedded JavaScript** or **Embedded HTML** code, following a &js or &html directive: &js<JavaScript commands>. |
| <=    | *Less than, Equal sign*: Less than or equal to operator. |
| '<    | *Not operator, Less than*: Greater than or equal to operator. |
| =     | *Equal sign (61)*: Equal to comparison operator.  
In **SET** command, **assignment** operator. |
| '='   | *Not operator, Equal sign*: Not equal to comparison operator |
| >     | *Greater than (62)*: Greater than operator. |
| >=    | *Greater than, Equal sign*: Greater than or equal to operator. |
| '>'   | *Not operator, Greater than*: Less than or equal to operator. |
| ?     | *Question mark (63)*: Pattern match operator.  
Regular expression 0 or 1 character quantifier suffix. **Regular expression** mode prefix. For example, (?i) case mode on; (?-i) case mode off.  
In **$ZCONVERT** translation table results, represents an untranslatable character.  
In **$ZSEARCH**, wild card for a single character.  
In **READ** and **WRITE** commands, column position indicator.  
In **ZBREAK** command, display help text.  
In Dynamic SQL, an **input parameter** variable supplied by the %**Execute()** method. |
<p>| ?#    | <em>Question mark and pound sign</em>: <strong>Regular expression</strong> embedded comment prefix. For example, (?# this is a comment). |
| @     | <em>At sign (64)</em>: <strong>Indirection operator</strong>. For subscript indirection, appears as: @array@(subscript). |</p>
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name and Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, a</td>
<td>The letter “A” (65,97): Pattern match code (following a ?).</td>
</tr>
<tr>
<td>C, c</td>
<td>The letter “C” (67,99): Pattern match code (following a ?).</td>
</tr>
<tr>
<td>E, e</td>
<td>The letter “E” (69,101): Scientific notation operator. for example, 4E3=4000. The uppercase “E” is the standard exponent operator; the lowercase “e” is a configurable exponent operator, using the ScientificNotation() method of the %SYSTEM.Process class. Pattern match code (following a ?).</td>
</tr>
<tr>
<td>I, i</td>
<td>The letter “I” (73,105): Integer indicator in $NUMBER function.</td>
</tr>
<tr>
<td>i%</td>
<td>The letter “i” percent: instance variable syntax: i%propertyname.</td>
</tr>
<tr>
<td>L, l</td>
<td>The letter “L” (76,108): Pattern match code (following a ?).</td>
</tr>
<tr>
<td>N, n</td>
<td>The letter “N” (78,110): Pattern match code (following a ?).</td>
</tr>
<tr>
<td>P, p</td>
<td>The letter “P” (80,112): Pattern match code (following a ?).</td>
</tr>
<tr>
<td>U, u</td>
<td>The letter “U” (85,117): Pattern match code (following a ?).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name and Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>[</td>
<td>Open square bracket (91): Contains operator.</td>
</tr>
<tr>
<td>[]</td>
<td>Square brackets (91,93): Used to enclose a namespace name, directory name, or the null string in an extended global reference ^[&quot;namespace&quot;]global. Used to specify a process-private global with the following syntax: ^[&quot;&quot;]ppgname or ^[&quot;&quot;, &quot;&quot;]ppgname.</td>
</tr>
<tr>
<td></td>
<td>In a structured system variable (SSVN) used to enclose a namespace name ^$[&quot;namespace&quot;]GLOBAL() to specify an extended SSVN reference.</td>
</tr>
<tr>
<td></td>
<td>In XECUTE command or a procedure definition, encloses a public variables list: [a,b,c]</td>
</tr>
<tr>
<td></td>
<td>In ZWRITE or argumentless WRITE command display, encloses an object reference (oref).</td>
</tr>
<tr>
<td></td>
<td>Regular expression match any character in a list [ABCD] or a range [A-D].</td>
</tr>
<tr>
<td></td>
<td>JSON dynamic array expression, which returns an instance of %DynamicArray. For example, SET JSONarray=[1,2,3].</td>
</tr>
<tr>
<td>[: :]</td>
<td>Square brackets and colons: Regular expression character type keyword. For example, [:alpha:].</td>
</tr>
<tr>
<td>\</td>
<td>Backslash (92): Integer division operator (drop remainder). Can be used with modulo (#) operator to determine a bit value; for example, $ZA\16#2 returns the value (0 or 1) of the $ZA 16 bit. Regular expression escape prefix. JSON string escape prefix, for example&quot;.</td>
</tr>
<tr>
<td>]</td>
<td>Close square bracket (93): Follows operator.</td>
</tr>
<tr>
<td>]]</td>
<td>Double close square brackets: Sorts After operator.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Name and Usage</td>
</tr>
<tr>
<td>--------</td>
<td>----------------</td>
</tr>
<tr>
<td>^</td>
<td><strong>Caret (94):</strong> Global variable name prefix, for example, ^myglobal(i). Routine invocation prefix, for example, DO ^routine or DO label^routine. Implied namespace prefix with the format ^system^dir. $BITLOGIC bitstring XOR (exclusive or) operator. <strong>Regular expression</strong> beginning of string anchor; for example, ^A. Regular expression character type keyword inverse. For example, [:^alpha:].</td>
</tr>
<tr>
<td>^^</td>
<td><strong>Double caret:</strong> Implied namespace prefix for the current system, with the format ^^: or ^^dirpath.</td>
</tr>
<tr>
<td>^$</td>
<td><strong>Caret dollar:</strong> Structured system variable prefix. For example, ^$GLOBAL() or ^$&quot;namespace&quot;GLOBAL().</td>
</tr>
<tr>
<td>^$[</td>
<td><strong>Caret dollar bracket:</strong> In a structured system variable (SSVN) used to enclose a namespace name ^$[&quot;namespace&quot;]GLOBAL() to specify an extended SSVN reference.</td>
</tr>
<tr>
<td>^$</td>
<td></td>
</tr>
<tr>
<td>^%</td>
<td><strong>Caret percent:</strong> System global prefix, for example, ^%utility or ^%qStream.</td>
</tr>
<tr>
<td>^{</td>
<td><strong>Caret parenthesis:</strong> A naked global reference, where the most recently named subscripted global name is implied. For example: ^(1,2)</td>
</tr>
<tr>
<td>^[</td>
<td><strong>Caret open square bracket:</strong> see Square brackets.</td>
</tr>
<tr>
<td>^</td>
<td></td>
</tr>
<tr>
<td>_</td>
<td><strong>Underscore (95):</strong> Concatenate operator. As first character of a name.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Name and Usage</td>
</tr>
<tr>
<td>--------</td>
<td>----------------</td>
</tr>
<tr>
<td>{}</td>
<td>Curly braces (123,125): Code block delimiters used in procedures, for TRY and CATCH blocks, or with the IF, FOR, DO WHILE, and WHILE commands. In SQL compute code, encloses a field name. For example, SET (Age)=18, or SET {f1} = {f2}. In XECUTE command, encloses code in which variables are treated as private. Regular expression quantifier; for example, {5} = 5 times; {3,6} = at least 3 times but not more than 6 times. Regular expression character type letter code or keyword with \p prefix; for example, \p{LL}, \p{lower}. Regular expression single character keyword with \N prefix; for example, \N{comma}. JSON dynamic object expression, which returns an instance of %DynamicObject. For example, SET JSONobj={&quot;name&quot;:&quot;Sam&quot;}.</td>
</tr>
<tr>
<td>{*}</td>
<td>Asterisk within curly braces: In SQL compute code, specifies the current SQL field name.</td>
</tr>
<tr>
<td></td>
<td>Vertical bar (124): $BITLOGIC bitstring OR operator. Regular expression OR operator. For other uses, see ^</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>Tilde (126): $BITLOGIC bitstring NOT (one's complement) operator. In Windows pathnames, indicates 8.3 compression of long names. For example: c:\PROGRA~1. To convert compressed directory names, use the NormalizeDirectory() method of the %Library.File class.</td>
</tr>
</tbody>
</table>
# Abbreviations Used in ObjectScript

A table of abbreviations for commands, functions, and special variables available in ObjectScript.

## Table of Abbreviations

The following are the name abbreviations used in ObjectScript. Most, but not all, ObjectScript commands, functions, and special variables have name abbreviations. Other uses of letters as code characters are found in the table of symbols used in ObjectScript.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A</td>
<td>$ASCII function</td>
</tr>
<tr>
<td>B</td>
<td>BREAK command</td>
</tr>
<tr>
<td>C</td>
<td>CLOSE command</td>
</tr>
<tr>
<td>$C</td>
<td>$CHAR function</td>
</tr>
<tr>
<td>D</td>
<td>DO command or DO keyword of DO WHILE command</td>
</tr>
<tr>
<td>$D</td>
<td>$DATA function (with arguments) or $DEVICE special variable (no arguments).</td>
</tr>
<tr>
<td>E</td>
<td>ELSE command (legacy version only)</td>
</tr>
<tr>
<td>$E</td>
<td>$EXTRACT function</td>
</tr>
<tr>
<td>$EC</td>
<td>$ECODE special variable</td>
</tr>
<tr>
<td>$ES</td>
<td>$ESTACK special variable</td>
</tr>
<tr>
<td>$ET</td>
<td>$ETRAP special variable</td>
</tr>
<tr>
<td>F</td>
<td>FOR command</td>
</tr>
<tr>
<td>$F</td>
<td>$FIND function</td>
</tr>
<tr>
<td>$FN</td>
<td>$FNUMBER function</td>
</tr>
<tr>
<td>G</td>
<td>GOTO command</td>
</tr>
<tr>
<td>$G</td>
<td>$GET function</td>
</tr>
<tr>
<td>^$G</td>
<td>^$GLOBAL structured system variable</td>
</tr>
<tr>
<td>H</td>
<td>HALT command (no arguments) or HANG command (with argument)</td>
</tr>
<tr>
<td>$H</td>
<td>$HOROLOG special variable</td>
</tr>
<tr>
<td>I</td>
<td>IF command</td>
</tr>
<tr>
<td>$I</td>
<td>$INCREMENT function (with arguments) or $IO special variable (no arguments).</td>
</tr>
<tr>
<td>$IN</td>
<td>$INUMBER function</td>
</tr>
<tr>
<td>J</td>
<td>JOB command</td>
</tr>
<tr>
<td>$J</td>
<td>$JUSTIFY function (with arguments) or $JOB special variable (no arguments).</td>
</tr>
<tr>
<td>^$J</td>
<td>^$JOB structured system variable</td>
</tr>
<tr>
<td>K</td>
<td>KILL command</td>
</tr>
<tr>
<td>$K</td>
<td>$KEY special variable</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Name</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>L</td>
<td>LOCK command</td>
</tr>
<tr>
<td>$L</td>
<td>$LENGTH function</td>
</tr>
<tr>
<td>^$L</td>
<td>^$LOCK structured system variable</td>
</tr>
<tr>
<td>$LB</td>
<td>$LISTBUILD function</td>
</tr>
<tr>
<td>$LD</td>
<td>$LISTDATA function</td>
</tr>
<tr>
<td>$LF</td>
<td>$LISTFIND function</td>
</tr>
<tr>
<td>$LFS</td>
<td>$LISTFROMSTRING function</td>
</tr>
<tr>
<td>$LG</td>
<td>$LISTGET function</td>
</tr>
<tr>
<td>$LI</td>
<td>$LIST function</td>
</tr>
<tr>
<td>$LL</td>
<td>$LISTLENGTH function</td>
</tr>
<tr>
<td>$LS</td>
<td>$LISTSAME function</td>
</tr>
<tr>
<td>$LTS</td>
<td>$LISTTOSTRING function</td>
</tr>
<tr>
<td>$LU</td>
<td>$LISTUPDATE function</td>
</tr>
<tr>
<td>$LV</td>
<td>$LISTVALID function</td>
</tr>
<tr>
<td>M</td>
<td>MERGE command</td>
</tr>
<tr>
<td>MVC</td>
<td>MVCRT command</td>
</tr>
<tr>
<td>MVP</td>
<td>MVPRINT command</td>
</tr>
<tr>
<td>N</td>
<td>NEW command</td>
</tr>
<tr>
<td>$N</td>
<td>$NEXT function</td>
</tr>
<tr>
<td>$NA</td>
<td>$NAME function</td>
</tr>
<tr>
<td>$NC</td>
<td>$NCONVERT function</td>
</tr>
<tr>
<td>$NUM</td>
<td>$NUMBER function</td>
</tr>
<tr>
<td>O</td>
<td>OPEN command</td>
</tr>
<tr>
<td>$O</td>
<td>$ORDER function</td>
</tr>
<tr>
<td>P</td>
<td>PRINT command</td>
</tr>
<tr>
<td>$P</td>
<td>$PIECE function (with arguments) or $PRINCIPAL special variable (no arguments).</td>
</tr>
<tr>
<td>Q</td>
<td>QUIT command</td>
</tr>
<tr>
<td>$Q</td>
<td>$QUERY function (with arguments) or $QUIT special variable (no arguments).</td>
</tr>
<tr>
<td>$QL</td>
<td>$QLENGTH function</td>
</tr>
<tr>
<td>$QS</td>
<td>$QSUBSCRIPT function</td>
</tr>
<tr>
<td>R</td>
<td>READ command</td>
</tr>
<tr>
<td>$R</td>
<td>$RANDOM function</td>
</tr>
<tr>
<td>^$R</td>
<td>^$ROUTINE structured system variable</td>
</tr>
<tr>
<td>$RE</td>
<td>$REVERSE function</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Name</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>RET</td>
<td>RETURN command</td>
</tr>
<tr>
<td>S</td>
<td>SET command</td>
</tr>
<tr>
<td>$S</td>
<td>$SELECT function (with arguments) or $STORAGE special variable (no arguments).</td>
</tr>
<tr>
<td>$SC</td>
<td>$SCONVERT function</td>
</tr>
<tr>
<td>$SEQ</td>
<td>$SEQUENCE function</td>
</tr>
<tr>
<td>$ST</td>
<td>$STACK function (with arguments) or $STACK special variable (no arguments).</td>
</tr>
<tr>
<td>$SY</td>
<td>$SYSTEM special variable</td>
</tr>
<tr>
<td>$T</td>
<td>$TEXT function (with arguments) or $TEST special variable (no arguments).</td>
</tr>
<tr>
<td>TC</td>
<td>TCOMMIT command</td>
</tr>
<tr>
<td>$TL</td>
<td>$TLEVEL special variable</td>
</tr>
<tr>
<td>$TR</td>
<td>$TRANSLATE function</td>
</tr>
<tr>
<td>TRO</td>
<td>TROLLBACK command</td>
</tr>
<tr>
<td>TS</td>
<td>TSTART command</td>
</tr>
<tr>
<td>U</td>
<td>USE command</td>
</tr>
<tr>
<td>V</td>
<td>VIEW command</td>
</tr>
<tr>
<td>$V</td>
<td>$VIEW function</td>
</tr>
<tr>
<td>W</td>
<td>WRITE command</td>
</tr>
<tr>
<td>$WA</td>
<td>$WASCII function</td>
</tr>
<tr>
<td>$WC</td>
<td>$WCHAR function</td>
</tr>
<tr>
<td>$WE</td>
<td>$WEXTRACT function</td>
</tr>
<tr>
<td>$WF</td>
<td>$WFIND function</td>
</tr>
<tr>
<td>$WL</td>
<td>$WLENGTH function</td>
</tr>
<tr>
<td>$WRE</td>
<td>$WREVERSE function</td>
</tr>
<tr>
<td>X</td>
<td>XECUTE command</td>
</tr>
<tr>
<td>$X</td>
<td>$X special variable (no abbreviation).</td>
</tr>
<tr>
<td>$Y</td>
<td>$Y special variable (no abbreviation).</td>
</tr>
<tr>
<td>$ZA</td>
<td>$ZA special variable (no abbreviation).</td>
</tr>
<tr>
<td>ZB</td>
<td>ZBREAK command</td>
</tr>
<tr>
<td>$ZB</td>
<td>$ZBOOLEAN function (with arguments) or $ZB special variable (no arguments, no abbreviation).</td>
</tr>
<tr>
<td>$ZC</td>
<td>$ZCYC function (with argument) or $ZCHILD special variable (no arguments).</td>
</tr>
<tr>
<td>$ZCVT</td>
<td>$ZCONVERT function</td>
</tr>
<tr>
<td>$ZD</td>
<td>$ZDATE function</td>
</tr>
<tr>
<td>$ZDA</td>
<td>$ZDASCII function</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Name</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>$ZDCHAR</td>
<td>$ZDC</td>
</tr>
<tr>
<td>$ZDATEH</td>
<td>$ZDH</td>
</tr>
<tr>
<td>$ZDATETIME</td>
<td>$ZDT</td>
</tr>
<tr>
<td>$ZDATETIMEH</td>
<td>$ZDTH</td>
</tr>
<tr>
<td>$ZERROR</td>
<td>$ZE</td>
</tr>
<tr>
<td>$ZF</td>
<td>$ZF functions (no abbreviation). See also $ZF(-100), $ZF(-3), $ZF(-4), $ZF(-5), and $ZF(-6) functions.</td>
</tr>
<tr>
<td>$ZHEX</td>
<td>$ZH</td>
</tr>
<tr>
<td>$ZINSERT</td>
<td>$ZI</td>
</tr>
<tr>
<td>$ZIO</td>
<td>$ZI</td>
</tr>
<tr>
<td>$ZJOB</td>
<td>$ZJ</td>
</tr>
<tr>
<td>$ZKILL</td>
<td>$ZK</td>
</tr>
<tr>
<td>$ZLOAD</td>
<td>$ZL</td>
</tr>
<tr>
<td>$ZASCII</td>
<td>$ZLA</td>
</tr>
<tr>
<td>$ZCHAR</td>
<td>$ZLC</td>
</tr>
<tr>
<td>$ZMODE</td>
<td>$ZM</td>
</tr>
<tr>
<td>$ZNSPACE</td>
<td>$ZN</td>
</tr>
<tr>
<td>$ZNAME</td>
<td>$ZN</td>
</tr>
<tr>
<td>$ZORDER</td>
<td>$ZO</td>
</tr>
<tr>
<td>$ZPRINT</td>
<td>$ZP</td>
</tr>
<tr>
<td>$ZPARENT</td>
<td>$ZP</td>
</tr>
<tr>
<td>$ZPREVIOUS</td>
<td>$ZP</td>
</tr>
<tr>
<td>$ZQUIT</td>
<td>$ZQ</td>
</tr>
<tr>
<td>$ZASCII</td>
<td>$ZQA</td>
</tr>
<tr>
<td>$ZCHAR</td>
<td>$ZQC</td>
</tr>
<tr>
<td>$ZREMOVE</td>
<td>$ZR</td>
</tr>
<tr>
<td>$ZREFERENCE</td>
<td>$ZR</td>
</tr>
<tr>
<td>$ZSAVE</td>
<td>$ZS</td>
</tr>
<tr>
<td>$ZSTORAGE</td>
<td>$ZS</td>
</tr>
<tr>
<td>$ZSEARCH</td>
<td>$ZSE</td>
</tr>
<tr>
<td>$ZTIME</td>
<td>$ZT</td>
</tr>
<tr>
<td>$ZTIMEH</td>
<td>$ZTH</td>
</tr>
<tr>
<td>$ZTIMESTAMP</td>
<td>$ZTS</td>
</tr>
<tr>
<td>$ZTIMEZONE</td>
<td>$ZTZ</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Name</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>$ZV</td>
<td>$ZVERSION special variable</td>
</tr>
<tr>
<td>$ZU</td>
<td>$ZUTIL functions</td>
</tr>
<tr>
<td>ZW</td>
<td>ZWRITE command</td>
</tr>
<tr>
<td>$ZWA</td>
<td>$ZWASCII function</td>
</tr>
<tr>
<td>$ZWC</td>
<td>$ZWCHAR function</td>
</tr>
</tbody>
</table>
ObjectScript Commands

This document provides detailed descriptions of the commands supported by ObjectScript. In this manual, ObjectScript commands are divided into three groups:

- General Commands.
- Routine and Debugging Commands.
- Legacy Commands.

Within each group, the commands are presented in alphabetical order.

For more information on ObjectScript commands generally, see the Commands chapter of Using Caché ObjectScript.

You can abbreviate most commands to the first letter of the command name, or, in the case of commands that begin with the letter Z, to the first two letters of the command name. In the Synopsis for each command, the full name syntax is first presented, and below it is shown the abbreviated name (if one exists).

The Synopsis for each command contains only literal syntactical punctuation. The Synopsis does not include punctuation for format conventions, such as what elements of the syntax are optional. This information is provided in the table of arguments immediately following the Synopsis.

The one exception is the ellipsis (...). An ellipsis following a comma indicates that the argument (or argument group) preceding the comma can be repeated multiple times as a comma-separated list. An ellipsis within curly braces { . . . } indicates that a block of code containing one or more commands can be specified within the curly braces. The curly braces are literal characters that must be specified in the code.

Most commands take one or more arguments. Arguments are expressions (for example, a function and its parameters, a variable, an operator and its operands, an object property, or an object method) that define or control the action of the command. Multiple arguments used with a command are generally referred to as an argument list. Some commands have arguments that themselves take argument parameters. For example, each argument of the DO command can take a parameter list. This is indicated in the syntax.

Some commands are argumentless, and can be invoked without any arguments. Some commands never take arguments; other commands take arguments only in certain circumstances. Such commands change their meaning depending on whether they are argumentless, or specify an argument list.

Most commands can take an optional postconditional expression, which specifies a condition that dictates whether or not the command should be executed. A postconditional expression is appended to the command name by a colon (:). No spaces or line breaks are permitted between a command name and its postconditional expression. While a postconditional expression is not, strictly speaking, a command argument, they are here presented with the arguments. An argumentless command can take a postconditional expression.

Most ObjectScript commands are the same on all hardware platforms. Any platform-specific features of a command are marked with the type of platform that supports it; for example, Windows or UNIX®. Any command not marked with a platform limitation is supported by all platforms.
**BREAK**

Interrupts execution at a breakpoint. Enables or disables user interrupts.

```
BREAK:pc "extend"
B:pc "extend"

BREAK:pc flag
B:pc flag
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>Optional — A postconditional expression.</td>
</tr>
<tr>
<td>extend</td>
<td>Optional — A letter code indicating the kind of breakpoints to enable or disable, specified as a quoted string. Valid values are listed in BREAK Extended Arguments. Cannot be used with the flag argument.</td>
</tr>
<tr>
<td>flag</td>
<td>Optional — An integer flag that specifies interrupt behavior. The flag value can be quoted or unquoted. Valid values are: 0 and 4 which disable CTRL-C interrupts, and 1 and 5 which enable CTRL-C interrupts. The default is determined by context (see BREAK flag for details). Cannot be used with the extend argument.</td>
</tr>
</tbody>
</table>

**Description**

The **BREAK** command has three forms:

- **BREAK without an argument** breaks code execution at the current location.
- **BREAK extend** breaks code execution at regular breakpoint intervals.
- **BREAK flag** enables or disables CTRL-C interrupts.

**Note:** Older versions of ObjectScript accepted only the abbreviation (B) for the BREAK command. Current versions accept either form.

**Required Permission**

To use **BREAK** statements when running code, the user must be assigned to a role (such as %Developer or %Manager) that provides the %Development resource with U (use) permission. A user is assigned to a role either through the SQL GRANT statement, or by using the Management Portal System Administration, Security, Users option. Select a user name to edit its definition, then select the Roles tab to assign that user to a role.

**Arguments**

**pc**

An optional postconditional expression. Caché executes the **BREAK** command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

**extend**

**BREAK extend** supports letter string codes to specify breakpoint behavior. Quotes are required. See BREAK Extended Arguments for a table of these extend codes.
BREAK

flag

BREAK flag supports four different ways to handle CTRL-C interrupts:

• BREAK 0: dismisses any pending, but not yet signaled, CTRL-C trap. Disables future signaling of CTRL-C.
• BREAK 1: dismisses any pending CTRL-C trap. Enables future signaling of CTRL-C.
  This means that typing CTRL-C following execution of the BREAK 1 causes a CTRL-C signal.
• BREAK 4: does not dismiss any pending CTRL-C trap. Disables future signaling of CTRL-C.
  This means that a pending CTRL-C trap is signaled when a future BREAK 1 or BREAK 5 command enables CTRL-C.
• BREAK 5: does not dismiss any pending CTRL-C trap. Enables future signaling of CTRL-C.
  This means that a pending CTRL-C trap should be signaled by an ObjectScript command shortly following BREAK 5.

Most, but not all ObjectScript commands poll for CTRL-C. See BREAK flag to Enable or Disable Interrupts for further details and examples.

Argumentless BREAK

Argumentless BREAK interrupts code execution when encountered. You can use argumentless BREAK in program source code, with or without a postconditional, to interrupt program execution at that point and return control to the Terminal prompt. Argumentless BREAK is used for debugging purposes.

If you include an argumentless BREAK within a routine, it sets a breakpoint, which interrupts routine execution and returns the process to the Terminal prompt. By imbedding breakpoints in your code, you can establish specific contexts for debugging. Each time execution reaches a BREAK, Caché suspends the routine and returns you to the Terminal prompt. You can then use other ObjectScript commands to perform debugging activities. For example, you might use the WRITE command to examine the values of variables at the current stopping point and the SET command to supply new values for these or other variables. You can also invoke the Routine Line Editor (XECUTE ^%), which provides basic editing capabilities for modifying the routine. After you suspend routine execution with a BREAK, you can resume normal execution by using an argumentless GOTO. Alternatively, you can resume execution at a different location by specifying this location as the GOTO command argument.

Note: InterSystems recommends that you use the ZBREAK command to invoke the Caché Debugger, rather than using the BREAK command in code. The Debugger provides much more extensive debugging capabilities.

You can configure argumentless BREAK behavior for the current process using the BreakMode() method of the %SYSTEM.Process class. You can configure argumentless BREAK behavior system-wide by setting the BreakMode property in the Config.Miscellaneous class.

Like all argumentless commands, you must insert at least two blank spaces between an argumentless BREAK and a command following it on the same line.

Using Argumentless BREAK with a Condition

You may find it useful to specify a condition on an argumentless BREAK command in code so that you can rerun the same code simply by setting a variable rather than having to change the routine. For example, you may have the following line in a routine:

BREAK: $DATA(debug)

You can then set the variable debug to suspend the routine and return the job to the Terminal prompt or clear the variable debug to continue running the routine.
BREAK Extended Arguments to Set Regular Breakpoints

You do not have to place argumentless BREAK commands at every location where you want to suspend your routine. BREAK has a series of "extended" arguments (extend) that can periodically suspend a routine as if you scattered argumentless BREAKs throughout the code. BREAK command extended arguments are listed in the following table.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;S&quot;</td>
<td>Use BREAK &quot;S&quot; (Single Step) to step through your code a single command (generated token) at a time. Not all ObjectScript commands generate a token; some generate multiple tokens and thus are parsed as multiple steps (see below). Caché stops breaking on commands invoked by a DO command or an XECUTE command, or within a FOR loop or a user-defined function, and resumes with the next token when the command or loop is done.</td>
</tr>
<tr>
<td>&quot;S+&quot;</td>
<td>BREAK &quot;S+&quot; acts like BREAK &quot;S&quot;, except that Caché includes breaking on commands invoked by a DO command or an XECUTE command, or within a FOR loop or a user-defined function.</td>
</tr>
<tr>
<td>&quot;S-&quot;</td>
<td>Use BREAK &quot;S-&quot; to disable break stepping (&quot;S&quot; or &quot;L&quot;) at the current level and enable single stepping at the previous level. Acts like BREAK &quot;C-&quot; at the current level and BREAK &quot;S&quot; at the previous level.</td>
</tr>
<tr>
<td>&quot;L&quot;</td>
<td>Use BREAK &quot;L&quot; (Line Stepping) to step through your code a single routine line at a time, breaking at the beginning of every line. Lines that do not generate tokens are ignored (see below). Caché stops breaking on commands invoked by a DO command or an XECUTE command, or within a FOR loop or a user-defined function, and resumes with the next line when the command or loop is done.</td>
</tr>
<tr>
<td>&quot;L+&quot;</td>
<td>BREAK &quot;L+&quot; acts like BREAK &quot;L&quot;, except that Caché also continues to break at the beginning of every routine line on commands invoked by a DO command or an XECUTE command, or within a FOR loop or a user-defined function.</td>
</tr>
<tr>
<td>&quot;L-&quot;</td>
<td>Use BREAK &quot;L-&quot; to disable break stepping (&quot;S&quot; or &quot;L&quot;) at the current level and enable line stepping at the previous level. Acts like BREAK &quot;C-&quot; at the current level and BREAK &quot;L&quot; at the previous level.</td>
</tr>
<tr>
<td>&quot;C&quot;</td>
<td>Use BREAK &quot;C&quot; (Clear Break) to stop all break stepping (&quot;L.&quot; and &quot;S&quot;) at the current level. Breaking resumes at a previous routine level after the job executes a QUIT if a BREAK state is in effect at that previous level.</td>
</tr>
<tr>
<td>&quot;C-&quot;</td>
<td>Use BREAK &quot;C-&quot; to stop all break stepping (&quot;L&quot; and &quot;S&quot;) at the current level and all previous levels. This allows stepping to be removed at all levels without affecting other debugging features.</td>
</tr>
<tr>
<td>&quot;OFF&quot;</td>
<td>BREAK &quot;OFF&quot; removes all debugging that has been established for the process. It removes all breakpoints and watchpoints, and turns off stepping at all program stack levels. It also removes the association with the debug and trace devices, but does not close them.</td>
</tr>
</tbody>
</table>

BREAK “S” and BREAK “L” break on statements that generate tokens. Not all ObjectScript commands or lines generate a token. For example, BREAK “S” and BREAK “L” both ignore label lines, comments, and TRY statements. BREAK “S” breaks at a CATCH statement (if the CATCH block is entered); BREAK “L.” does not.

One difference between BREAK “S” and BREAK “L.” is that many command lines generate more than one token and thus consist of more than one step. This is not always obvious. For example, the following are all one line (and one ObjectScript command), but BREAK “S” parses each as two steps: SET x=1, y=2, KILL x,y, WRITE “hello”!. IF x=1, y=2.
To resume code execution after a breakpoint, issue a GOTO command (abbreviated as G) at the Terminal prompt. See Debugging in Using Caché ObjectScript for more information.

Issuing a BREAK "OFF" command is equivalent to issuing the following series of commands:

```objectscript
ZBREAK /CLEAR
ZBREAK /TRACE:OFF
ZBREAK /DEBUG:""
ZBREAK /ERRORTRAP:ON
BREAK "C-"
```

**BREAK flag to Enable or Disable Interrupts**

Use BREAK flag to control whether user interrupts, such as CTRL-C, are enabled or disabled. The practical difference between these disable/enable options is as follows:

- **BREAK 0** and **BREAK 1** can be used to create a code block where a CTRL-C signal cannot interrupt a critical sequence of commands. However, a loop on such a block may be difficult for an interactive user to interrupt using CTRL-C. This is because there is a slight delay between detecting a CTRL-C trap and polling a CTRL-C signal. This delay may permit the next BREAK command loop to dismiss the CTRL-C user interrupt.

A program block containing **BREAK 4** and **BREAK 5** can be used to create code where a CTRL-C signal cannot interrupt a critical sequence of commands without affecting the ability of an interactive user to interrupt a loop operation on this block using CTRL-C.

The default flag behavior of BREAK is dependent upon the login mode, as follows:

- If you log in at the Terminal prompt, the default is **BREAK 1**. Interrupts, such as CTRL-C, are always enabled. The B (/break) protocol specified in an OPEN or USE command has no effect.
- If you are in a routine, the default is **BREAK 0**. Interrupts, such as CTRL-C, are enabled or disabled by the B (/break) protocol specified in an OPEN or USE command.

For further details on OPEN and USE mode protocols, refer to Terminal I/O in the Caché I/O Device Guide.

**BREAK flag Examples**

The following example uses $ZJOB to determine if interrupts are enabled or disabled:

```objectscript
BREAK 0
DO InterruptStatus
BREAK 1
DO InterruptStatus
WRITE "all done"
InterruptStatus()
IF $ZJOB\4#2=1 {WRITE "Interrupts enabled",!}
ELSE {WRITE "Interrupts disabled",!}
```

The following example uses a READ in a FOR loop for user input of a series of numbers. It sets **BREAK 0** to disable user interrupts during the READ operation. However, if the user inputs a value that is not a number, **BREAK 1** enables user interrupts so that the user can either reject or accept the value they just input:

```objectscript
SET y="^"
InputLoop
TRY {
  FOR {  
    BREAK 0
    READ "input a number ",x
    IF x="" { WRITE !,"all done" QUIT }
    ELSEIF 0=$ISVALIDNUM(x) {
      BREAK 1
      WRITE !,x," is not a number",!
      WRITE "you have four seconds to press CTRL-C",!
      WRITE "or accept this input value",!
      HANG 4 }
    ELSE {  }
    SET y=y_x_"^"
  }
  WRITE !,"the number list is ",y,!
}
```
```ObjectScript
} CATCH { WRITE "Rejecting bad input",!
    DO InputLoop
}
```

### See Also

- **ZBREAK** command
- **GOTO** command
- **OPEN** command
- **USE** command
- **Debugging** in *Using Caché ObjectScript*
- **Terminal I/O** in *Caché I/O Device Guide*
CATCH

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

```
CATCH exceptionvar
{
  . . .
}
```

**Argument**

<table>
<thead>
<tr>
<th>exceptionvar</th>
<th>Optional — An exception variable. Specified as a local variable, with or without subscripts, that receives a reference to a Caché Object (an OREF). This argument can, optionally, be enclosed with parentheses.</th>
</tr>
</thead>
</table>

**Description**

The `CATCH` command defines an exception handler, a block of code to execute when an exception occurs in a `TRY` block of code. The `CATCH` command is followed by a block of code statements, enclosed in curly braces.

If you specify a `TRY` block, a `CATCH` block is required; every `TRY` block must have a corresponding `CATCH` block. Only one `CATCH` block is permitted for each `TRY` block. The `CATCH` block must immediately follow its `TRY` block. No lines of executable code are permitted between a `TRY` block and its `CATCH` block. No label is permitted between a `TRY` block and its `CATCH` block, or on the same line as the `CATCH` command. You can, however, include comments between a `TRY` block and its `CATCH` block.

A `CATCH` block is entered when an exception occurs. If no exception occurs, the `CATCH` block should not be executed. You should *never* use a `GOTO` statement to enter a `CATCH` block.

You can exit a `CATCH` block using `QUIT` or `RETURN`. `QUIT` exits the current block structure and continues execution with the next command outside of that block structure. For example, if you are within a nested `CATCH` block, issuing a `QUIT` exits that `CATCH` block to the enclosing block structure. You cannot use an argumented `QUIT` to exit a `CATCH` block; attempted to do so results in a compile error. To exit a routine completely from within a `CATCH` block, issue a `RETURN` statement.

The `CATCH` command has two forms:

- Without an argument
- With an argument

The argumented form is preferred.

**CATCH Exception Handling**

`CATCH exceptionvar` receives an object instance reference (OREF) from the `TRY` block, either explicitly passed by the `THROW` command, or implicitly from the system runtime environment in the event of a system error. For information on OREFs, see “OREF Basics” in Using Caché Objects.

This object instance reference provides properties that contain information about the exception.

An exception can pass four exception properties to `CATCH`. These are, in order: Name, Code, Location, and Data. A thrown exception cannot pass a Location parameter. You can use the `%IsA()` instance method to determine what type of exception passed in these properties.

In the following example, the `TRY` block can generate a system exception (undefined local variable), throw an SQL exception, throw a `%Status` exception, or throw a general exception. This general-purpose `CATCH` exception handler determines which type of exception occurred and displays the appropriate properties. It displays all four properties for a
system exception (the Data property is the empty string for some types of system errors). It displays two properties for an SQL exception (Code and Data). It supplies two properties to $SYSTEM.Status.Error() to generate an error message string for a %Status exception. It displays three properties for a general ObjectScript exception (Name, Code, and Data). It uses the $ZCVT function to format a Name value containing angle brackets for browser display:

```
TRY {
SET x=$RANDOM(4)
IF x=0 { KILL undefvar
    WRITE undefvar }
ELSEIF x=1 {
    SET oref=##class(%Exception.SQL).%New("-999","SQL error message")
    THROW oref }
ELSEIF x=2 {
    SET oref=##class(%Exception.StatusException).%New("5002","LISTBUILD("My Status Error")")
    THROW oref }
ELSE {
    SET oref=##class(%Exception.General).%New("<MY BAD>"","999","General error message")
    THROW oref
}
WRITE "this should not display",!
CATCH exp { WRITE "In the CATCH block",!
    IF 1=exp.%IsA("%Exception.SystemException") {
        WRITE "System exception",!
        WRITE "Name: ",$ZCVT(exp.Name,"O","HTML"),!
        WRITE "Location: ",exp.Location,!
        WRITE "Code: 
    }
    ELSEIF 1=exp.%IsA("%Exception.SQL") {
        WRITE "SQL exception",!
        WRITE "SQLCODE: 
    }
    ELSEIF 1=exp.%IsA("%Exception.StatusException") {
        WRITE "%Status exception",!
        RETURN
    }
    ELSEIF 1=exp.%IsA("%Exception.General") {
        WRITE "General ObjectScript exception",!
        WRITE "Name: ",$ZCVT(exp.Name,"O","HTML"),!
        WRITE "Code: 
    }
    ELSE { WRITE "Some other type of exception",! RETURN }
    WRITE exp.Code,!
    WRITE "Data: ",exp.Data,!
    RETURN }
```

**Nested TRY/CATCH Blocks**

Only one **CATCH** block is permitted for each **TRY** block. However, it is possible to nest paired **TRY/CATCH** blocks.

You can nest an inner **TRY/CATCH** pair within an outer **CATCH** block, such as the following:

```
TRY {
    /* TRY code */
}
CATCH exvar1 {
    /* CATCH code */
    TRY {
        /* nested TRY code */
    }
    CATCH exvar2 {
        /* nested CATCH code */
    }
}
```

You can nest an inner **TRY/CATCH** pair within an outer **TRY** block, such as the following:
TRY { /* TRY code */
    TRY { /* nested TRY code */
    CATCH exvar2 {
        /* nested CATCH code */
    }
    }
    CATCH exvar1 { /* CATCH code */
    }
}

**Execution Stack**

The %Exception object contains the execution stack at the time the object was created. You can access this execution stack using the `StackAsArray()` method. The following example shows this execution stack:

```ObjectScript
TRY {
    WRITE "In the TRY block",!
    WRITE 7/0
}
CATCH exobj {
    WRITE "In the CATCH block",!
    WRITE $ZCVT($ZERROR,"O","HTML"),!
    TRY {
        WRITE "In the nested TRY block",!
        KILL fred
        WRITE fred
    }
    CATCH exobj2 {
        WRITE "In the nested CATCH block",!
        WRITE $ZCVT($ZERROR,"O","HTML"),!!
        WRITE "The Execution Stack",!
        DO exobj2.StackAsArray(.stk)
        ZWRITE stk
    }
}
```

### CATCH and $ZTRAP, $ETRAP

You cannot set `$ZTRAP` or `$ETRAP` within a TRY block. However, you can set `$ZTRAP` or `$ETRAP` within a CATCH block. You can also set `$ZTRAP` or `$ETRAP` before entering the TRY block.

If an exception occurs within the CATCH block, the specified `$ZTRAP` or `$ETRAP` exception handler is taken.

### TRY / CATCH Loop

A loop where a TRY block invokes a CATCH block that loops back to the TRY block does not loop infinitely. It eventually issues a `<FRAMESTACK>` error.

### Disabling CATCH

Issuing a `ZBREAK /ERRORTRAP:OFF` command disables CATCH exception handling.

### Argument

**exceptionvar**

A local variable, used to receive the exception object reference from the THROW command or from the system runtime environment in the event of a system error. When a system error occurs, `exceptionvar` receives a reference to an object of type `%Exception.SystemException` When a user-specified error occurs, `exceptionvar` receives a reference to an object of type `%Exception.General`, `%Exception.StatusException`, or `%Exception.SQL`. For further details, refer to the `%Exception.AbstractException` abstract class in the [InterSystems Class Reference](#).

The `exceptionvar` argument can optionally be enclosed with parentheses, thus: `CATCH(var) { code block }`. This parentheses syntax is provided for compatibility, and has no effect on functionality.
Examples: System Exceptions

The following example shows an argumentless CATCH invoked by a divide-by-zero runtime error. It displays the $ZERROR and $ECODE error values. Argumentless CATCH is not recommended because it is less reliable than passing an exceptionvar. If an error occurs in the CATCH block, $ZERROR will contain this most recent error, not the error that invoked the CATCH. In this example the QUIT command exits the CATCH block, but does not prevent “fall-through” to the next line outside the block structure:

```objectscript
TRY {
  WRITE !,"TRY block about to divide by zero",!!
  SET a=7/0
  WRITE !,"this should not display"
}
CATCH {
  WRITE "CATCH block exception handler",!!
  WRITE "$ZERROR is: ",$ZERROR,!
  WRITE "$ECODE is :",$ECODE,!
  QUIT
  WRITE !,"this should not display"
}
WRITE !,"this is where the code falls through"
```

The following example shows a CATCH invoked by a divide-by-zero runtime error and receiving an argument. This is the preferred coding practice. The myexp OREF argument receives a system-generated exception object. It displays the Name, Code, and Location properties of this exception instance. In this example the RETURN command exits the program, so no “fall-through” occurs:

```objectscript
TRY {
  WRITE !,"TRY block about to divide by zero",!!
  SET a=7/0
  WRITE !,"this should not display"
}
CATCH myexp {
  WRITE "CATCH block exception handler",!!
  WRITE "Name: ",$ZCVT(myexp.Name,"O","HTML"),!
  WRITE "Code: ",myexp.Code,!
  WRITE "Location: ",myexp.Location,!
  RETURN
}
WRITE !,"this is where the code falls through"
```

The following example shows a CATCH receiving a system exception object. The CATCH block code displays the system exception as a $ZERROR-formatted string using the AsSystemError() method of the %Exception.SystemException class. ($ZERROR is also displayed, for comparison purposes.) This CATCH block then displays the error name, error code, error data, and error location as separate properties:

```objectscript
TRY {
  WRITE !,"this global is not defined",!
  SET a="badglobal()"
  WRITE !,"this should not display"
}
CATCH myvar {
  WRITE !,"this is the exception handler",!
  WRITE "AsSystemError is: ",myvar.AsSystemError(),!
  WRITE "$ZERROR is: ",$ZERROR,!
  WRITE "Error name=",$ZCVT(myvar.Name,"O","HTML"),!
  WRITE "Error code=" ,myvar.Code,!
  WRITE "Error data=" ,myvar.Data,!
  WRITE "Error location=" ,myvar.Location,!
  RETURN
}
```

Examples: Thrown Exceptions

The following example shows a CATCH invoked by the THROW command. The myvar argument receives a user-defined exception object with four properties. Note that in this example the THROW does not supply a value for the omitted Location property of the %Exception.General class:
TRY {  
  SET total=1234  
  WRITE !,"Throw an exception!"  
  THROW ##class(%Exception.General).%New("Example Error",999,"MyThrow")  
  WRITE !,"this should not display"  
}  
CATCH myvar {  
  WRITE !,"this is the exception handler"  
  WRITE !,"Error data="myvar.Data  
  WRITE !,"Error code="myvar.Code  
  WRITE !,"Error name="myvar.Name  
  WRITE !,"Error location="myvar.Location,!  
  RETURN  
}
The following two examples generate birth dates in the TRY block. If they generate a birth date that is in the future, they use THROW to issue a general exception, passing the user-defined exception to the CATCH block. (You may have to run these examples more than once to generate a date that throws an exception.)
The first of these examples does not specify a CATCH exceptionvar. It uses the OREF name defined in the TRY block to specify the exception properties:

TRY {  
  WRITE !,"In the TRY block",!  
  SET badDOB=##class(%Exception.General).%New("BadDOB","999","Birth date is in the future")  
  FOR x=1:1:20 {  
    SET rndDOB = $RANDOM(7)_$RANDOM(10000)  
    IF rndDOB > $HOROLOG { THROW badDOB }  
    ELSE { WRITE !,"Birthdate ",$ZDATE(rndDOB,1,,4)," is valid",!  
  }  
}  
CATCH {  
  WRITE !,"In the CATCH block"  
  WRITE !,"Birthdate ",$ZDATE(rndDOB,1,,4)," is invalid"  
  WRITE !,"Error code="badDOB.Code  
  WRITE !,"Error name="badDOB.Name  
  WRITE !,"Error data="badDOB.Data  
  RETURN  
}
The second of these examples specifies a CATCH exceptionvar. It uses this renamed OREF to specify the exception properties. This is the preferred usage:

TRY {  
  WRITE !,"In the TRY block",!  
  SET badDOB=##class(%Exception.General).%New("BadDOB","999","Birth date is in the future")  
  FOR x=1:1:20 {  
    SET rndDOB = $RANDOM(7)_$RANDOM(10000)  
    IF rndDOB > $HOROLOG { THROW badDOB }  
    ELSE { WRITE !,"Birthdate ",$ZDATE(rndDOB,1,,4)," is valid",!  
  }  
}  
CATCH err {  
  WRITE !,"In the CATCH block"  
  WRITE !,"Birthdate ",$ZDATE(rndDOB,1,,4)," is invalid"  
  WRITE !,"Error code="err.Code  
  WRITE !,"Error name="err.Name  
  WRITE !,"Error data="err.Data  
  RETURN  
}

Example: Nested TRY/CATCH
The following example shows a CATCH invoked by a divide-by-zero runtime error. The CATCH block contains an inner TRY block paired with an inner CATCH block. This inner CATCH block is invoked by a thrown exception. For the purposes of demonstration, this THROW is invoked randomly in this program. In a real-world program, the inner CATCH block would be invoked by an exception test, such as a mismatch between AsSystemError() (the caught error) and $ZERROR (the most-recent error):

TRY {  
  WRITE !,"Outer TRY block",!!  
  SET a=7/0  
  WRITE !,"this should not display"  
}  
CATCH myexp {  
  WRITE !,"Outer CATCH block",!  
  WRITE "Name: ",$ZCVT(myexp.Name,"O","HTML"),!  
}
WRITE "Code: ",myexp.Code,
WRITE "Location: ",myexp.Location,
SET rndm=$RANDOM(2)
IF rndm=1 {RETURN }
TRY {
WRITE !,"Inner TRY block",!
SET oref=##class(%Exception.General).%New("<MY BAD>","999","General error message")
THROW oref
RETURN }
} CATCH myexp2 {
WRITE !,"Inner CATCH block",!
IF 1=myexp2.%IsA("%Exception.General") {
WRITE "General ObjectScript exception",!
WRITE "Name: ",%ZCVT(myexp2.Name,"O","HTML"),!
WRITE "Code: ",myexp2.Code,!
}
ELSE { WRITE "Some other type of exception",!
QUIT
}
WRITE !,"back to Outer CATCH block",!
RETURN
}

See Also

- THROW command
- TRY command
- ZBREAK command
- Error Processing in Using Caché ObjectScript
CLOSE

Closes a file or a device.

CLOSE:pc closearg,...
C:pc closearg,...

where closearg is:

device:parameters

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>Optional — A postconditional expression.</td>
</tr>
<tr>
<td>device</td>
<td>The device to be closed.</td>
</tr>
<tr>
<td>parameters</td>
<td>Optional — One or more parameters used to set characteristics of the device. A single parameter may be specified as a quoted string: CLOSE device:&quot;D&quot;. Multiple parameters must be specified enclosed by parentheses and separated by colons.</td>
</tr>
</tbody>
</table>

Description

CLOSE device releases ownership of the specified device, optionally sets a parameter, and returns it to the pool of available devices.

If the process does not own the specified device, or if the specified device is not open or does not exist, Caché ignores CLOSE and returns without issuing an error.

Arguments

pc

An optional postconditional expression. Caché executes the CLOSE command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

device

The device to be closed. A device can be a physical device, a TCP connection, or a sequential file. Specify the same device ID (mnemonic or number) as specified on the corresponding OPEN command. For more information on specifying device IDs, refer to the OPEN command.

You can specify a single device ID or as a comma-separated list of device IDs. CLOSE closes all of the listed devices that the process currently has open. It ignores any listed devices that do not exist or are not currently open by this process.

The device ID of the current device is contained in the $IO special variable.

parameters

A parameter or a colon-separated list of parameters used when closing the specified device. Parameter codes are not case-sensitive. Multiple parameters must be enclosed with parentheses and separated by colons.

The available parameter values are as follows:
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;D&quot;</td>
<td>Closes and deletes a sequential file. Can also be specified as /DEL, /DEL=1, /DELETE, or /DELETE=1.</td>
</tr>
<tr>
<td>&quot;R&quot;:newname</td>
<td>Closes and renames a sequential file. Can also be specified as /REN=newname or /RENAME=newname.</td>
</tr>
<tr>
<td>“K”</td>
<td>Closes at the Caché level but not at the operating system level. Used only on non-Windows systems.</td>
</tr>
</tbody>
</table>

If the specified parameter is not valid, CLOSE still closes the device.

Refer to Sequential File I/O in the Caché I/O Device Guide for further information.

**Examples**

In the following UNIX® example, the CLOSE command closes device C (/dev/tty02), but only if it is not the current device. The postconditional uses the $IO special variable to check for the current device.

```ObjectScript
CloseDevC
SET C="/dev/tty02"
OPEN C
... CLOSE:$IO'=C C
```

**Notes**

**Acquiring Ownership of a Device**

A process acquires ownership of a device with the OPEN command and makes it active with the USE command. If the closed device is the active (that is, current) device, the default I/O device becomes the current device. (The default I/O device is established at login.) When a process is terminated (for example, after a HALT), all its opened devices are automatically closed and returned to the system.

If the process’s default device is closed, any subsequent output (such as error messages) to that device causes the process to hang. In this case, you must explicitly reopen the default device.

**See Also**

- OPEN command
- I/O Devices and Commands in Caché I/O Device Guide
CONTINUE

Jumps to FOR, WHILE, or DO WHILE command and re-executes test and loop.

CONTINUE:pc

Argument

| pc | Optional — A postconditional expression. |

Description

The CONTINUE command is used within the code block following a FOR, WHILE, or DO WHILE command. CONTINUE causes execution to jump back to the FOR, WHILE, or DO WHILE command. The FOR, WHILE, or DO WHILE command evaluates its test condition, and, based on that evaluation, re-executes the code block loop. Thus, the CONTINUE command has exactly the same effect on execution as reaching the closing curly brace (} ) of the code block.

CONTINUE takes no arguments (other than the postconditional). At least two blank spaces must separate it from a command following it on the same line.

A CONTINUE can cause execution to jump out of a TRY or CATCH block to return to its control flow statement.

Arguments

pc

An optional postconditional expression that can make the command conditional. Caché executes the CONTINUE command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

Examples

The following example uses a CONTINUE with a postconditional expression. It loops through and prints out all the numbers from 1 to 10, except 3:

```objectscript
Loop
  FOR i=1:1:10 {
    IF i # 2 {
      CONTINUE:i=3
    WRITE !,i," is odd"
    } ELSE { WRITE !,i," is even"
    }
WRITE !,"done with the loop"
QUIT
```

The following example shows two nested FOR loops. The CONTINUE jumps back to the FOR in the inner loop:

```objectscript
Loop
  FOR i=1:1:3 {
    WRITE !,"outer loop: i=",i
    FOR j=2:2:10 {
      WRITE !,"inner loop: j=",j
      IF j '= 8 {CONTINUE }
      ELSE { WRITE " crazy eight"
      }
    WRITE !,"back to outer loop"
  }
QUIT
```

The following example shows a CONTINUE that exits a TRY block. The CONTINUE jumps back to the FOR statement outside the TRY block.
TryLoop
FOR i=1:1:10 {
WRITE !,"Top of FOR loop"
TRY {
WRITE !,"In TRY block: i="",i
IF i=7 {
WRITE " lucky seven"
} ELSE {
CONTINUE
}
CATCH exp {
WRITE !,"CATCH block exception handler"!,
WRITE "Error code="
RETURN
}
WRITE !,"Bottom of the FOR loop"
}
QUIT

See Also

- **DO WHILE** command
- **FOR** command
- **WHILE** command
DO

Calls a routine.

DO:pc doargument,...
D:pc doargument,...

where doargument is:

entryref(param,...):pc

Arguments

<table>
<thead>
<tr>
<th>pc</th>
<th>Optional — A postconditional expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>entryref</td>
<td>The name of the routine to be called.</td>
</tr>
<tr>
<td>param</td>
<td>Optional — Parameter values to be passed to the called routine.</td>
</tr>
</tbody>
</table>

Description

The DO command and the DO WHILE command are separate and unrelated commands. This page documents the DO command. In the DO WHILE command, the DO keyword and the WHILE keyword may be separated by several lines of code; however, you can immediately identify a DO WHILE command because the DO keyword is followed by an open curly brace.

The DO command has two forms:

- **Without an Argument**

  Note: DO without an argument uses an older block structure syntax (using a period (.) prefix for each line) that has been superseded by the DO WHILE curly brace block structure. Therefore, DO without an argument is considered obsolete. It should only be used to maintain existing applications and should not be used in new code.

- **With an Argument**

  DO with an argument calls a specified object method, subroutine, function, or procedure. Caché executes the called routine, then executes the next command after the DO command. You can call the routine with or without parameter passing.

  The DO command cannot accept a return value from the called routine. If the called routine concludes with an argumented QUIT, the DO command completes successfully, but ignores the QUIT argument value.

  Each invocation of DO places a new context frame on the call stack for your process. The $STACK special variable contains the current number of context frames on the call stack. This context frame establishes a new execution level, incrementing $STACK and $ESTACK, and providing scope for NEW and SET $ZTRAP operations issued during the DO operation. Upon successful completion, DO decrements $STACK and $ESTACK and reverts NEW and SET $ZTRAP operations.

Arguments

pc

An optional postconditional expression. If the postconditional expression is appended to the DO command keyword, Caché executes the DO command if the postconditional expression is TRUE (evaluates to a nonzero numeric value). Caché does not execute the DO command if the postconditional expression is FALSE (evaluates to zero).
If the postconditional expression is appended to an argument, Caché executes the argument if the postconditional expression is TRUE (evaluates to a nonzero numeric value). If the postconditional expression is FALSE (evaluates to zero), Caché skips that argument and proceeds to evaluate the next argument (if there is one) or the next command. Note that because Caché processes expressions from left to right, any parts of the argument containing expressions (such as a parameter value or an object reference) is evaluated and can cause an error before the postconditional expression is evaluated. If DO invokes an object method with an appended postconditional, the maximum number of object method parameters is 253.

For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

**entryref**

The name of the routine (Object Method, Subroutine, Procedure, or User-supplied Function) to be called. You can specify multiple routines as a comma-separated list.

**entryref** can take any of the following forms.

<table>
<thead>
<tr>
<th>Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>label+offset</code></td>
<td>Specifies a line label within the current routine. The optional <code>+offset</code> can only be used when calling a subroutine to which no parameters are passed; it cannot be used when calling a procedure or when passing parameters to a subroutine. <code>offset</code> is a nonnegative integer that specifies the number of lines after the label at which execution of the subroutine is to start.</td>
</tr>
<tr>
<td><code>label+offset^routine</code></td>
<td>Specifies a line label within the named routine that resides on disk. Caché loads the routine from disk and begins execution at the indicated label. The <code>+offset</code> is optional.</td>
</tr>
<tr>
<td><code>^routine</code></td>
<td>The name of a routine that resides on disk. The system loads the routine from disk and begins execution at the first executable line of the routine. Must be a literal value; a variable cannot be used to specify <code>routine</code>. (Note that the <code>^</code> character is a separator character, not part of the routine name.) If the routine has been modified, Caché loads the updated version of the routine when DO invokes the routine. If the routine is not in the current namespace, you can specify the namespace that contains the routine using an extended routine reference, as follows: `^&quot;namespace&quot;</td>
</tr>
<tr>
<td><code>oref.Method()</code></td>
<td>Specifies an object method. The system accesses the object and executes the specified method, passing the arguments (if any) specified in <code>param</code>, the method's argument list. Object calls use dot syntax: <code>oref</code> (the object reference) and <code>Method()</code> are separated by a dot; blank spaces are not permitted. A method must specify its open and close parentheses, even if there are no <code>param</code> arguments.</td>
</tr>
</tbody>
</table>

The following syntactic forms are supported: `DO oref.Method()`, `DO (oref).Method()`, `DO ..Method()`, `DO #class(cname).Method()`, `DO i%prop(subs).Method()`.

You cannot specify an `offset` when calling a CACHESYS % routine. If you attempt to do so, Caché issues a <NOLINE> error.

If you specify a nonexistent label, Caché issues a <NOLINE> error. If you specify a nonexistent routine, Caché issues a <NOROUTINE> error. If you specify a nonexistent method, Caché issues a <METHOD DOES NOT EXIST> error. If you use extended reference (for example, `DO ^"%SYS"|MyProg`) and specify a nonexistent namespace, Caché issues a <NAMESPACE> error. If you use extended reference and specify a namespace for which you do not have privileges, Caché issues a <PROTECT> error, followed by the database path, such as the following: `<PROTECT>`
For further details on these errors, refer to the $ZERROR special variable.

If you specify an offset that points to the middle of a multi-line statement, the system starts execution at the beginning of the next statement.

**param**

Parameter values to be passed to the subroutine, procedure, user-supplied function or object method. You can specify a single param value, or a comma-separated list of param values. A param list is enclosed in parentheses. When no param is specified, the enclosing parentheses are required when calling a procedure or user-supplied function, optional when calling a subroutine. Parameters can be passed by value or passed by reference. The same call can mix parameters passed by value and parameters passed by reference. When passing by value, you can specify a parameter as a value constant, expression, or unsubscripted local variable name. (See Passing By Value.) When passing by reference, the parameters must reference the name of a local variable or unsubscripted array in the form .name (See Passing By Reference.)

The maximum total param values for a DO entrypoint is 382; the maximum total param values for a DO method or DO with indirection is 380. This total can include up to 254 actual parameters and 128 postconditional parameters.

You can specify a variable number of parameters using ... syntax.

**The Argumentless DO Command**

The argumentless DO command executes the block of code that immediately follows it in the same program. Each line of this block of code is indicated by a period (.) prefix. Caché then executes the next command after that block of code. Argumentless DO blocks can be nested. A postconditional expression can be appended to the argumentless DO command keyword to specify whether or not the block of code that immediately follows the DO command should be executed or skipped.

The block structures provided by the IF, FOR, DO WHILE, and WHILE commands are a preferable means to perform the same operations. The argumentless DO command continues to be supported, but its use in new coding is discouraged. Note that DO establishes a new execution level; DO WHILE and the other block structure commands do not change execution level. For further details, see Argumentless DO (legacy version).

**The DO Command with Arguments**

The DO command with entryref arguments invokes the execution of one or more blocks of code that are defined elsewhere. Each block of code to execute is specified by its entryref. The DO command can specify multiple blocks of code to execute as a comma-separated list. The execution of the DO command, and the execution of each entryref in a comma-separated list can be governed by optional postconditional expressions.

DO can invoke the execution of a subroutine (with or without parameter passing), a procedure, or a user-supplied function. Upon completion of the execution of the block of code, execution resumes at the next command after the DO command. A block of code invoked by the DO command cannot return a value to the DO command; any value returned is ignored. Thus DO can execute a user-supplied function, but cannot receive the return value of that function.

DO cannot invoke most Caché-supplied system functions. Attempting to do so results in a <SYNTAX> error. A few system functions can be invoked by DO. These include $CASE, $CLASSMETHOD, $METHOD, $ZF(-100), and the deprecated SZUTIL functions. DO cannot receive the return value of a function. Like all DO command arguments, these functions can take a postconditional parameter.

You can specify a $CASE function as a DO command argument. For an example program, refer to the $CASE function.

**The DO Command without Parameter Passing**

The DO command without parameter passing is only used with subroutines. Use of DO entryref without parameter passing (that is, without specifying the param option) takes advantage of the fact that a calling routine and its called subroutine share the same variable environment. Any variable updates made by the subroutine are automatically available to the code following the DO command.
When using **DO** without parameter passing, you must make sure that both the calling routine and the called subroutine reference the same variables.

**Note:** Procedures handle variables entirely differently. Refer to Procedures in *Using Caché ObjectScript*.

In the following example, Start (the calling routine) and Exponent (the called subroutine) share access to three variables: `num`, `powr`, and `result`. Start sets `num` and `powr` to the user-supplied values. These values are automatically available to Exponent when it is called by the **DO** command. Exponent references `num` and `powr`, and places the calculated value in `result`. When Exponent executes the **RETURN** command, control returns to the **WRITE** command immediately after the **DO**. The **WRITE** command outputs the calculated value by referencing `result`:

Start  ; Raise an integer to a specified power.  
READ !,"Integer= ",num QUIT:num="  
READ !,"Power= ",powr QUIT:powr="  
DO Exponent()  
WRITE !,"Result= ",result,!  
RETURN  
Exponent()  
SET result=num  
FOR i=1:1:powr-1 { SET result=result*num }  
RETURN

In the following example, **DO** invokes the Admit() method on the object referred to by pat. The method does not receive parameters or return a value.

```objectscript
DO pat.Admit()
```

In the following example, **DO** calls, in succession, the subroutines Init and Read1 in the current routine and the subroutine Convert in routine Test.

```objectscript
DO Init,Read1,Convert^Test
```

In the following example, **DO** uses an extended reference to call the routine fibonacci in a different namespace (the SAMPLES namespace):

```objectscript
NEW $NAMESPACE
SET $NAMESPACE="USER"
DO ^|"SAMPLES"|fibonacci
```

**DO and GOTO**

The **DO** command can be used to invoke a subroutine (with or without parameter passing), a procedure, or a user-supplied function. At the completion of the call, Caché executes the next command following the **DO** command.

The **GOTO** command can only be used to invoke a subroutine without parameter passing. At the completion of the call, Caché issues a **QUIT**, ending execution.

**DO with Parameter Passing**

When used with parameter passing, **DO** `entryref` explicitly passes one or more values to the called subroutine, procedure, user-supplied function or object method. The passed values are specified as a comma-separated list with the `param` option.

With parameter passing, you must make sure that the called subroutine is defined with a parameter list. The subroutine definition takes the form:

```objectscript
>label(param)
```

where `label` is the label name of the subroutine, procedure, user-supplied function or object method, and `param` is a comma-separated list of one or more unsubscripted local variable names. For example,
Main
  SET x=1,y=2,z=3
  WRITE !,"In Main ",x,y,z
  DO Sub1(x,y,z)
   WRITE !,"Back in Main ",x,y,z
  QUIT
Sub1(a,b,c)
  WRITE !,"In Sub1 ",a,b,c
  QUIT

The list of parameters passed by the **DO** command is known as the *actual parameter list*. The list of parameter variables defined as part of the label of the coded routine is known as the *formal parameter list*. When **DO** calls the routine, the parameters in the actual parameter list are mapped, by position, to the corresponding variables in the formal parameter list. In the above example, the value of the first actual parameter (x) is placed in the first variable (a) of the subroutine’s formal parameter list; the value of the second actual parameter (y) is placed in the second variable (b); and so on. The subroutine can then access the passed values by using the appropriate variables in its formal parameter list.

If there are more variables in the actual parameter list than there are parameters in the formal parameter list, Caché issues a <PARAMETER> error.

If there are more variables in the formal parameter list than there are parameters in the actual parameter list, the extra variables are left undefined. In the following example, the formal parameter c is left undefined:

Main
  SET x=1,y=2,z=3
  WRITE !,"In Main ",x,y,z
  DO Sub1(x,,z)
   WRITE !,"Back in Main ",x,y,z
  QUIT
Sub1(a,b,c)
  WRITE !,"In Sub1 
    IF $DATA(a) {WRITE !,"a="a}
    ELSE {WRITE !,"a is undefined"}
    IF $DATA(b) {WRITE !,"b="b}
    ELSE {WRITE !,"b is undefined"}
    IF $DATA(c) {WRITE !,"c="c}
    ELSE {WRITE !,"c is undefined"}
  QUIT

You can specify a default value for a formal parameter, to be used when no actual parameter value is specified.

You can leave any variable undefined by omitting the corresponding parameter from the **DO** command’s actual parameter list. However, you must include a comma as a place holder for each omitted actual parameter. In the following example, the formal parameter b is left undefined:

Main
  SET x=1,y=2,z=3
  WRITE !,"In Main ",x,y,z
  DO Sub1(x,,z)
   WRITE !,"Back in Main ",x,y,z
  QUIT
Sub1(a,,c)
  WRITE !,"In Sub1 
    IF $DATA(a) {WRITE !,"a="a}
    ELSE {WRITE !,"a is undefined"}
    IF $DATA(b) {WRITE !,"b="b}
    ELSE {WRITE !,"b is undefined"}
    IF $DATA(c) {WRITE !,"c="c}
    ELSE {WRITE !,"c is undefined"}
  QUIT

You can specify a variable number of parameters using ... syntax:

Main
  DO Sub1(x,...)
  QUIT
Sub1(a,,c)
  WRITE a," ",b," ",c
  QUIT
The **DO** command can pass parameters either *by value* (for example, **DO** Sub1(x,y,z)) or *by reference* (for example, **DO** Sub1(.x,.y,.z)). You can mix passing by value and passing by reference within the same **DO** command. For further details, refer to **Parameter Passing** in *Using Caché ObjectScript*.

The following examples show the difference between passing by value and passing by reference:

```objectscript
Main /* Passing by Value */
  SET x=1,y=2,z=3
  WRITE !,"In Main ",x,y,z
  DO Sub1(x,y,z)
  WRITE !,"Back in Main ",x,y,z
  QUIT
Sub1(a,b,c)
  SET a=a+1,b=b+1,c=c+1
  WRITE !,"In Sub1 ",a,b,c
  QUIT

Main /* Passing by Reference */
  SET x=1,y=2,z=3
  WRITE !,"In Main ",x,y,z
  DO Sub1(.x,.y,.z)
  WRITE !,"Back in Main ",x,y,z
  QUIT
Sub1(&a,&b,&c)   /* The & prefix is an optional by-reference marker */
  SET a=a+1,b=b+1,c=c+1
  WRITE !,"In Sub1 ",a,b,c
  QUIT
```

**DO with Indirection**

You can use indirection to supply a target subroutine location for **DO**. For example, you might implement a generalized menu program by storing the various menu functions at different locations in a separate routine. In your main program code, you could use name indirection to provide the **DO** command with the location of the function corresponding to each menu choice.

You cannot use indirection with Caché object dot syntax. This is because dot syntax is parsed at compile time, not at runtime.

In *name indirection*, the value of the expression to the right of the indirection operator (@) must be a name (that is, a line label or a routine name). In the following code segment, name indirection supplies the **DO** with the location of a target function in the routine Menu.

```objectscript
READ !,"Enter the number for your choice: ",num QUIT:num="
DO @("Item"_num)^Menu
```

The **DO** command invokes the subroutine in Menu whose label is Item concatenated with the user-supplied *num* value (for example, Item1, Item2, and so on).

You can also use the *argument* form of indirection to substitute the value of an expression for a complete **DO** argument. For example, consider the following **DO** command:

```objectscript
DO @({eref":fstr>0")
```

This command calls the subroutine specified by the value of *eref* if the value of fstr is greater than 0.

For more information, refer to **Indirection** in *Using Caché ObjectScript*.

**DO with Argument Postconditionals**

You can use argument postconditional expressions to select a target subroutine for a **DO** command. If the postconditional expression evaluates to **FALSE** (0), Caché ignores the associated subroutine call. If the postconditional expression evaluates to **TRUE** (1), Caché executes the associated subroutine call, then returns to the **DO** command. You can use postconditionals on both the **DO** command and on its arguments.

For example, consider the command:

```objectscript
DO:F>0 A:F=1,B:F=2,C
```
The **DO** command has a postconditional expression; if \( F \) is not greater than 0, no part of the **DO** is executed. The **DO** command's arguments also have postconditional expressions. **DO** uses these argument postconditionals to select which subroutine(s) (A, B, or C) to execute. All subroutines that fulfill the truth condition are executed, in the order presented. Thus, in the above example, C, with no postconditional, is always executed: if \( F=1 \) both A and C are executed; if \( F=2 \), B and C are executed; if \( F=3 \) (or any other number) C is executed. To establish C as a true default, do the following:

```objectScript
DO:F>0 A:F=1,B:F=2,C:((F'=1)&&(F'=2))
```

In this example, one and only one subroutine is executed.

In the following example, the **DO** command takes a postconditional, and each of its arguments also takes a postconditional. In this case, the first argument is not executed, because its postconditional is 0. The second argument is executed because its postconditional is 1.

```objectScript
Main
   SET x=1,y=2,z=3
   WRITE !,"In Main ",x,y,z
   DO:1 Sub1(x,y,z):0,Sub2(x,y,z):1
   WRITE !,"Back in Main ",x,y,z
   QUIT
Sub1(a,b,c)
   WRITE !,"In Sub1 ",a,b,c
   QUIT
Sub2(d,e,f)
   WRITE !,"In Sub2 ",d,e,f
   QUIT
```

Most Object (oref) methods invoked by **DO** can take an argument postconditional. However, SSYSYSTEM object methods cannot take an argument postconditional. Attempted to do so generates a <SYNTAX> error.

Note that because Caché evaluates expressions in strict left-to-right order, an argument that contains expressions is evaluated (and can generate an error) before Caché evaluates the argument postconditional.

When using argument postconditionals, make sure there are no unwanted side effects. For example, consider the following command:

```objectScript
DO @^Control(i):z=1
```

In this case, ^Control(i) contains the name of the subroutine to be called if the postconditional \( z=1 \) tests TRUE. Whether or not \( z=1 \), Caché evaluates the value of ^Control(i) and resets the current global naked indicator accordingly. If \( z=1 \) is FALSE, Caché does not execute the **DO**. However, it does reset the global naked indicator just as if it had executed the **DO**. For more details on the naked indicator, see [Naked Global Reference](#) in Using Caché Globals.

For more information on how postconditional expressions are evaluated, see [Command Postconditional Expressions](#) in Using Caché ObjectScript.

### $TEST Behavior with **DO**

Argumentless **DO** always preserves the value of the $TEST special variable.

When you use **DO** to call a procedure, Caché preserves the value of $TEST by restoring it to its state at the time of the call upon quitting the procedure. However, when you use **DO** to call a subroutine (either with or without parameter passing), Caché does not preserve the value of $TEST across the call.

To save the $TEST value across a **DO** call, you can explicitly assign it to a variable before the call. You can then reference the variable in code that follows the call.

The following code illustrates some unexpected $TEST behavior that can result when using **DO** with the legacy **IF** command (which sets $TEST). This behavior does not occur with the standard (code block) **IF** command, because the standard **IF** does not set $TEST.
Start ; This routine uses the legacy IF command syntax
  SET x=1
  IF x=1 DO Sub1(x) ; sets $TEST to TRUE (1)
  ELSE  DO Sub2(x)
  QUIT
Sub1(y) ; a subroutine that evaluates a FALSE IF
  WRITE !,"hello from subroutine 1"
  IF y=2 WRITE " - IF in Sub1 was TRUE" ; Set $TEST to FALSE (0)
  ELSE  WRITE " - IF in Sub1 was FALSE"
  QUIT
Sub2(z) ; another subroutine
  WRITE !,"hello from subroutine 2"
  QUIT

At first glance, you might expect that Start will call only Sub1 and then exit.

In fact, execution of this code produces the following:

USER> DO ^Start
hello from subroutine 1 - IF in Sub1 was FALSE
hello from subroutine 2

This unexpected behavior results from the fact the $TEST value is reset in Sub1, that causes Caché to execute the ELSE command in Start. The processing sequence is as follows:

1. Caché evaluates the IF command expression in Start as TRUE. It sets $TEST to TRUE and calls Sub1.
2. Caché evaluates the IF command expression in Sub1 as FALSE. It sets $TEST to FALSE and then executes the following ELSE command.
3. Caché returns to Start when it encounters the QUIT.
4. Caché executes the ELSE in Start and performs the DO call to Sub2. It executes the ELSE because $TEST was set to FALSE in Sub1, replacing the TRUE value set by the IF command in Start.

To produce the expected behavior, you could replace the ELSE in either Start or Sub1 with an additional IF. For example, you might recast Start as follows:

Start
  SET x=1
  IF x=1 DO Sub1(x)
  IF x'=1 DO Sub2(x)
  QUIT

See Also

• GOTO command
• XECUTE command
• NEW command
• QUIT command
• Argumentless DO (legacy version) command
• IF (legacy version) command
• $CASE function
• $ESTACK special variable
• $STACK special variable
• Subroutines in Using Caché ObjectScript
• Procedures in Using Caché ObjectScript
• Parameter Passing in Using Caché ObjectScript
DO WHILE

Executes code while a condition exists.

DO {code} WHILE expression,...
D {code} WHILE expression,...

Arguments

<table>
<thead>
<tr>
<th>code</th>
<th>A block of ObjectScript commands enclosed in curly braces.</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A boolean test condition expression, or a comma-separated list of boolean test condition expressions.</td>
</tr>
</tbody>
</table>

Description

DO WHILE executes code, then evaluates expression. If expression evaluates to TRUE, DO WHILE loops and re-executes code. If expression is not TRUE, code is not re-executed, and the next command following DO WHILE is executed.

Note that DO WHILE is always written in block-oriented form. The code to be executed is placed between the DO and the WHILE keywords, and is enclosed by curly braces.

An opening or closing curly brace may appear on its own code line or on the same line as a command. An opening or closing curly brace may even appear in column 1 (though this is not recommended). It is a recommended programming practice to indent curly braces to indicate the beginning and end of a nested block of code. No whitespace is required before or after an opening curly brace. No whitespace is required before or after a closing curly brace. A comment may appear between the closing curly brace and the WHILE keyword.

The DO keyword may be abbreviated.

DO WHILE (unlike the unrelated DO command) does not create a new execution level. Commands that are sensitive to the execution level, such as NEW and SET $ZTRAP, that are invoked during the DO WHILE loop remain in effect after the loop concludes.

Arguments

code

A block of one or more ObjectScript commands. The code block may span several lines. The code block is enclosed by curly braces. The commands and comments within the code block and arguments within commands may be separated from one another by one or more blank spaces and/or line returns. However, as in all ObjectScript commands, each command keyword must be separated from its first argument by exactly one space.

expression

A test condition which can take the form of a single expression or a comma-separated list of expressions. For an expression list, Caché evaluates the individual expressions in left to right order. It stops evaluation if it encounters an expression that is FALSE. If all expressions evaluate to TRUE, Caché re-executes the code commands. DO WHILE executes repeatedly, testing expression for each loop. If any expression evaluates to FALSE, Caché ignores any remaining expressions, and does not loop. It executes the next command after DO WHILE.

Note: Caché performs no validation of expression before executing code. Therefore, DO WHILE always executes its code loop once, regardless of whether expression can successfully execute.
**Examples**

The following examples show first a **DO WHILE** in which expression is **TRUE**, and then a **DO WHILE** in which expression is **FALSE**. When expression is **FALSE**, the code block is executed once.

**DoWhileTrue**

```objectscript
SET x=1
DO { 
    WRITE !,"Looping",x
    SET x=x+1
} WHILE x<10 
WRITE !,"DONE"
```

This program writes **Looping1** through **Looping9** and then **DONE**.

**DoWhileFalse**

```objectscript
SET x=11
DO { 
    WRITE !,"Looping",x
    SET x=x+1
} WHILE x<10 
WRITE " DONE"
```

This program writes **Looping11** **DONE**.

**Notes**

**DO WHILE and WHILE**

The **DO WHILE** command executes the loop once and then tests *expression*. The **WHILE** command tests *expression* before executing the loop.

**DO WHILE and CONTINUE**

Within the *code* block of a **DO WHILE** command, encountering a **CONTINUE** command causes execution to immediately jump to the **WHILE** keyword. The **DO WHILE** command then evaluates the **WHILE expression** test condition, and, based on that evaluation, determines whether to re-execute the *code* block loop. Thus, the **CONTINUE** command has exactly the same effect on execution as reaching the closing curly brace of the *code* block.

The following example only displays the results of its odd-numbered loops:

```objectscript
SET x=0
DO {SET x=x+1 IF x#2=0 {CONTINUE} WRITE !,"Loop ",x} WHILE x<20
WRITE !,"DONE"
```

**DO WHILE, QUIT, and RETURN**

The **QUIT** command within the *code* block ends the **DO WHILE** loop and transfers execution to the command following the **WHILE** keyword, as shown in the following example:

```objectscript
Testloop
SET x=1
DO { 
    WRITE !,"Looping",x
    QUIT:x=5
    SET x=x+1
} WHILE x<10 
WRITE !,"DONE"
```

This program writes **Looping1** through **Looping5** and then **DONE**.

**DO WHILE** code blocks may be nested. That is, a **DO WHILE** code block may contain another flow-of-control loop (another **DO WHILE**, or a **FOR** or **WHILE** code block). A **QUIT** in an inner nested loop breaks out of the inner loop, to the next enclosing outer loop. This is shown in the following example:
Nestedloops
  SET x=1,y=1
  DO {
    WRITE "outer loop ",!
    DO {
      WRITE "inner loop 
      WRITE " y=",y,!
      QUIT:y=7
      SET y=y+2
    } WHILE y<100
    WRITE "back to outer loop x=",x,!!
    SET x=x+1
  } WHILE x<6
  WRITE "Done"

You can use RETURN to terminate execution of a routine at any point, including from within a DO WHILE loop or nested loop structure. RETURN always exits the current routine, returning to the calling routine or terminating the program if there is no calling routine. RETURN always behaves the same, regardless of whether it is issued from within a code block.

**DO WHILE and GOTO**

A GOTO command within the code block may direct execution to a label outside the loop, terminating the loop. (A label is also sometimes referred to as a tag.) A GOTO command within the code block may direct execution to a label within the same code block. A GOTO can be used to exit a nested code block. Other uses of GOTO, though supported, are not recommended.

The following example uses GOTO to exit a DO WHILE loop:

```objectscript
mainloop
  DO {
    WRITE !,"In an infinite DO WHILE loop"
    GOTO label1
  } WHILE 1=1
  WRITE !,"This should not display"
label1
  WRITE !,"Went to label1 and quit"
QUIT
```

The following example uses GOTO to transfer execution within a DO WHILE loop:

```objectscript
mainloop ; Example of a GOTO to within the code block
  SET x=1
  DO {
    WRITE !,"In the DO WHILE loop"
    GOTO label1
    WRITE !,"This should not display"
  label1
    WRITE !,"Still in the DO WHILE loop after GOTO"
    SET x=x+1
    WRITE !,"x= ",x
  } WHILE x<3
  WRITE !,"DO WHILE loop done"
```

**See Also**

- WHILE command
- FOR command
- IF command
- CONTINUE command
- GOTO command
- QUIT command
- RETURN command
**ELSE**

Clause of block-oriented **IF** command.

```
ELSE { code }
```

Refer to **IF** command for complete syntax.

**Description**

**ELSE** is not a separate command, but a clause of the block-oriented **IF** command. You can specify a single **ELSE** clause as the final clause of an **IF** command, or you can omit the **ELSE** clause. Refer to the **IF** command for details and examples.

**Note:** An earlier version of the **ELSE** command may exist in legacy applications where it is used with a line-oriented **IF** command. These commands may be recognized because they do not use curly braces. The old and new forms of **IF** and **ELSE** are syntactically different and should not be combined; therefore, an **IF** of one type should not be paired with an **ELSE** of the other type.

The earlier line-oriented **ELSE** command could be abbreviated as E. The block-oriented **ELSE** keyword cannot be abbreviated.

The **ELSE** keyword must be followed by an opening and closing curly brace ({}). Usually these curly braces enclose a block of code. However, an **ELSE** with no code block is permissible, as in the following:

```
SET x=1
Loop
  IF x=1{
    WRITE "Once only"
    SET x=x+1
    GOTO Loop
  }
ELSE{}
WRITE !,"All done"
```

There are no whitespace restrictions on the **ELSE** keyword.

**See Also**

- **IF** command
- **IF (legacy version)** command
- Flow of Control Commands in *Using Caché ObjectScript*
ELSEIF

Clause of block-oriented IF command.

```
ELSEIF expression,... { code }
```

Refer to IF command for complete syntax.

**Description**

ELSEIF is not a separate command, but a clause of the block-oriented IF command. You can specify one or more ELSEIF clauses in an IF command, or you can omit the ELSEIF clause. Refer to the IF command for details and examples.

**See Also**

- IF command
- Flow of Control Commands in Using Caché ObjectScript
FOR

Executes a block of code repeatedly, testing at the beginning of each loop.

```
FOR var=forparameter { code }
F var=forparameter { code }
FOR var=forparameter1,forparameter2,... { code }
F var=forparameter1,forparameter2,... { code }
```

where `forparameter` can be:

```
expr
start:increment
start:increment:end
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>var</td>
<td><code>Optional</code> — A local variable or instance variable initialized by the <code>FOR</code> command. Commonly, this a numeric counter that is incremented each time the <code>code</code> block is executed.</td>
</tr>
<tr>
<td>expr</td>
<td><code>Optional</code> — The value assigned to <code>var</code> before executing the <code>code</code> block. Can be a single value or a comma-separated list of values.</td>
</tr>
<tr>
<td>start</td>
<td><code>Optional</code> — The numeric value assigned to <code>var</code> before the first execution of the <code>code</code> block. Used with <code>increment</code> and (optionally) <code>end</code> to govern multiple iterations of the <code>FOR</code> loop.</td>
</tr>
<tr>
<td>increment</td>
<td><code>Optional</code> — The numeric value used to increment (or decrement) <code>var</code> after each iteration of the <code>FOR</code> loop.</td>
</tr>
<tr>
<td>end</td>
<td><code>Optional</code> — The numeric value used to terminate the <code>FOR</code> loop. Looping ends when <code>var</code> is incremented to a value equal to or greater than <code>end</code>.</td>
</tr>
<tr>
<td>code</td>
<td>A block of ObjectScript commands enclosed in curly braces.</td>
</tr>
</tbody>
</table>

**Description**

`FOR` is a block-oriented command. Commonly, it consists of a counter and an executable block of code enclosed in curly braces. The number of times this block of code is executed is determined by the counter, which is tested at the top of each loop. Less commonly, a `FOR` command does not specify an incrementing counter. It can be argumentless (looping infinitely until exited), or can specify an expression as its argument (looping once).

The `FOR` command has two basic forms:

- Without an argument
- With an argument

**FOR Without an Argument**

`FOR` without an argument executes the loop code block infinitely until exited by a command within the code block. Caché repeats the commands within the curly braces until it encounters a `QUIT`, `RETURN`, or `GOTO` command that breaks out of the loop. The following example exits the loop when \( x=3 \):
SET x=8
FOR { WRITE "Running loop x=",x,!
    SET x=x-1
    QUIT:x=3
}
WRITE "Next command after FOR code block"

An error, of course, also breaks out of a FOR loop, as shown in the following example. This FOR loop is exited by a divide-by-zero error, caught by the CATCH block:

TRY {
    SET x=8
    FOR { SET y=4/x
        WRITE "Running loop 4/",x,"=",y,!
        SET x=x-1
    }
    WRITE "Next command after FOR code block"
    CATCH exp {
        WRITE !,"this is the exception handler",!
        IF 1=exp.%IsA("%Exception.SystemException") {
            WRITE "Name: ",ZCVT(exp.Name,"O","HTML"),!
            WRITE "Location: ",exp.Location,!
        } ELSE {WRITE "Unexpected exception type",!
    }
    RETURN
}

FOR With an Argument

The action FOR performs depends on the argument form you use:

FOR var=expr executes the code block once, setting var to the value of expr. FOR var=expr1,expr2...,exprN executes the code block N times, setting var for each loop to each successive value of expr.

FOR var=start:increment executes the code block infinitely, unless exited. On the first iteration, Caché sets var to the value of start. Each execution of the FOR command increments the var value by the specified increment value. Caché repeats until it encounters a QUIT, RETURN, or GOTO command in the code block.

FOR var=start:increment:end sets var to the value of start. Caché then executes the code block based on the conditions described in this table:

<table>
<thead>
<tr>
<th>increment is positive</th>
<th>increment is negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>If start &gt; end, do not execute the code block. Stop looping when var is equal to or greater than end, or if Caché encounters a QUIT, RETURN, or GOTO command.</td>
<td>If start &lt; end, do not execute the code block. Stop looping when var is equal to or less than end, or if Caché encounters a QUIT, RETURN, or GOTO command.</td>
</tr>
</tbody>
</table>

Caché evaluates the start, increment, and end values when it begins execution of the loop. Any changes made to these values within the loop are ignored and have no effect on the number of loop executions.

When the loop terminates, var contains a value that reflects the increment resulting from the last execution of the loop. However, var is never incremented beyond the maximum value specified in end.
A **FOR** loop can include multiple comma-separated *forparameter* arguments, but only one *var* argument. Valid syntax is as follows:

```objectscript
FOR var=start1:increment1:end1,start2:increment2:end2
```

### Arguments

**var**

The *var* argument can be a simple local variable, a subscripted local variable, or an instance variable (such as `i%property`). The **FOR** command initializes (or sets) this variable; *var* does not need to be defined prior to the **FOR** command.

- When using loop counter syntax, *var* is a local variable that holds the current counter value for the **FOR** loop. It initially contains the numeric value specified by *start*. It is then recalculated using *increment* for each repetition of the **FOR** loop.
- When using *var=expr* syntax, a local variable initialized to the *expr* value.

**expr**

The value Caché assigns to *var* before executing the loop commands. The value of *expr* can be specified as a literal or any valid expression. An *expr* can be a single value or a comma-separated list of values. If a single value, Caché executes the **FOR** loop once, supplying this *var* value to the code block. If a comma-separated list of values, Caché executes the **FOR** loop as many times as there are values, each loop equating *var* to the value, and supplying this *var* value to the code block.

**start**

The numeric value Caché assigns to *var* on the first iteration of the **FOR** loop. The value of *start* can be specified as a literal or any valid expression; Caché evaluates *start* for its numeric value. A non-numeric *start* value is evaluated as 0.

**increment**

The numeric value Caché uses to increment *var* after each iteration of the **FOR** loop. It is required if you specify *start*. The value of *increment* can be specified as a literal or any valid expression; Caché evaluates *increment* for its numeric value. *increment* can be an integer or a fractional number; it can be a positive number (to increment) or a negative number (to decrement).

A value of 0 or a non-numeric *increment* value causes the **FOR** loop to repeat infinitely, unless exited. This is true even if *end*=0. However, if *start* is greater than *end*, the loop does not execute and an *increment* of 0 is ignored.

**end**

The numeric value Caché uses to terminate a **FOR** loop. When *var* equals (or exceeds) this value, the **FOR** loop is executed one last time and then terminated. The value of *end* can be specified as a literal or any valid expression; Caché evaluates *end* for its numeric value.

If *start=* *end* the **FOR** loop executes once. If *start* is greater than *end* (and *increment* is a positive number) the **FOR** loop does not execute.

**code**

A block of one or more ObjectScript commands enclosed in curly braces. This executable block of code can contain multiple commands, labels, comments, line returns, indents, and blank spaces as needed. When the **FOR** command concludes, execution continues with next command after the closing curly brace.

An opening or closing curly brace may appear on its own code line or on the same line as a command. An opening or closing curly brace may even appear in column 1 (though this is not recommended). It is a recommended programming practice to indent curly braces to indicate the beginning and end of a nested block of code. No whitespace is required before or after an opening curly brace. No whitespace is required before a closing curly brace, including a curly brace that follows...
an argumentless command. There is only one whitespace requirement for curly braces: a closing curly brace must be separated from the command that follows it by a space, tab, or line return.

**Exiting a FOR Loop**

You can exit a FOR loop by issuing a QUIT, RETURN, CONTINUE, or GOTO:

- **QUIT** exits the current FOR block structure. Therefore, a QUIT in a FOR block causes Caché to begin execution at the next line after the FOR block. A QUIT only exits the current FOR block; if a FOR block is nested in another FOR block (or in any other block structure), issuing a QUIT exits the inner FOR block to the outer block structure.

  QUIT behavior within a FOR block (and some other block structures) differs from QUIT behavior when not in a block structure. A QUIT outside of one of these block structures exits the current routine, not just the current code block. For further details, refer to the QUIT command reference page.

  A QUIT exits a FOR block only if the QUIT appears within the FOR block. If the FOR loop invokes a subroutine, issuing a QUIT in the subroutine terminates the subroutine, not the FOR loop that invoked it.

- **RETURN** exits the current routine, whether or not it is issued from within a FOR block structure.

- **CONTINUE** exits the current FOR loop. It causes execution to immediately jump back to the FOR command. The FOR command then increments and evaluates its arguments, and, based on that evaluation, determines whether to re-execute the code block loop. Thus, the CONTINUE command has exactly the same effect on execution as reaching the closing curly brace of the code block.

- **GOTO** can exit the current FOR block structure by transferring control outside of the FOR code block. A FOR loop is not terminated by a GOTO that transfers control within the FOR code block.

**FOR Loop or WHILE Loop**

You can use either a FOR or a WHILE to perform the same operation: loop until an event causes execution to break out of the loop. However, which loop construct you use has consequences for performing single-step (BREAK "S+" or BREAK "L+") debugging on the code module.

A FOR loop pushes a new level onto the stack. A WHILE loop does not change the stack level. When debugging a FOR loop, popping the stack from within the FOR loop (using BREAK "C" GOTO or QUIT 1) allows you to continue single-step debugging with the command immediately following the end of the FOR command construct. When debugging a WHILE loop, issuing a using BREAK "C" GOTO or QUIT 1 does not pop the stack, and therefore single-step debugging does not continue following the end of the WHILE command. The remaining code executes without breaking.

For further details, refer to the BREAK command and Debugging with BREAK in the “Command-line Routine Debugging” chapter of *Using ObjectScript*.

**Examples**

**Argumentless FOR**

In the following example, demonstrating argumentless FOR, the user is prompted repeatedly for a number that is then passed to the Calc subroutine by the DO command. The FOR loop terminates when the user enters a null string (presses ENTER without inputting a number), which causes the QUIT command to execute.

```
Mainloop
  FOR { 
    READ !,*Number: *,num
    QUIT:num=""
    DO Calc(num)
  }
Calc(a)
  WRITE !,*The number squared is *,a*a
  QUIT
```
Using FOR var=expr

When you specify var=expr, Caché executes the FOR loop as many times as there are comma-separated values in expr. The value(s) in expr can be a literal or any valid expression. If you specify an expression, it must evaluate to a single value.

In the following example, the FOR command executes the code block once, with num having the value 4. It writes the number 12:

```objectscript
Loop
  SET val=4
  FOR num=val {
    WRITE num*3,! 
  }
  WRITE "Next command after FOR code block"
```

In the following example, the FOR command executes the code block once, with alpha(7) having the value “abcdefg”:

```objectscript
Loop
  SET val="abc"
  FOR alpha(7)=val_"defg" {
    WRITE alpha(7),! 
  }
  WRITE "Next command after FOR code block"
```

In the following example, the FOR command executes the code block eight times, supplying each successive perfect number to the code block:

```objectscript
FOR pnum=6,28,496,8128,433550336,8589869056,137438691328,2305843008139952128 {
  WRITE "Perfect number ",pnum
  SET rp=$REVERSE(pnum)
  IF 54=$ASCII(rp,1) {
    WRITE " ends in 6",! }
  ELSEIF 56=$ASCII(rp,1),50=$ASCII(rp,2) {
    WRITE " ends in 28",! }
  ELSE {WRITE " is something unknown to mathematics",! }
}
```

Using FOR var=start:increment:end

The arguments start, increment, and end specify a start, increment, and end value, respectively. All three are evaluated as numbers. They can be integer or real, positive or negative. If you supply string values, they are converted to their numeric equivalents at the start of the loop.

When Caché first enters the loop, it assigns the start value to var and compares the var value to the end value. If the var value is less than the end value (or greater than it, in the case of a negative increment value), Caché executes the loop commands. It then updates the var value using the increment value. (The var value is decremented if a negative increment is used.)

Execution of the loop continues until the incrementing of the var value would exceed the end value (or until Caché encounters a QUIT, RETURN, or GOTO). At that point, to prevent var from exceeding end, Caché suppresses variable assignment and loop execution ends. If the increment causes the var value to equal the end value, Caché executes the FOR loop one last time and then terminates the loop.

The following code executes the WRITE command repetitively to output, in sequence, all of the characters in string1, except for the last character. Because the end value is specified as len-1, the last character is not output. This is because the test is performed at the top of the loop, and the loop is terminated when the variable value (index) exceeds (not just matches) the end value (len-1).

```objectscript
Stringwriteloop
  SET string1="123 Primrose Path"
  SET len=$LENGTH(string1)
  FOR index=1:1:len-1 {
    WRITE $EXTRACT(string1,index) 
  }
```
**Using FOR var=start:increment**

In this form of the FOR command there is no end value; the loop must contain a QUIT, RETURN, or GOTO command to terminate the loop.

The start and increment values are evaluated as numbers. They can be integer or real, positive or negative. If string values are supplied, they are converted to their numeric equivalents at the start of the loop. Caché evaluates the start and increment values when it begins execution of the loop. Any changes made to these values within the loop are ignored.

When Caché first enters the loop, it assigns the start value to var and executes the loop commands. It then updates the var value using the increment value. (The var value is decremented if a negative increment is used.) Execution of the loop continues until Caché encounters a QUIT, RETURN, or GOTO within the loop.

The following example uses start:increment syntax to return all of the multiples of 7 that are less than three digits in length:

```
FOR i(1)=0:7 { 
    QUIT:$LENGTH(i(1))=3 
    WRITE "multiple of 7 = ",i(1),! }
```

The following example uses start:increment syntax to compute an average for a series of user-supplied numbers. The postconditional QUIT is included to terminate execution of the loop when the user enters a null string (that is, presses ENTER without inputting a value). When the postconditional expression (num="") tests TRUE, Caché executes the QUIT and terminates the loop.

The loop counter (the i variable) is used to keep track of how many numbers have been entered. i is initialized to 0 because the counter increment occurs after the user inputs a number. Caché terminates the loop when the user enters a null. After the loop is terminated, the SET command references i (as a local variable) to calculate the average.

```
Averageloop
SET sum=0
FOR i=0:1 { 
    READ !,"Number: ",num
    QUIT:num=""
    SET sum=sum+num
}
SET average=sum/i
```

**Using FOR with Multiple forparameters**

A FOR command can contain only one var= argument, but can contain multiple forparameter arguments, specified as a comma-separated list. For example, the syntax var=expr1,expr2,expr3 would cause the code block to be executed three times, with a different var value for each execution.

These forparameter arguments are evaluated and executed in strict left-to-right order. Therefore an error in one forparameter does not prevent the execution of the forparameters that precede it.

A single FOR command can contain both types of parameter syntax: expr syntax and start:increment:end syntax.

The following example combines expr syntax with start:increment:end syntax. The two forparameters are separated by a comma. The first time through the FOR, Caché uses the expr syntax, and invokes the Test subroutine with x equal to the value of y. In the second (and subsequent) iterations, Caché uses the start:increment:end syntax. It sets x to 1, then 2, etc.

On the final iteration, x=10.

```
Mainloop
SET y="beta"
FOR x=y,1:1:10 { 
    DO Test
}
QUIT
Test
WRITE !,"Running test number ",x
QUIT
```

The following example is a sampling program that includes three forparameter arguments with start:increment:end syntax. It sets i to 1, then increments single-digit numbers by 1 for 1 through 10; the second forparameter takes the i value of 10
and increments it by 10s through 100; the third forparameter takes the i value of 100 and increments it by 100s through 1000. Note that this example repeats the 10 and 100 values:

```
FOR i=1:1:10, i=10:100, i=100:1000 {WRITE i, !}
```

The following example performs the same operation as the previous example, without repeating the 10 and 100 values:

```
FOR i=1:1:9, i+1:10:99, i+10:100:1000 {WRITE i, !}
```

**Incrementing with Argumentless FOR**

The argumentless FOR operates the same as the FOR var=start:increment form. The only difference is that it does not provide a way to keep track of the number of loop executions.

The following example shows how the previous loop counter example might be rewritten using the argumentless FOR. The assignment i=i+1 replaces the loop counter.

```
Average2loop
SET sum=0
SET i=0
FOR {
   READ !,"Number: ",num QUIT:num="
   SET sum=sum+num, i=i+1
}
SET average=sum/i
WRITE !!, "Average is: ",average
QUIT
```

**Notes**

**FOR and NEW**

A NEW command can affect var. Issuing an argumentless NEW command or an exclusive NEW command (that does not specifically exclude var) in the body of the FOR loop can result in var being undefined in the new frame context.

A NEW command that does not include var has no effect on FOR loop execution, as shown in the following example:

```
SET a=1,b=1,c=8
FOR i=a:b:c {
   WRITE !,"count is ",i
   NEW a,c
   WRITE " loop"
   NEW (i)
   WRITE " again"
}
```

**FOR and Watchpoints**

You have limited use of watchpoints with FOR. If you establish a watchpoint for the control (index) variable of a FOR command, Caché triggers the specific watchpoint action only on the initial evaluation of each FOR command argument. This restriction is motivated by performance considerations.

The following example contains three kinds of FOR command arguments for the watched variable x: a range, with initial value, increment, and limit (final value); a single value; and a range with initial value, increment, and no limit. Breaks occur when x has the initial values 1, 20, and 50.

```
USER> ZBREAK *x
USER> FOR x=1:1:10,20,50:2 {SET t=x QUIT:x>69} <BREAK>
USER 2f0> WRITE x=1
USER 2f0> g
USER> FOR x=1:1:10,20,50:2 {SET t=x QUIT:x>69} <BREAK>
USER 2f0> WRITE t=10
x=20
USER> 2f0> g
USER> FOR x=1:1:10,20,50:2 {SET t=x QUIT:x>69}
```
FOR

See Also

- FOR (legacy version) command
- DO WHILE command
- WHILE command
- IF command
- CONTINUE command
- DO command
- QUIT command
- RETURN command
GOTO

Transfers control.

```
GOTO:pc
GOTO:pc goargument,...
G:pc
G:pc goargument,...
```

where *goargument* is:

```
location:pc
```

**Arguments**

<table>
<thead>
<tr>
<th>pc</th>
<th>Optional — A postconditional expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>Optional — The point to which control will be transferred.</td>
</tr>
</tbody>
</table>

**Description**

The GOTO command has two forms:

- **Without an argument**
- **With an argument**

**GOTO Without an Argument**

GOTO without an argument resumes normal program execution after Caché encounters an error or a BREAK command in the currently executing code. You can use the argumentless GOTO only from the Terminal prompt.

The following example shows the use of an argumentless GOTO. In this example, the second WRITE is not executed because of the <BREAK> error; issuing a GOTO resumes execution, executing the second WRITE:

```
USER>WRITE "before" BREAK WRITE "after"
before
WRITE "before" BREAK WRITE "after" ^
<BREAK>
USER 1SO>GOTO
after
USER>
```

Note that there must be two spaces after the BREAK command.

If a NEW command is in effect when you issue an argumentless GOTO, Caché issues a <COMMAND> error, and the new context is maintained. Use the QUIT 1 command, then argumentless GOTO to resume after a NEW.

Argumentless GOTO can also be used at the Terminal prompt to continue execution after an error. See Processing Errors at the Terminal Prompt in the “Error Processing” chapter of Using Caché ObjectScript.

**GOTO With an Argument**

GOTO with the argument *location* transfers control to the specified location. If you specify a postconditional expression on either the command or the argument, Caché transfers control only if the postconditional expression evaluates to TRUE (nonzero).

You can use GOTO *location* from the Terminal prompt to resume an interrupted program at a different location.

You can specify a $CASE function as a GOTO command argument.
Arguments

**pc**

An optional postconditional expression that can make the command conditional. If the postconditional expression is appended to the `GOTO` command keyword, Caché executes the `GOTO` command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the `GOTO` command if the postconditional expression is false (evaluates to zero). If the postconditional expression is appended to an argument, Caché executes the argument if the postconditional expression is true (evaluates to a nonzero numeric value). If the postconditional expression is false (evaluates to zero), Caché skips that argument and evaluates the next argument (if there is one) or the next command. For further details, refer to Command Postconditional Expressions in *Using Caché ObjectScript*.

**location**

The point to which control will be transferred. It is required in routine code. It is optional from the Terminal prompt. You can specify `location` as a single value or as a comma-separated list of values (with postconditional expressions) and can take any of the following forms:

- `label+offset` specifies a line label within the current routine. The optional `+offset` is a nonnegative integer that specifies the number of lines after the label at which execution is to start. The `offset` counts lines of code (including label lines), and counts comment lines; `offset` does not count blank lines and blank lines within comments. However, `offset` does count all Embedded SQL lines, including all blank lines.

- `label+offset^routine` specifies a line label within the named routine, which resides on disk. Caché loads the routine from disk and continues execution at the indicated label. The optional `+offset` is a nonnegative integer that specifies the number of lines after the label at which execution is to start.

  ^routine specifies a routine that resides on disk. Caché loads the routine from disk and continues execution at the first line of executable code within the routine. If the routine has been modified, Caché loads the updated version of the routine when `GOTO` invokes the routine. Unlike the `DO` command, `GOTO` does not return to the invoking program following routine execution. If you specify a nonexistent routine, Caché issues a `<NOROUTINE>` error message. For more information, refer to the `$ZERROR` special variable.

**Note:** `GOTO` does not support extended routine reference. To execute a routine in another namespace, use the `DO` command.

You can also reference `location` as a variable containing any of the above forms. In this case, though, you must use name indirection. `location` cannot specify a subroutine label that is defined with a formal parameter list or the name of a user-defined function or procedure. If you specify a nonexistent `label`, Caché issues a `<NOLINE>` error message. For more information, refer to Indirection in *Using Caché ObjectScript*.

You cannot specify an `offset` when calling a CACHESYS % routine. If you attempt to do so, Caché issues a `<NOLINE>` error.

**Examples**

In the following example, `GOTO` directs execution to one of three locations depending on the user-supplied age value. The location is a subroutine label that is stored in variable `loc` and then referenced by means of name indirection (`@loc`).

```objectscript
mainloop
  SET age=""
  READ !,"What is your age? ",age QUIT:age=""
  IF age<30 {
    SET loc="Young" }
  ELSEIF (age>29)&(age<60) {
    SET loc="Midage" }
  ELSEIF age>59 {
    SET loc="Elder" }
  ELSE {
    WRITE "data input error"
    QUIT }
  GOTO @loc
```
QUIT
Young
  WRITE !,"You're still young"
QUIT
Midage
  WRITE !,"You're in your prime"
QUIT
Elder
  WRITE !,"You have a lifetime of wisdom to impart"
QUIT

Note that this type of GOTO using name indirection is not permitted from within a procedure block.

As an alternative, you could omit the IF command and code the GOTO with a comma-separated list using postconditionals on the arguments, as follows:

GOTO Young:age<30,Midage:(age>29)&(age<60),Elder:age>59

You might also code this example using a DO command to call the appropriate subroutine location. In this case, though, when Caché encounters a QUIT, it returns control to the command following the DO.

The following example shows how offset counts lines of code. It counts the intervening label line and the comment line; it does not count the blank line:

Main
  GOTO Branch+7
QUIT
Branch
  WRITE "Line 1",!
SubBranch
  WRITE "Line 3",!
  /* comment line */
  WRITE "Line 5",!
  WRITE "Line 6",!
  WRITE "Line 7",!
  WRITE "Line 8",!
QUIT

Notes

How Control is Transferred When QUIT is Encountered

Unlike the DO command, GOTO transfers control unconditionally. When Caché encounters a QUIT in a subroutine called by DO, it passes control to the command following the most recent DO.

When Caché encounters a QUIT after a GOTO transfer, it does not return control to the command following the GOTO. If there was a preceding DO, it returns control to the command following the most recent DO. If there was no preceding DO, then it returns to the Terminal prompt.

In the following code sequence, the QUIT in C returns control to the WRITE command following the DO in A:

testgoto
A
  WRITE !,"running A"
  DO B
  WRITE !,"back to A, all done"
QUIT
B
  WRITE !,"running B"
  GOTO C
  WRITE !,"this line in B should never execute"
QUIT
C
  WRITE !,"running C"
QUIT

Using GOTO with Code Blocks

GOTO can be used to exit a code block, but not to enter a code block.
If you use **GOTO** inside a **FOR**, **IF**, **DO WHILE**, or **WHILE** loop, you can go to a *location* outside of all code blocks, a *location* within the current code block, or go from a nested code block to a *location* in the code block that encloses it. You cannot go from a code block to a *location* within another code block, either an independent code block, or a code block nested within the current code block. For code examples, refer to the individual commands.

A **GOTO** to a *location* outside a code block terminates the loop. A **GOTO** to a *location* within a code block does not terminate the loop. A **GOTO** from a nested code block to an enclosing code block terminates the inner (nested) loop, but not the outer loop.

A **GOTO** can be used to exit a **TRY** or **CATCH** code block, but not to enter one of these code blocks. You also cannot specify a **GOTO** to a label on the same line as the **TRY** or **CATCH** keyword. Attempting to do so results in a <NOLINE> error.

**GOTO Restrictions**

The following **GOTO** operations are not permitted:

- **GOTO** should not be used to enter or exit a procedure.
- **GOTO** should not be used within nested argumentless **DO** code blocks to jump between levels.
- **GOTO** cannot be used with name indirection (**GOTO @name**) within a procedure block.

**See Also**

- **DO** command
- **FOR** command
- **IF** command
- **DO WHILE** command
- **WHILE** command
- **BREAK** command
- **QUIT** command
- **$CASE** function
- **Processing Errors at the Terminal Prompt** in the “Error Processing” chapter of *Using Caché ObjectScript*
- **Debugging With BREAK** in the “Command-line Routine Debugging” chapter of *Using Caché ObjectScript*
HALT

Terminates execution of the current process.

HALT:pc
H:pc

**Argument**

| pc | Optional — A postconditional expression. |

**Description**

The HALT command terminates execution of the current process. If a $HALT special variable is defined in the current context (or a prior context), issuing a HALT command invokes the halt trap routine specified in $HALT, rather than terminating the current process. Typically, a halt trap routine performs some cleanup or reporting operations, then issues a second HALT command to terminate execution.

HALT behaves the same whether it is encountered by running routine code or is entered from the Terminal prompt. In either case, it terminates the current process.

HALT has the same minimum abbreviation as the HANG command. HANG is distinguished by its required hangtime argument.

**Effects of HALT**

When HALT terminates a process, the system automatically relinquishes all locks and closes all devices owned by the process. This ensures that the halted process does not leave behind locked variables or unreleased devices.

If there is a transaction in progress when HALT terminates a process, the resolution of the transaction depends on the type of process. A HALT in a background job (non-interactive process) always rolls back the transaction in progress. A HALT in an interactive process (such as using Terminal to run a routine) prompts you to resolve the transaction in progress. The prompt is as follows:

You have an open transaction.
Do you want to perform a (C)ommit or (R)ollback? R =>

Specify “C” to commit the current transaction. Specify “R” (or just press the Enter key) to roll back the current transaction.

**Halt Traps**

Execution of a HALT command is interrupted by a halt trap. Halt traps are established using the $HALT special variable.

If a halt trap has been established for the current context frame, issuing a HALT command invokes the halt trap routine specified by $HALT. The HALT command itself is not executed.

If a halt trap has been established for a lower context frame, a HALT command removes context frames from the frame stack until the context frame with the halt trap is reached. HALT then invokes the halt trap routine specified by $HALT and ceases execution.

**Arguments**

**pc**

An optional postconditional expression that can make the command conditional. Caché executes the HALT command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.
Examples

In the following example, `HALT` allows the user to end the current application and return to the operating system. The system performs all necessary cleanup for the user. Note the use of the postconditional on the command.

```
Main
  READ !,"Do you really want to stop (Y or N)? ",ans QUIT:ans=""
  HALT:(ans[="Y"]!(ans="y")
  DO Start
  Start()
    WRITE !,"This is the Start routine"
  QUIT
```

In the following example, `HALT` invokes the halt trap routine specified in `$HALT`. In this case, it is the second `HALT` command that actually halts execution. (For demonstration purposes, this example uses `HANG` statements so that you have time to view the displayed output.)

```
Main
  NEW $ESTACK
  SET $HALT="OnHalt"
  WRITE !,"Main $ESTACK= ",$ESTACK // 0
  HANG 2
  DO SubA
    WRITE !,"this should never display"
    SubA()
    WRITE !,"SubA $ESTACK= ",$ESTACK // 1
    HANG 2
    HALT // invoke the OnHalt routine
    WRITE !,"this should never display"
  OnHalt()
    WRITE !,"OnHalt $ESTACK= ",$ESTACK // 0
    HANG 2
    // clean-up and reporting operations
    HALT // actually halt the current process
```

`$SYSTEM.Process.Terminate()`

You can use the `$SYSTEM.Process.Terminate()` method to halt the current process or to halt other running processes.

The following example halts the current process:

```
DO $SYSTEM.Process.Terminate()
```

The following example halts the process with the PID 7732:

```
DO $SYSTEM.Process.Terminate(7732)
```

The effects of the `Terminate()` method are the same as the `HALT` command for the current process, or the `^RESJOB` utility for other processes.

`^RESJOB` and `^JOBEXAM`

The `HALT` command is used to halt the current process.

The `^RESJOB` or `^JOBEXAM` utility can be used to halt other running processes. These utilities cannot be used to halt the current process. They can be used to display information about all running processes, including the current process.

These utilities must be invoked from the `%SYS` namespace. You must have appropriate privileges to invoke these utilities. Utility names are case-sensitive.

- `^RESJOB` allows you to directly halt a process if you know the process ID (PID). You can use the ? option to display a listing of all of the running processes.

- `^JOBEXAM` first displays a listing of all of the running processes, then allows you to specify which process to halt (terminate), suspend, or resume. View `^JOBEXAM` allows you to display a listing of all of the running processes; it does not provide options to halt, suspend, or resume a process.

The following is an example invocation of `^RESJOB` from Terminal:
%SYS>DO ^RESJOB
Force a process to quit Cache
Process ID (? for status report): 7732
Process ID (? for status report):
%SYS>

At the prompt, you type the process ID (PID) for the process you wish to halt. ^RESJOB halts the process, then prompts you for the next process ID. Press the Enter key at the prompt when you are finished entering process IDs. You can specify ? at the prompt to display a list of currently running processes.

- Current process: attempting to use ^RESJOB to halt the current process fails with the message This is your current process, not proceeding with kill. ^RESJOB then prompts you for another process ID.
- Non-running process: specifying the process ID of a non-running process fails with the message [no such Cache process]. ^RESJOB then prompts you for another process ID.
- System processes: you cannot use ^RESJOB to halt certain system processes. Attempting to do so fails with the message Can NOT kill the name process. ^RESJOB then prompts you for another process ID.
- Transaction-in-progress: using ^RESJOB to halt a process with a transaction-in-progress is the same as issuing a HALT command in that process. A non-interactive process rolls back the incomplete transaction; an interactive process prompts you at its Terminal prompt to either commit or roll back the incomplete transaction.

See Also

- $HALT special variable
- Debugging in Using Caché ObjectScript
HANG

Suspends execution for a specified number of seconds.

```
HANG:pc hangarg
H:pc hangarg
```

where `hangarg` can be:

```
hangtime,...
```

**Arguments**

<table>
<thead>
<tr>
<th>pc</th>
<th>Optional — A postconditional expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>hangtime</td>
<td>The amount of time to wait, in seconds. An expression that resolves to a positive numeric value, or a comma-separated list of numeric expressions.</td>
</tr>
</tbody>
</table>

**Description**

HANG suspends the executing routine for the specified time period. If there are multiple arguments, Caché suspends execution for the duration of each argument in the order presented. The HANG time is calculated using the system clock, which determines its precision.

HANG has the same minimum abbreviation (H) as the HALT command. HANG is distinguished by its required `hangtime` argument.

**Arguments**

- **pc**

  An optional postconditional expression. Caché executes the HANG command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

- **hangtime**

  The amount of time to wait, in seconds. This time can be expressed as any numeric expression. You can specify `hangtime` as an integer to specify whole seconds, or as fractional number to specify fractional seconds. You can use exponentiation (**), arithmetic expressions, and other numeric operators.

  You can set `hangtime` to 0 (zero), in which case no hang is performed. Setting `hangtime` to a negative number or a nonnumeric value is the same as setting it to 0.

  You can specify multiple `hangtime` arguments as a comma-separated list, as described below.

**Examples**

The following example suspends the process for 10 seconds:

```
WRITE 1, $ZTIME($PIECE($HOROLOG,"\",2))
HANG 10
WRITE 1, $ZTIME($PIECE($HOROLOG,"\",2))
```

The following example suspends the process for 1/2 second. $ZTIME$STAMP, unlike $HOROLOG$, can return fractional seconds if the precision parameter of the $ZTIME$ function is specified.
HANG .5

WRITE !,$ZTIME($PIECE($ZTIMESTAMP,"",2),1,2)

Returns values such as the following:
14:34:19.75
14:34:20.25

Notes

Multiple HANG Arguments

You can specify hangtime as a comma-separated list of numeric expressions. Caché suspends execution for the duration of each argument in the order presented. Negative numbers are treated as zero. Therefore, a hangtime of 16,-15 would hang for 16 seconds.

That each hangtime argument is separately executed can affect operations that use the current time in hang calculations, as shown in the following example:

```objectscript
SET start=$ZHOROLOG
SET a=$ZHOROLOG+5
HANG 4,a-$ZHOROLOG
SET end=$ZHOROLOG
WRITE !,*elapsed hang=",end-start
```

In this example, HANG first suspends execution for 4 seconds, then suspends execution for the current time before the hang plus 5 seconds, minus the current time when the second hang argument is parsed. Because HANG executes each argument in turn, the total hang time in this example is (roughly) 5 seconds, rather than the (roughly) 9 seconds one might otherwise expect.

HANG Compared with Timed READ

You can use HANG to pause the routine while the user reads an output message. However, you can handle this type of pause more effectively with a timed READ command. A timed READ allows the user to continue when ready, but a HANG does not because it is set to a fixed duration.

HANG and ^JOBEXAM

The HANG command is used to pause execution of the current process.

The ^JOBEXAM utility can be used to suspend and resume execution of other running processes; it cannot be used to suspend execution of the current process. You cannot use ^JOBEXAM to resume process execution that has been paused by a HANG command. If you use ^JOBEXAM to suspend a process that has been paused by a HANG command, the ^JOBEXAM resume activates the process, which must complete whatever portion of the HANG time that remained when it was suspended.

^JOBEXAM displays State information about all running processes. A process that is paused by a HANG command is listed as State HANGW. A process that is suspended by ^JOBEXAM is listed as State SUSPW.

The ^JOBEXAM utility must be invoked by the Terminal from the %SYS namespace. You must have appropriate privileges to invoke this utility. Utility names are case-sensitive. You can execute ^JOBEXAM as follows:

• DO ^JOBEXAM: displays a listing of all running processes. It provides letter code options to terminate, suspend, or resume a running process.

• DO View^JOBEXAM displays a listing of all running processes. It does not provide options to terminate, suspend, or resume a process.

See Also

• READ command
• $ZTIME function
• $HOROLOG special variable
• $ZTIMESTAMP special variable
**IF**

Evaluates an expression, then selects which block of code to execute based on the truth value of the expression.

```
IF expression1,... {
  code
}
ELSEIF expression2,... {
  code
}
ELSE {
  code
}
```

or

```
I expression1,... {
  code
}
ELSEIF expression2,... {
  code
}
ELSE {
  code
}
```

### Arguments

<table>
<thead>
<tr>
<th>expression1</th>
<th>A boolean test condition for the IF clause. A single condition or a comma-separated list of conditions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression2</td>
<td>A boolean test condition for an ELSEIF clause. A single condition or a comma-separated list of conditions.</td>
</tr>
<tr>
<td>code</td>
<td>A block of ObjectScript commands enclosed in curly braces.</td>
</tr>
</tbody>
</table>

### Description

This page describes the **IF**, **ELSEIF**, and **ELSE** command keywords, all of which are considered to be component clauses of the **IF** command. The **IF** keyword can be abbreviated as **I**; the other two keywords cannot be abbreviated.

An **IF** command consist of one **IF** clause, followed by any number of **ELSEIF** clauses, followed by one **ELSE** clause. The **ELSEIF** and **ELSE** clauses are optional, but it is a good programming practice to always specify an **ELSE** clause.

The **IF** command first evaluates the **IF** clause `expression1` and, if `expression1` is TRUE, it executes the code block within the curly braces that follow it and the **IF** command completes.

If `expression1` is FALSE, execution jumps to the next clause of the **IF** statement. It evaluates the first **ELSEIF** clause (if present). If `expression2` in the **ELSEIF** clause is TRUE, it executes the **ELSEIF** code block within the curly braces that follow it and the **IF** command completes. If `expression2` is FALSE, the next **ELSEIF** clause (if present) is evaluated in the same way. Each successive **ELSEIF** clause is tested in the order listed until one of them evaluates TRUE, or all of them evaluate FALSE.

If the **IF** clause and all **ELSEIF** clauses evaluate to FALSE, execution continues with the **ELSE** clause. It executes the **ELSE** code block within the curly braces that follow it and the **IF** command completes. If the **ELSE** clause is omitted, the **IF** command completes.

**IF** is a block-oriented command. Each command keyword is followed by a block of code enclosed in curly braces. **IF**, **ELSEIF**, and **ELSE** clauses may use white space (line returns, indents, and blank spaces) freely. However, each **IF** and
ELSEIF keyword and the first character of its boolean test expression must be on the same line, separated by one blank space. A boolean test expression can span multiple lines and contain multiple blank spaces.

An opening or closing curly brace may appear on its own code line or on the same line as a command. An opening or closing curly brace may even appear in column 1 (though this is not recommended). It is a recommended programming practice to indent curly braces to indicate the beginning and end of a nested block of code. No whitespace is required before or after an opening curly brace. No whitespace is required before or after a closing curly brace, including a curly brace that follows an argumentless command. There is only one whitespace requirement for curly braces: the final closing curly brace of the last clause of the IF command must be separated from the command that follows it by a space, tab, or line return.

**Note:** An earlier, line-oriented IF command syntax may exist in legacy applications. This line-oriented form of IF does not use curly braces or support the ELSEIF clause. These two forms of IF are syntactically different and should not be combined. Thus, an IF of one type should not be paired with an ELSE of the other type. The earlier line-oriented ELSE could be abbreviated as E; the block-oriented ELSE keyword cannot be abbreviated.

In a program where a block-oriented IF is nested within a line-oriented IF, the block-oriented IF requires an ELSE clause. The following is valid syntax:

```objectscript
IF x=1 WRITE "x is 1"
IF y=1 { WRITE "y is 1" }
ELSE { WRITE "y is not 1" }
ELSE WRITE "x is not 1"
```

The following is not valid syntax:

```objectscript
IF x=1 WRITE "x is 1"
IF y=1 { WRITE "y is 1" }
ELSE WRITE "x is not 1"
```

The block-oriented IF command does not read or set the value of the $TEST special variable. If a boolean test expression evaluates to TRUE, it executes the block of code within the curly braces, regardless of the value of $TEST.

**Arguments**

**expression1**

A test condition for the IF clause. It can take the form of a single expression or a comma-separated list of expressions. For an expression list, Caché evaluates the individual expressions in left to right order. It stops evaluation if it encounters an expression in the comma-separated list that evaluates to FALSE. If all expressions in the comma-separated list evaluate to TRUE, Caché executes the block of code associated with the IF clause. If any expression in the list evaluates to FALSE, Caché ignores any remaining expressions, and does not execute the block of code associated with the IF clause.

Commonly, expression1 is a boolean expression that evaluates to TRUE or FALSE (for example, x=7). Refer to the Operators and Expressions chapter of Using Caché ObjectScript. IF interprets a literal value as a boolean TRUE and FALSE as follows:

- **TRUE:** any non-zero numeric value, or a numeric string that evaluates to a non-zero numeric value. For example, 1, 7, -.007, "7-7", and "7dwarves".
- **FALSE:** a zero numeric value, or a string that evaluates to a zero numeric value. A non-numeric string evaluates to a zero numeric value. For example, 0, -0.00, 7-7, "0", "TRUE", "FALSE", "strike3", and the empty string ("").

For further details, refer to Strings as Numbers in the “Data Types and Values” chapter of Using Caché ObjectScript.

**expression2**

A test condition for an ELSEIF clause. It can take the form of a single expression or a comma-separated list of expressions. It is evaluated the same way as expression1.
IF with QUIT

If a QUIT command is encountered within an IF code block (or an ELSEIF code block or an ELSE code block) the QUIT behaves as a regular QUIT command, as if the code block did not exist. This behavior differs from a QUIT within any other type of curly brace code block (FOR, WHILE, DO...WHILE, TRY, or CATCH).

- If the IF code block is nested within a loop structure (such as a FOR code block), the QUIT exits the loop structure block and continues execution with the command that follows the loop structure code block.
- If the IF code block is within a TRY block or a CATCH block, the QUIT exits the TRY or CATCH block and continues execution with the command that follows the TRY or CATCH block.
- If the IF code block is not nested within a loop structure, or within a TRY or CATCH block, the QUIT exits the current routine.

Issuing a RETURN exits the current routine, whether or not it is issued from within a block structure.

The following example demonstrates the behavior of QUIT when the IF is not in a loop structure. The QUIT exits the routine:

```objectscript
SET y=$RANDOM(10)
IF y#2=0 {
  WRITE y," is even",!
  QUIT
  WRITE "never written"
}
ELSE {
  WRITE y," is odd",!
  QUIT
  WRITE "never written"
}
WRITE "QUIT out of the IF (never written)"
```

The following example demonstrates the behavior of QUIT when the IF is in a loop structure. The QUIT exits the FOR loop, then execution of the routine continues:

```objectscript
FOR x=1:1:8 {
  IF x#2=0 {
    WRITE x," is even",!
    QUIT:x=4
  }
  ELSE {
    WRITE x," is odd",!
    QUIT
  }
}
WRITE "QUIT out of the FOR loop (written)"
```

The following example demonstrates the behavior of QUIT when the IF is in a TRY block. The QUIT exits the TRY block, then execution of the routine continues with the next code after the CATCH block:

```objectscript
TRY {
  SET y=$RANDOM(10)
  IF y#2=0 {
    WRITE y," is even",!
    QUIT
  }
  ELSE {
    WRITE y," is odd",!
    QUIT
  }
  WRITE "QUIT out of the IF (never written)"
}
CATCH exp1 {
  WRITE "only written if an error occurred",!
  WRITE "Error Name: ",SZCVT(exp1.Name,"O","HTML"),!
}
TRY {
  WRITE "on to the next TRY block"
}
CATCH exp2 {
```
WRITE "Error Name: ",$ZCVT(exp2.Name,"O","HTML"),!
}

**IF with GOTO**

If a **GOTO** is encountered within an **IF** code block, program execution obeys that statement, with certain restrictions:

A **GOTO** statement can jump to a location outside of the **IF** command, or within the code block of the current clause. A **GOTO** statement cannot jump into another code block: neither a code block that belongs to another clause of the current **IF** command, nor a code block that belongs to another **IF**, **FOR**, **DO WHILE**, or **WHILE** command.

**Example**

In the following example, the **IF** command is used to categorize responders into one of three groups and invokes the appropriate subroutine. The three groups are females aged 44 or less, males aged 44 or less, and either females or males from age 45 through 120. In this example, the sex test expressions use the Contains operator ( \[ ) . (See **Operators** in *Using Caché ObjectScript.*)

```objectscript
Mainloop
NEW sex,age
READ !,"What is your sex? (M or F): ",!,sex QUIT:sex=""
READ !,"What is your age? ",!,age QUIT:age=""
IF "Ff"[sex,age<45 {
   DO SubA(age)
} ELSEIF "Mm"[sex,age<45 {
   DO SubB(age)
} ELSEIF "FfMm"[sex,age>44,age<125 {
   DO SubC(age)
} ELSE {
   WRITE !,"Invalid data value input"
} SubA(y)
   WRITE !,"Young woman ",y," years old"
SubB(y)
   WRITE !,"Young man ",y," years old"
SubC(y)
   WRITE !,"Older person ",y," years old"
```

**See Also**

- **DO WHILE** command
- **FOR** command
- **WHILE** command
- **GOTO** command
- **QUIT** command
- **$CASE** function
- **IF (legacy version)** command
### JOB

Runs a process in background.

```
JOB:pc jobargument,...
J:pc jobargument,...
```

where *jobargument* is one of the following:

**Local Jobs:**

- `routine(routine-params):(process-params):timeout`
- `routine(routine-params)[joblocation]:(process-params):timeout`
- `routine(routine-params)|joblocation|:(process-params):timeout`
- `##class(className).methodName(args):(process-params):timeout`
- `..methodName(args):(process-params):timeout`
- `$CLASSMETHOD(className,methodName,args):(process-params):timeout`

**Remote Jobs:**

- `routine[joblocation]`
- `routine|joblocation|`

### Arguments

<table>
<thead>
<tr>
<th><strong>pc</strong></th>
<th>Optional — A postconditional expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>routine</strong></td>
<td>The routine to be executed by the process created by <strong>JOB</strong>.</td>
</tr>
<tr>
<td><strong>routine-params</strong></td>
<td>Optional — A comma-separated list of parameters to pass to the routine. These parameters can be values, expressions, or existing local variable names. If specified, the enclosing parentheses are required. Routine parameters can only be passed to local jobs.</td>
</tr>
<tr>
<td><strong>className.methodName(args)</strong></td>
<td>The class method to be executed by the process created by <strong>JOB</strong>. The <em>className</em> cannot be $SYSTEM; it can be %SYSTEM. If you specify .. in place of <em>className</em>, <strong>JOB</strong> uses the current class context (the $THIS class). A comma-separated list of <em>args</em> arguments is optional; the enclosing parentheses are required. Omitted arguments are not permitted. For further details on using $CLASSMETHOD, refer to the section Dynamically Accessing Objects in the “Object-specific ObjectScript Features” chapter of Using Caché Objects.</td>
</tr>
<tr>
<td><strong>..methodName(args)</strong></td>
<td>Optional — A colon-separated list of positional parameters used to set various elements in the job’s environment. The <strong>process-params</strong> list is enclosed in parentheses and the parenthesized list preceded by a colon. All <strong>process-params</strong> are optional; the parentheses are required. To indicate a positional parameter is missing, its colon must be present, though trailing colons may be omitted. The <strong>process-params</strong> argument can only be specified for local jobs.</td>
</tr>
<tr>
<td><strong>process-params</strong></td>
<td>Optional — The number of seconds to wait for the jobbed process to start. Fractional seconds are truncated to the integer portion. The preceding colon is required. The <strong>timeout</strong> argument can only be specified for local jobs. If omitted, Caché waits indefinitely.</td>
</tr>
<tr>
<td><strong>timeout</strong></td>
<td>Optional — The number of seconds to wait for the jobbed process to start. Fractional seconds are truncated to the integer portion. The preceding colon is required. The <strong>timeout</strong> argument can only be specified for local jobs. If omitted, Caché waits indefinitely.</td>
</tr>
</tbody>
</table>
joblocation

Optional — An explicit or implied namespace used to specify the system and directory on which to run a local or remote job. An implied namespace is a directory path preceded by two caret characters: "^^dir". Enclose joblocation in either square brackets or vertical bars.

You cannot specify a joblocation when jobbing a class method. If joblocation specifies a remote system, you cannot specify routine-params, process-params, or timeout.

If joblocation specifies a local job, you cannot specify the first process parameter (nspace) because this would conflict with the joblocation parameter. Therefore, only the second, third, and fourth process parameters can be specified, and the missing nspace parameter must be indicated by a colon.

Description

JOB creates a separate process known as a job, jobbed process, or background job. The created process runs in the background, independently of the current process, usually without user interaction. A jobbed process inherits its configuration environment from the invoking process, except what is explicitly specified in the JOB command. For example, a jobbed process inherits the locale settings of the parent process, not the system default locale.

By contrast, a routine invoked with the DO command runs in the foreground as part of the current process.

JOB can create a local process on your local system, or it can invoke the creation of a remote process on another system. For more on remote jobs, see Remote Jobs.

When a job begins, Caché can call a user-written JOB^%ZSTART routine. When a job ends, Caché can call a user-written JOB^%ZSTOP routine. These entry points can be used for maintaining a log of job activity and troubleshooting problems encountered. For further details, see the section on “Using the Caché ^%ZSTART and ^%ZSTOP Routines” in Caché Specialized System Tools and Utilities.

Arguments

pc

An optional postconditional expression. Caché executes JOB if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

routine

The process to be started. It can take any of the following forms:

<table>
<thead>
<tr>
<th>Process Specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>Specifies a line label within the current routine.</td>
</tr>
<tr>
<td>^routine</td>
<td>Specifies a routine that resides on disk. Caché loads the routine from disk and starts execution at the first line of executable code within the routine.</td>
</tr>
<tr>
<td>label^routine</td>
<td>Specifies a line label within the named routine, which resides on disk. Caché loads the routine from disk and starts execution at the indicated label.</td>
</tr>
</tbody>
</table>
### Process Specification

<table>
<thead>
<tr>
<th>Specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label+offset</td>
<td>Specifies an offset of a specified number of lines from the line label. Use of offsets can cause problems with program maintenance, and is discouraged. You cannot specify an offset when calling a CACHESYS % routine. If you attempt to do so, Caché issues a &lt;NOLINE&gt; error.</td>
</tr>
<tr>
<td>label+offset+routine</td>
<td></td>
</tr>
</tbody>
</table>

If you are within a procedure block, calling the **JOB** command starts a child process that is outside the scope of the procedure block. It therefore cannot resolve a `label` reference within the procedure block. Therefore, for the **JOB** to reference a label within a procedure, the procedure cannot use a procedure block.

If you specify a nonexistent label, Caché issues a <NOLINE> error. If you specify a nonexistent routine, Caché issues a <NOROUTINE> error. For further details on these errors, refer to the `$ZERROR` special variable.

### routine-params

A comma-separated list of values, expressions, or existing local variable names. The enclosing parentheses are required. This list is known as the *actual parameter list*. The routine must have a *formal parameter list* with the same or a greater number of parameters. If you specify extra actual parameters, Caché issues a <PARAMETER> error. Note that if the invoked routine contains a formal parameter list, you must specify the enclosing parentheses, even if you do not pass any parameters. You cannot omit items from the list of `routine-params`.

You can pass routine parameters only by value, which means that you cannot pass arrays. This is different from the **DO** command where you can pass parameters by value and by reference. A special consequence of this restriction is you cannot pass arrays with the **JOB** command, since they are passed only by reference.

You cannot pass an object reference (oref) to a jobbed process. This is because a reference to an object existing in the invoking process context cannot be referenced in the new jobbed process context. Attempting to pass an oref results in passing an empty string ("") to the jobbed process.

When the routine starts, Caché evaluates any expressions and maps the value of each parameter in the actual list, by position, to the corresponding variable in the formal list. If there are more variables in the formal list than there are parameters in the actual list, Caché leaves the extra variables undefined.

Routine parameters can only be passed to local processes. You cannot specify routine parameters when creating a remote job. See Remote Jobs.

### process-params

A colon-separated list of positional parameters used to set various elements in the job’s environment. The preceding colon and enclosing parentheses are required. All of the positional parameters are optional. You can specify up to seven positional parameters for `process-params`. These six parameters are:

```
```

Since the parameters are positional, you must specify them in the order shown. If you omit a parameter that precedes a specified parameter, you must include a colon as a placeholder for it.

Process parameters cannot be specified for a remote job.

The following table describes the process parameters:
<table>
<thead>
<tr>
<th>Process Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nspace</code></td>
<td>The default namespace of the process. The specified <em>routine</em> is drawn from this namespace. If you omit <code>nspace</code>, your current default namespace is the default namespace of the jobbing process. An invalid namespace may prevent the job from starting. A local job cannot specify both the <em>joblocation</em> argument and namespace as a process parameter; you must omit the <code>nspace</code> process parameter, retaining the placeholder colon.</td>
</tr>
<tr>
<td><code>switch</code></td>
<td>An integer consisting of the sum of one or more of the following values: An integer bit mask that can represent zero or more of the following flags: 1 — Pass the symbol table to the spawned job. 2 — Do not use a JOB Server. 4 — Pass an open TCP/IP socket to the spawned job using the principal I/O device ($PRINCIPAL). (Deprecated, use 16 instead. See below.) 8 — Establish the process-specific window for two-digit years of the spawned job to be the system-wide default sliding window definition. Otherwise, the spawned job inherits the sliding window definition of the process issuing the <code>JOB</code> command. 16 — Pass an open TCP/IP socket to the spawned job using current I/O device ($IO). 128 through 16384 (in multiples of 32) — An additional integer value that specifies a partition size (in kilobytes) for the JOBbed child process. See “Specifying Child Process Partition Size” for more information. The <code>switch</code> value can be the sum of any combination of these integers. For example, a <code>switch</code> value of 13 (1+4+8) passes the symbol table (1), passes the open TCP/IP socket (4), and establishes a process-specific window for two-digit years that is the system-wide default (8).</td>
</tr>
<tr>
<td><code>principal-input</code></td>
<td>Principal input device for the process. <em>UNIX</em>: See <code>principal-output</code> for default.</td>
</tr>
<tr>
<td><code>principal-output</code></td>
<td>Principal output device for the process. The default is the device you specify for <code>principal-input</code>. <em>UNIX</em>: If you do not specify either device, the process uses the default principal device for processes started with the <code>JOB</code> command, which is <code>/dev/null</code>.</td>
</tr>
<tr>
<td><code>priority</code></td>
<td><em>UNIX</em>: An integer that specifies the priority for the child process (subject to operating system constraints). If not specified, the child process takes the parent process’ base priority plus the system-defined job priority modifier. You can use the <code>$VIEW</code> function to determine the current priority of a job. Windows has a Normal priority of 7. <em>UNIX</em>: priority ranges between -20 and 20, with 0 as Normal priority. In <em>UNIX</em>, a process cannot give itself an increased priority unless running as root.</td>
</tr>
<tr>
<td><code>os-directory</code></td>
<td>An operating system working directory for file I/O. The default is to use the working directory inherited from the parent process. This parameter may be ignored on some systems.</td>
</tr>
</tbody>
</table>
Switch 4 and Switch 16

The use of `switch=4` is discouraged, because this establishes the passed TCP device as the principal device of the child job. In this case, the child job could halt when it detected the TCP remote connection dropped and would not perform error trapping. Instead, users should use `switch=16`, and then in the child job use the `%SYSTEM.INetInfo.TCPName()` method to get the passed TCP device name. In this case, the child job could continue to run when it detected the TCP remote connection dropped, because the principal device of the child job is not the passed TCP device.

**timeout**

The number of seconds to wait for the jobbed process to start before timing out and aborting the job. The preceding colon is required. You must specify `timeout` as an integer value or expression. If a jobbed process times out, Caché aborts the process and sets the `$TEST` special variable to 0 (FALSE). Execution then proceeds to the next command in the calling routine; no error message is issued. If a jobbed process succeeds, Caché sets `$TEST` to 1 (TRUE). Note that `$TEST` can also be set by the user, or by a `LOCK`, `OPEN`, or `READ` timeout.

Timeout can only be specified for a local process.

**Examples**

The following example starts the monitor routine in the background. If the process does not start in 20 seconds, Caché sets `$TEST` to FALSE (0).

```
JOB ^monitor::20
WRITE $TEST
```

The following example starts execution of the monitor routine at the line label named Disp.

```
JOB Disp^monitor
```

The following example starts the Add routine, passing it the value in variable `num1`, the value 8, and the value resulting from the expression `a+2`. The Add routine must contain a formal parameter list that includes at least three parameters.

```
JOB ^Add(num1,8,a+2)
```

The following example starts the Add routine, which has a formal parameter list, but passes no parameters. In this case, the Add routine must include code to assign default values to its formal parameters, since they receive no values from the calling routine.

```
JOB ^Add()
```

The following example creates a process running your current routine at label AA. The process parameters pass your current symbol table to the routine. It can use a JOB Server.

```
JOB AA:("":1)
```

This following example passes the routine parameters `VAL1` and the string "DR," to the routine `^PROG`, starting at entry point ABC, in the current namespace. The routine expects two arguments. Caché does not pass the current symbol table to this job, it will use a JOB Server if possible, and use tta5: as principal input and output device.

```
JOB ABC^PROG(VAL1,"DR."):(:0:"tta5:")
```

The following examples show the jobbing of a class method, with a timeout of ten seconds. They use tta5: as principal input and output device.

The following example uses `##class` syntax to invoke a class method:

```
JOB ##class(MyClass).Run():(:0:"tta5:"):10
```

The following example uses the `$CLASSMETHOD` function to invoke a class method:

```
JOB $CLASSMETHOD(MyClass,Run):(:0:"tta5:"):10
```
$CLASSMETHOD parameters must be passed by value, not by reference, when using JOB $CLASSMETHOD.

The following example uses relative dot syntax (..) to refer to a method of the current object:

```objectscript
JOB ..CleanUp()::0:0:tta5:*):10
```

or simply:

```objectscript
JOB ..CleanUp()::10
```

For further details, refer to “Object-specific ObjectScript Features” in Using Caché Objects.

**Notes**

**Caché Assigns Job Numbers and Memory Partitions**

After you start a jobbed process, Caché allocates a separate memory partition for it and assigns it a unique job number (also referred to as a Process ID or PID). The job number is stored in the $JOB special variable. The status of the job (including whether or not it was started by a JOB command) is stored in the $ZJOB special variable.

Since jobbed processes have separate memory partitions, they do not share a common local variable environment with the process that created them or with each other. When you start a jobbed process, you can use parameter passing (routine-params) to pass values from the current process to the jobbed process.

If the JOB command fails, it is usually because:

- There are no free partitions.
- There is not enough memory to create a partition with the characteristics specified by process-params.

**Jobbed Process Permissions are Platform-dependent**

Processes created by the JOB command run as the Caché service account user. This means that you must insure that the Caché service account has explicit permissions to access all necessary resources.

A spawned job process may run under a different userid than that of the process that issued the JOB command. The userid of the spawned job process depends on the platform:

- On Windows platforms, the job process uses the userid established for the Caché instance.
- On UNIX® platforms, the job process uses the userid of the process that issued the JOB command.

Thus, when you spawn a job, you must make sure that the userid for the job process has the necessary permissions to use any files read or written during the job execution.

**Communicating Between Jobs**

Parameter passing by value can occur in only one direction and only at job start up. For processes to communicate with each other, they must use mutually agreed upon global variables. Such variables are commonly known as scratch globals because their sole purpose is to allow processes to exchange information among themselves.

- Processes can use the %SYSTEM.Event class methods to communicate between jobs.
- You can pass all local variables in the current process to the invoked process by specifying a special process parameter.
- Processes can communicate between jobs through the IPC (Interprocess Communication) devices (device numbers 224 through 255) or, on UNIX® operating systems, through UNIX® pipes.

**Establishing Device Ownership**

Caché assumes that the invoked routine includes code (that is, OPEN and USE commands) to handle device ownership for the new process. The default device is the null device.
Caché does not assign a default device to any process other than the process started at sign in.

**Setting Job Priority**

The %PRIO utility allows you to control the priority at which a UNIX® jobbed process runs. The available options are NORMAL (uses load balancing to adjust CPU usage), LOW, and HIGH. A jobbed process with a priority of HIGH competes on an equal basis with interactive processes for CPU resources.

Caché also allows you to establish default priorities for jobbed processes.

You can use the SetBatch() method to establish a process as executing in batch mode. A batch mode process has a lower priority than a non-batch process.

**Using the JOB Command in a Raw Partition (UNIX®)**

You can use the JOB command in a raw partition in either of two ways:

- Issue the JOB command while in the raw partition.
- Issue the JOB command while in another namespace, and specify the raw partition as the nspace process parameter of the JOB command. Here nspace is an implied namespace. An implied namespace is a directory path preceded by two caret characters: "^^dir". Implied namespace syntax is described in Global Structure in Using Caché Globals.

Commands and jobbed processes running in a raw partition must always specify the full pathname when making references to filenames, and must not use any pathname that starts with "." or "..", as these are special UNIX® files and are not present in a raw partition. Violating either of these rules causes a <DIRECTORY> error.

To obtain the full pathname of the current namespace, you can invoke the NormalizeDirectory() method, as shown in the following example:

```
WRITE "#class(%Library.File).NormalizeDirectory(""
```

Alternatively, you can use the UNIX® batch command & instead of the ObjectScript JOB command to start a routine as a batch job in a raw disk partition.

**Remote Jobs**

Before starting a remote job, you must establish an ECP connection and set the netjob parameter to true. This enables the server to handle job requests from remote ECP client systems.

You must configure the ability to receive remote job requests on any system that will receive them.

On the receiving system, go to the Management Portal, select System Administration, Configuration, Additional Settings, Advanced Memory. Locate netjob to view and edit. When “true”, incoming remote job requests via ECP will be honored on this server. The default is “true”.

The license on the remote system must support enough users to run remotely initiated jobs. You can determine the number of available Caché licenses using class methods of the %SYSTEM.License class, as described in the InterSystems Class Reference.

**JOB Syntax for Remote Job Request**

You can send a remote job from one Caché system to another using the following syntax:

```
JOB routine[joblocation]
JOB routine|joblocation|
```

The two forms are equivalent; you can use either square brackets or vertical bars to enclose the joblocation parameter. A remote job cannot pass routine parameters, process parameters, or a timeout.
**joblocation**
A specification of the location of the job. The enclosing square brackets or vertical bars are required.

The action Caché takes depends on the job location syntax you are using.

<table>
<thead>
<tr>
<th><strong>joblocation Syntax</strong></th>
<th><strong>Result</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>[&quot;namespace&quot;]</td>
<td>Caché checks whether this explicit namespace has its default dataset on the local system or on a remote system. If the default dataset is on the local system, the system starts the job using the parameters you specify. If the default dataset is on a remote system, the system starts the remote job in the directory of the namespace’s default dataset.</td>
</tr>
<tr>
<td>[&quot;dir&quot;,&quot;sys&quot;]</td>
<td>Caché converts this location to the implied namespace form [&quot;^sys^dir&quot;].</td>
</tr>
<tr>
<td>[&quot;^sys^dir&quot;]</td>
<td>The job runs in the specified directory on the specified remote system. Caché does not allow any routine parameters, process parameters, or timeout specification.</td>
</tr>
<tr>
<td>[&quot;^^dir&quot;]</td>
<td>The job runs in the specified directory (implied namespace) as a local job on the current system using the parameters you specify. An implied namespace is a directory path preceded by two caret characters: &quot;^^dir&quot;.</td>
</tr>
<tr>
<td>[&quot;dir&quot;,&quot;&quot;]</td>
<td>Caché issues a &lt;COMMAND&gt; error.</td>
</tr>
</tbody>
</table>

**Global Mapping with Remote Jobs (Windows)**

Caché does not provide global mapping for remote jobs, whether or not global mapping has been defined on the requesting system. To avoid the lack of global mapping, use extended references with your global specifications that point to the location of any globals not in that namespace. If the namespace you specify in an extended reference is not defined on the system you specify, you receive a <NAMESPACE> error. Namespaces and the syntax for extended global references are described in Global Structure in Using Globals.

**Using the $ZCHILD and $ZPARENT Special Variables**

$ZPARENT contains the PID (Process ID) of the process which jobbed the current process, or 0 if the current process was not created through the JOB command.

$ZCHILD contains the PID of the last process created by the JOB command, whether or not the attempt was successful.

By using $ZCHILD it is possible to determine the execution status of a remote job by comparing the $ZCHILD value before and after running the JOB command. If the before and after values are different, and the after value is nonzero, the after $ZCHILD value is the PID of the newly created remote job, indicating that the process was successfully created. If the after value is zero, or the after value is the same as the before value, the remote job was not created.

$ZCHILD can only tell you that a remote job was created; it does not tell you if the remote job ran successfully. The best way to determine if a remote process ran without error and ran to completion is to provide some sort of logging and error trapping in the code being run. The remote job mechanism does not inform the parent process in any way about remote process errors or remote process termination, successful or otherwise.

**Using JOB Servers**

JOB Servers are Caché processes that wait to process job requests. Jobbed processes that attach to JOB Servers avoid the added overhead of having to create a new process. Whenever a user issues a JOB command with the switch parameter set to use JOB Server if available, Caché checks to see if any JOB Servers are available to handle it. If not, it will create a process. If there is a free JOB Server, the job attaches to that JOB Server.

When a job halts while running in a JOB Server, the JOB Server hibernates until it receives another job request. A jobbed process not running in a JOB Server exits and the process is deleted.
There are some unavoidable differences between the JOB Server environment and the jobbed process environment, which may be a security concern with jobbed processes executing in JOB Servers. A jobbed process takes on the security attributes of the process that issued the JOB command at the Caché level.

**Input and Output Devices**

Only one process can own a device at a time. This means that a job executing in a JOB Server is unable to perform input or output to your principal I/O devices even though you may close device 0.

Therefore, if you expect a JOB Server to perform input, you must specify:

- An alternative input device for it
- The null device for an output device (if you do not want to see the output)

Failure to follow these guidelines may cause the job executing in the JOB Server to hang if it needs to do any input or output from/to your principal I/O devices. You may find that frequently job output does get through to your terminal (for example, if you have the SHARE privilege), but typically it will not.

**Troubleshooting Jobs That Will Not Execute**

If your job does not start, check your I/O specification. Your job will not start if Caché cannot open the devices you requested. Note that the null device (/dev/null on UNIX®) is always available.

If your job starts but then halts immediately, make sure you have sufficient swap space. Your job receives an error if you do not have enough swap space.

If your job does not start, make sure that you have used the correct namespace in the JOB command. You can test whether a namespace is defined by using the `Exists()` method:

```objectscript
WRITE #class(%SYS.Namespace).Exists("USER"),! ; an existing namespace
WRITE #class(%SYS.Namespace).Exists("LOSER")   ; a nonexistent namespace
```

If the JOB command is still not working, try the following:

- Execute the routine with the DO command.
- Make sure you are not exceeding the number of processes for which you are licensed in Caché. You can determine the number of available Caché licenses using class methods of the %SYSTEM.License class, as described in the *InterSystems Class Reference*.
- If there is a timeout parameter in your JOB command, check whether the speed of your system is using up the timeout period.

When many jobs are executing concurrently, the JOB command may hang while waiting for a routine buffer or a license slot. You can interrupt a JOB command by using Ctrl-C.

**JOB Command Completion**

A jobbed process continues to completion even if the process that created it logs off before that completion.

**JOB Command with TCP Devices**

You can use the JOB command to implement a TCP concurrent server. A TCP concurrent server allows multiple clients to be served simultaneously. In this mode a client does not have to wait for the server to finish serving other clients. Instead, each time a client requests the server, it spawns a separate subjob for that client which remains open as long as the client needs it. As soon as this subjob has been spawned (indicated by the return of the JOB command), another client may request service and the server will create a subjob for that client as well.
A concurrent server uses the **JOB** command with the `switch` process parameter bit mask bit 4 or bit 16 turned on, and passes to the spawned process the input and output process parameters.

- If you specify `switch` bit 4, you must specify the TCP device in both `principal–input` and `principal–output` process parameters. You must use the same device for both `principal–input` and `principal–output`, as follows:

  ```objectscript
  JOB child:(:4:tcp:tcp)
  ```

  The spawned process then sets this single I/O device, as follows:

  ```objectscript
  SET tcp=$IO
  ```

- If you specify `switch` bit 16, you can specify different devices for the TCP device, the `principal–input`, and the `principal–output` process parameters, as follows:

  ```objectscript
  USE tcp
  JOB child:(:16:input:output)
  ```

  USE `tcp` preceding the **JOB** command specifies the current device (rather than the principal device), as the TCP device. The spawned process can then set these devices, as follows:

  ```objectscript
  SET tcp=##class(%SYSTEM.INetInfo).TCPName()
  SET input=$PRINCIPAL
  SET output=$IO
  WRITE tcp," ",input," ",output
  ```

It is important to note that the **JOB** command will pass the TCP socket to the jobbed process if the 4 or 16 bit is set. This capability may be combined with other features of the **JOB** command by adding the appropriate bit code for each additional feature. For example, when `switch` includes the bit with value 1, the symbol table is passed. To turn on concurrency and pass the symbol table, `switch` would have a value of 5 (4+1) or 17 (16+1).

Before you issue the **JOB** command, the TCP device must:

- Be open
- Be listening on a TCP port
- Have accepted an incoming connection

After the **JOB** command, the device in the spawning process is still listening on the TCP port, but no longer has an active connection. The application should check the **$ZA** special variable after issuing the **JOB** command to make sure that the CONNECTED bit in the state of the TCP device was reset.

The spawned process starts at the designated entry point using the specified device(s) as TCP device, principal–input, and principal–output device. The TCP device has the same name in the child process as in the parent process. The TCP device has one attached socket. The **USE** command is used to establish the TCP device in “M” mode, which is equivalent to “PSTE”. The “P” (pad) option is needed to pad output with record terminator characters. When this mode is set, **WRITE !** sends LF (line feed) and **WRITE #** sends FF (form feed), in addition to flushing the write buffer.

The TCP device in the spawned process is in a connected state: the same state the device would receive after it is opened from a client. The spawned process can use the TCP device with an explicit USE statement. It can also use the TCP device implicitly.

The following example shows a very simple concurrent server that spawns off a child job whenever it detects a connection from a client. **JOB** uses a *switch* value of 17, consisting of the concurrent server bit 16 and the symbol table bit 1:

```objectscript
server ;
    SET io="|TCP|1"
    SET "serverport=7001"
    OPEN io:("serverport:""MA")#:200
    IF $TEST-0 {
        WRITE !,"Cannot open server port"
        QUIT
    }
    ELSE { WRITE !,"Server port opened" }
loop ;
    USE io READ x   ; Read for accept
    USE 0 WRITE !,"Accepted connection"
    USE io JOB child:(:17::) ; Concurrent server bit is on
    GOTO loop
child ;
    SET io=##class(%SYSTEM.INetInfo).TCPName()
    SET input=SPRINCIPAL
    SET output=$IO
    USE io:(::"M") ; Ensure that "M" mode is used
    WRITE $JOB,!  ; Send job id on TCP device to be read by client
    QUIT
client ;
    SET io="|TCP|2"
    SET host="127.0.0.1"
    OPEN io:(host:"serverport:""M")#:200 ; Connect to server
    IF $TEST-0 {
        WRITE !,"Cannot open connection"
        QUIT
    }
    ELSE { WRITE !,"Client connection opened" }
    USE io READ x#3:200  ; Reads from subjob
    IF x="" {
        USE 0
        WRITE !,"No message from child"
        CLOSE io
        QUIT
    }
    ELSE {  
        USE 0
        WRITE !,"Child is on job ",x
        CLOSE io
        QUIT
    }
```

The child uses the inherited TCP connection to pass its job ID (in this case assumed to be 3 characters) back to the client, after which the child process exits. The client opens up a connection with the server and reads the child’s job ID on the open connection. In this example, the IPv4 value "127.0.0.1" for the variable "host" indicates a loopback connection to the local host machine (the corresponding IPv6 loopback value is "0:0:0:0:0:0:0:1" or "::1"). You can set up a client on a different machine from the server if "host" is set to the server’s IP address or name. Further details on IPv4 and IPv6 formats can be found in the section “Use of IPv6 Addressing” in the chapter “Server Configuration Options” in the *Caché Programming Orientation Guide*.  

80 Caché ObjectScript Reference
In principle, the child and client can conduct extended communication, and multiple clients can be talking concurrently with their respective children of the server.

**Specifying Child Process Partition Size**

The partition size (\$ZSTORAGE size) for a background job is determined as follows:

- If job servers are active, for any job server process the job partition size will always be 262144 (in kilobytes).

- If job servers are inactive, the job partition size defaults to the default partition size for non-background jobs (bbsiz). This is the system-wide default in effect when you execute \texttt{JOB}, regardless of the partition size of the parent process from which you issue \texttt{JOB}.

If job servers are enabled, but all of them are active and you start a new job, that job process will not be a job server process, so its \$ZSTORAGE value will default to bbsiz. This is also true when job servers are active, but due to loading the job is run in a non-job-server process.

If job servers are inactive, you can specify the partition size of the jobbed child process in the \texttt{JOB} statement. You specify the partition size (in kilobytes) of the jobbed process in the second process parameter of the \texttt{JOB} command. For example, \texttt{JOB ^myroutine:(:8192)}. The value you specify must be a multiple of 32 and must range from 128 through 16384. It also cannot exceed the default partition size; it can only be used to specify a lower value than the default.

You can optionally specify the partition-size process parameter value in combination with other process information that you would normally put in the second process parameter of \texttt{JOB}. Consider the following \texttt{JOB} command: \texttt{JOB ^myroutine:(:544+1)}. This command specifies that the symbol table of the jobbing process should be passed to the jobbed process and that the jobbed process should have a partition size of 544K. Although you can specify this second parameter, which passes two values (544 and 1) as 545, 544 + 1 is clearer and has exactly the same effect.

Note that a job itself can programmatically set its own partition size using \texttt{SET $ZSTORAGE}.

**See Also**

- ^$JOB structured system variable
- \$JOB special variable
- \$TEST special variable
- \$ZJOB special variable
- \$ZCHILD special variable
- \$ZPARENT special variable
- \$ZF(-1) function
- \$ZF(-2) function
- TCP Client/Server Communication in \textit{Caché I/O Device Guide}
- \textit{The Spool Device} in \textit{Caché I/O Device Guide}
- Using the Management Portal to monitor and control processes in \textit{Managing Caché} in \textit{Caché System Administration Guide}
KILL

Deletes variables.

KILL: pc killargument,...
K: pc killargument,...

where killargument can be:

variable,...
(variable,...)

Arguments

<table>
<thead>
<tr>
<th>pc</th>
<th>Optional — A postconditional expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>Optional — A variable name or comma-separated list of variable names. Without parentheses: the variable(s) to be deleted. With parentheses: the variable(s) to be kept.</td>
</tr>
</tbody>
</table>

Description

There are three forms of the KILL command:

• KILL without an argument, known as an argumentless KILL.
• KILL with a variable list, known as an inclusive KILL.
• KILL with a variable list enclosed in parentheses, known as an exclusive KILL.

The KILL command without an argument deletes all local variables. It does not delete process-private globals, globals, or user-defined special variables.

KILL with a variable or comma-separated variable list as an argument:

KILL variable,...

is called an inclusive KILL. It deletes only the variable(s) you specify in the argument. Killing a variable kills all subscripts of that variable at all lower levels than the specified variable. 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.

KILL with a variable or comma-separated variable list enclosed in parentheses as an argument:

KILL (variable,...)

is called an exclusive KILL. It deletes all local variables except those you specify in the argument. The variables you specify can only be local variables. You cannot specify a subscripted variable; specifying a local variable preserves the variable and all of its subscripts. The local variables you specify do not have to be actual defined variables, but they must be valid local variable names.

Note: KILL can delete local variables created by Caché objects. Therefore, do not use either argumentless KILL or exclusive KILL in any context where they might affect system structures (such as %objTX currently used in %Save) or system objects (such as the stored procedure context object). In most programming contexts, these forms of KILL should be avoided.

You can use the $DATA function to determine whether a variable is defined or undefined, and whether a defined variable has subscripts. Killing a variable changes its $DATA status to undefined.
Using **KILL** to delete variables frees up local variable storage space. To determine or set the maximum storage space (in kilobytes) for the current process, use the $ZSTORAGE special variable. To determine the available storage space (in bytes) for the current process, use the $STORAGE special variable.

**Arguments**

**pc**

An optional postconditional expression. Caché executes the **KILL** command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

**variable**

If not enclosed in parentheses: the variable(s) to be deleted by the **KILL** command. **variable** can be a single variable or a comma-separated list of variables.

If enclosed in parentheses: the local variable(s) to be kept by the **KILL** command; the **KILL** command deletes all other local variables. **variable** can be a single variable or a comma-separated list of variables.

**Examples**

In the following example, an inclusive **KILL** deletes local variables *a*, *b*, and *d*, and the deletes the process-private global ^||ppglob and all of its subscripts. No other variables are affected:

```objectscript
SET ^||ppglob(1)="fruit"
SET ^||ppglob(1,1)="apples"
SET ^||ppglob(1,2)="oranges"
SET a=1,b=2,c=3,d=4,e=5
KILL a,b,d,"^||ppglob"
WRITE "a=",$DATA(a)," b=",$DATA(b)," c=",$DATA(c)," d=",$DATA(d)," e=",$DATA(e),!
WRITE " ^||ppglob(1)=",$DATA("^||ppglob(1)),
 " ^||ppglob(1,1)=",$DATA("^||ppglob(1,1)),
 " ^||ppglob(1,2)=",$DATA("^||ppglob(1,2))
```

In the following example, an inclusive **KILL** deletes the local variable *a*(1) and its subscripts *a*(1,1), *a*(1,2) and *a*(1,1,1); it does not delete the local variables *a*, *a*(2), or *a*(2,1):

```objectscript
SET a="food",a(1)="fruit",a(2)="vegetables"
SET a(1,1)="apple",a(1,1,1)="mackintosh",a(1,2)="banana"
SET a(2,1)="artichoke"
WRITE "before KILL: ",!
WRITE $DATA(a)," ",$DATA(a(1))," ",$DATA(a(1,1))," ",$DATA(a(1,1,1))," ",
 $DATA(a(1,2))," ",$DATA(a(2,1)),!
KILL a(1)
WRITE "after KILL: ",!
WRITE $DATA(a)," ",$DATA(a(1))," ",$DATA(a(1,1))," ",$DATA(a(1,1,1))," ",
 $DATA(a(1,2))," ",$DATA(a(2,1))
```

In the following example, an exclusive **KILL** deletes all local variables except for variables *d* and *e*:

```objectscript
SET a=1,b=2,c=3,d=4,e=5
KILL (d,e)
WRITE "a=",$DATA(a)," b=",$DATA(b)," c=",$DATA(c)," d=",$DATA(d)," e=",$DATA(e)
```

Note that because an exclusive **KILL** deletes object variables, the above program works from a terminal session, but does not work within an object method.

The following example, an inclusive **KILL** deletes two process-private globals and an exclusive **KILL** deletes all local variables except for variables *d* and *e*.

```objectscript
SET ^|a="one",^|b="two",^|c="three"
SET a=1,b=2,c=3,d=4,e=5
KILL (a,z) "|c,(d,e)
WRITE "a=",$DATA(a)," b=",$DATA(b)," c=",$DATA(c)," d=",$DATA(d)," e=",$DATA(e),!
WRITE " ^|a=",$DATA(^|a)," ^|b=",$DATA(^|b)," ^|c=",$DATA(^|c)
```
Notes

**KILL and Objects**

Object instance references (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.

For information on OREFs, see “OREF Basics” in *Using Caché Objects*.

Prior to Caché version 5.0, the `%Close()` method could be used to remove objects from memory. With Caché version 5.0 and subsequent, the `%Close()` method performs no operation and always completes successfully. Do not use `%Close()` when writing new code.

**KILL Using an Object Method**

You can specify an object method on the left side of a `KILL` expression. The following example specifies the `%Get()` method:

```
SET obj=##class(test).%New()  // Where test is class with a multidimensional property md
SET myarray=[{obj}]
SET index=0, subscript=2
SET myarray.%Get(index).md(subscript)="value"
WRITE $DATA(myarray.%Get(index).md(subscript)),!
```

### Inclusive KILL

An inclusive `KILL` deletes only those variables explicitly named. The list can include local variables, process-private globals, and globals—either subscripted or unsubscripted. The inclusive `KILL` is the only way to delete global variables.

You can delete a range of subscripted global variables by using the `KillRange()` method.

### Exclusive KILL

An exclusive `KILL` deletes all local variables *except* those that you explicitly name. Listed names are separated by commas. The enclosing parentheses are required.

The exception list can contain only local unsubscripted variable names. For example, if you have a local variable array named fruitbasket, which has several subscript nodes, you can preserve the entire local variable array by specifying `KILL (fruitbasket)`: you cannot use exclusive `KILL` to selectively preserve individual subscript nodes. An exclusive kill list cannot specify a process-private global, a global, or a special variable; attempting to do so results in a `<SYNTAX>` error. Local variables not named in the exception list are deleted; subsequent references to such variables generate an `<UNDEFINED>` error. The exclusive `KILL` has no effect on process-private globals, globals, and special variables. However, it does delete local variables created by system objects.

### Using KILL with Arrays

You can use an inclusive `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.

- To delete a local variable array, use any form of `KILL`.
- To delete a selected node within a local variable array, you must use an inclusive `KILL`.
- To delete a global variable array, you must use an inclusive `KILL`.
To delete a selected node within a global variable array, you must use an inclusive `KILL`.

For further details on global variables with subscripted nodes, see **Global Structure** in *Using Caché Globals*.

To delete an array, simply supply its name to an inclusive `KILL`. For example, the following command deletes global array `^fruitbasket` and all of its subordinate nodes.

```
SET ^fruitbasket(1)="fruit"
SET ^fruitbasket(1,1)="apples"
SET ^fruitbasket(1,2)="oranges"
WRITE "Before KILL:\n",
WRITE ^fruitbasket(1)," contains ",^fruitbasket(1,1),
" and ",^fruitbasket(1,2),!
KILL ^fruitbasket
WRITE "After KILL:\n",
WRITE ^fruitbasket(1)," contains ",^fruitbasket(1,1),
" and ",^fruitbasket(1,2),!
```

To delete an array node, supply the appropriate subscript. For example, the following `KILL` command deletes the node at subscript 1,2.

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

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.

To delete multiple local variable arrays, you can use either the inclusive form or exclusive form of `KILL`, as described above. For example, the following command removes all local arrays except array1 and array2.

```
KILL (array1,array2)
```

To delete multiple array nodes, you can use only the inclusive form of `KILL`. For example, the following command removes the three specified nodes, deleting one node from each array.

```
KILL array1(2,4),array2(3,2),array3{1,7}
```

The nodes can be in the same or different arrays.

You may delete a specified local or global array node by using the `ZKILL` command. Unlike `KILL`, `ZKILL` does not delete all nodes subordinate to the specified node.

### Effects of KILL with Parameter Passing

With parameter passing, values are passed to a user-defined function or to a subroutine called with the `DO` command. The values to be passed to the user-defined function or subroutine are supplied in a comma-separated list called the **actual parameter list**. Each value supplied is mapped, by position, into a corresponding variable in the **formal parameter list** defined for the user-defined function or subroutine.

Depending on how the actual parameter list is specified, parameter passing can occur in either of two ways: **by value** or **by reference**. For more information on these two types of parameter passing, see **Parameter Passing** in *Using Caché ObjectScript*. 

---

**Caché ObjectScript Reference** 85
Killing a variable in the formal parameter list has different results depending on whether passing by value or passing by reference is in effect.

If you are passing a variable by value:

- Killing a variable in the formal list has no effect outside the context of the invoked function or subroutine. This is because Caché automatically saves the current value of the corresponding actual variable when the function or subroutine is invoked. It then automatically restores the saved value on exit from the function or subroutine.

In the following passing by value example, the `KILL` in `Subrt1` deletes the formal variable `x` but does not affect the actual variable `a`:

```
Test
SET a=17
WRITE !,"Before Subrt1 a: ",$DATA(a)
DO Subrt1(a)
WRITE !,"After Subrt1 a: ",$DATA(a)
QUIT
Subrt1(x)
WRITE !,"pre-kill x: ",$DATA(x)
KILL x
WRITE !,"post-kill x: ",$DATA(x)
QUIT
```

If you are passing a variable by reference:

- Performing `KILL` and including the variable in the formal list also kills the corresponding actual variable. When the function or subroutine terminates, the actual variable will no longer exist.
- Performing a `KILL` and excluding the variable in the formal list causes both the formal variable and the actual variable passed by reference to be preserved.

In the following passing by reference example, the `KILL` in `Subrt1` deletes both the formal variable `x` and the actual variable `a`:

```
Test
SET a=17
WRITE !,"Before Subrt1 a: ",$DATA(a)
DO Subrt1(.a)
WRITE !,"After Subrt1 a: ",$DATA(a)
QUIT
Subrt1(&x)
WRITE !,"pre-kill x: ",$DATA(x)
KILL x
WRITE !,"post-kill x: ",$DATA(x)
QUIT
```

As a general rule, you should not `KILL` variables specified in a formal parameter list. When Caché encounters a function or subroutine that uses parameter passing (whether by value or by reference), it implicitly executes a `NEW` command for each variable in the formal list. When it exits from the function or subroutine, it implicitly executes a `KILL` command for each variable in the formal list. In the case of a formal variable that uses passing by reference, it updates the corresponding actual variable (to reflect changes made to the formal variable) before executing the `KILL`.

**Transaction Processing**

A `KILL` of a global variable is journaled as part of the current transaction; this global variable deletion is rolled back during transaction rollback. A `KILL` of a local variable or a process-private global variable is *not* journaled, and thus this variable deletion is unaffected by a transaction rollback.

**See Also**

- `ZKILL` command
- `$DATA` function
- `$STORAGE` special variable
LOCK

Enables a process to apply and release locks to control access to data resources.

LOCK:pc
L:pc

LOCK:pc +lockname#locktype:timeout,...
L:pc +lockname#locktype:timeout,...

LOCK:pc +(lockname#locktype,...):timeout,...
L:pc +(lockname#locktype,...):timeout,...

Arguments

| pc    | Optional — A postconditional expression. |
| +     | Optional — The lock operation indicator (a + character, – character, or no character) to apply or remove a lock. A + (plus sign) applies the specified lock(s) without unlocking any prior locks. This can be used to apply an incremental lock. A – (minus sign) unlocks (or decrements) a lock. If you omit the lock operation indicator (no character), Caché unlocks all prior locks and applies the specified lock(s). |
| –     | |

| lockname | A lock name associated with the resource(s) to be locked or unlocked. Must be a valid identifier, following the same naming conventions as local variables or globals. |

| #locktype | Optional — A letter code specifying the type of lock to lock or unlock, specified in quotation marks. Available values are “S” (shared lock), “E” (escalating lock), “I” (immediate unlock), and “D” (deferred unlock). When specifying, the preceding # symbol is mandatory. For example, #"S". You can specify more than one letter code. For example, #"SEI". “S” and “E” are specified for both locking and unlocking operations; “I” and “D” are only specified for unlocking operations. If omitted, the lock type defaults to an exclusive lock (non-S) that does not escalate (non-E) and that always defers releasing an unlocked lock to the end of the current transaction (non-I / non-D). |

| :timeout | Optional — The time to wait before the attempted lock operation times out. Can be specified with or without the optional #locktype. When specifying, the preceding : symbol is mandatory. For example, LOCK "a(1):10 or LOCK "a(1)#"E":10. Specify timeout as an integer number of seconds. A value of 0 means to make one attempt, then time out. Fractional seconds are truncated to the integer portion. If omitted, Caché waits indefinitely. |

Description

There are two basic forms of the LOCK command:

- Without arguments
- With arguments

**LOCK without Arguments**

The argumentless LOCK releases (unlocks) all locks currently held by the process in all namespaces. This includes exclusive and shared locks, both local and global. It also includes all accumulated incremental locks. For example, if there are three incremental locks on a given lock name, Caché releases all three locks and removes the lock name entry from the lock table.
If you issue an argumentless **LOCK** during a transaction, Caché places all locks currently held by the process in a Delock state until the end of the transaction. When the transaction ends, Caché releases the locks and removes the corresponding lock name entries from the lock table.

The following example applies various locks during a transaction, then issues an argumentless **LOCK** to release all of these locks. The locks are placed in a Delock state until the end of the transaction. The **HANG** commands give you time to check the lock’s ModeCount in the Lock Table:

```
TSTART
LOCK +^a(1)      // ModeCount: Exclusive
HANG 2
LOCK +^a(1)#"E"  // ModeCount: Exclusive/1+1e
HANG 2
LOCK +^a(1)#"S"  // ModeCount: Exclusive/1+1e,Shared
HANG 2
LOCK             // ModeCount: Exclusive/1+1e->Delock,Shared->Delock
HANG 10
TCOMMIT          // ModeCount: locks removed from table
```

Argumentless **LOCK** releases all locks held by the process without applying any locks. Completion of a process also releases all locks held by that process.

**LOCK with Arguments**

**LOCK** with arguments specifies one or more lock names on which to perform locking and unlocking operations. What lock operation Caché performs depends on the **lock operation indicator** argument you use:

- **LOCK lockname** unlocks all locks previously held by the process in all namespaces, then applies a lock on the specified lock name(s).
- **LOCK +lockname** applies a lock on the specified lock name(s) without unlocking any previous locks. This allows you to accumulate different locks, and allows you to apply incremental locks to the same lock.
- **LOCK –lockname** performs an unlock operation on the specified lock name(s). Unlocking decrements the lock count for the specified lock name; when this lock count decrements to zero, the lock is released.

A lock operation may immediately apply the lock, or it may place the lock request on a wait queue pending the release of a conflicting lock by another process. A waiting lock request may time out (if you specify a **timeout**) or may wait indefinitely (until the end of the process).

**LOCK with Multiple Lock Names**

You can specify multiple locks with a single **LOCK** command in either of two ways:

- **Without Parentheses**: By specifying multiple lock arguments without parentheses as a comma-separated list, you can specify multiple independent lock operations, each of which can have its own **timeout**. (This is functionally identical to specifying a separate **LOCK** command for each lock argument.) Lock operations are performed in strict left-to-right order. For example:

  ```
  LOCK var1(1):10,+var2(1):15
  ```

  Multiple lock arguments without parentheses each can have their own **lock operation indicator** and their own **timeout** argument. However, if you use multiple lock arguments, be aware that a lock operation without a plus sign **lock operation** indicator unlocks all prior locks, including locks applied by an earlier part of the same **LOCK** command. For example, the command **LOCK ^b(1,1), ^c(1,2,3), ^d(1)** would be parsed as three separate lock commands: the first releasing the processes’ previously held locks (if any) and locking ^b(1,1), the second immediately releasing ^b(1,1) and locking ^c(1,2,3), the third immediately releasing ^c(1,2,3) locking ^d(1). As a result, only ^d(1) would be locked.

- **With Parentheses**: By enclosing a comma-separated list of lock names in parentheses, you can perform these locking operations on multiple locks as a single atomic operation. For example:

  ```
  LOCK +(var1(1),var2(1)):10
  ```
All lock operations in a parentheses-enclosed list are governed by a single lock operation indicator and a single timeout argument; either all of the locks are applied or none of them are applied. A parentheses-enclosed list without a plus sign lock operation indicator unlocks all prior locks then locks all of the listed lock names.

**Arguments**

**pc**

An optional postconditional expression that can make the command conditional. Caché executes the LOCK command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). You can specify a postconditional expression on an argumentless LOCK command or a LOCK command with arguments. For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

**lock operation indicator**

The lock operation indicator is used to apply (lock) or remove (unlock) a lock. It can be one of the following values:

| No character | Unlock all prior locks belonging to the current process and attempt to apply the specified lock. For example, LOCK ^a(1) performs the following atomic operation: it releases all locks previously held by the process (whether local or global, exclusive or shared, escalating or non-escalating) and it attempts to lock ^a(1). This can result in one of two outcomes: (1) all prior locks unlocked and ^a(1) locked; (2) all prior locks unlocked and ^a(1) placed in a lock waiting state pending the release of a conflicting lock held by another process. |
| Plus sign (+) | Attempt to apply the specified lock without performing any unlocks. This allows you to add a lock to the locks held by the current process. One use of this option is to perform incremental locking of a lock name. |
| Minus sign (-) | Unlocks the specified lock. If the lock name has a lock count of 1, an unlock remove the lock from the lock table. If the lock name has a lock count of more than 1, an unlock removes one of its incremental locks (decrements the lock count). By default, this unlocks an exclusive, non-escalating lock. To remove a shared lock and/or an escalating lock, you must specify the corresponding #locktype. |

If your LOCK command contains multiple comma-separated lock arguments, each lock argument can have its own lock operation indicator. Caché parses this as multiple independent LOCK commands.

**lockname**

A lockname is the name of a lock for a data resource; it is not the data resource itself. That is, your program can specify a lock named ^a(1) and a variable named ^a(1) without conflict. The relationship between the lock and the data resource is a programming convention; by convention, processes must acquire the lock before modifying the corresponding data resource.

Lock names are case-sensitive. Lock names follow the same naming conventions as the corresponding local variables and global variables. A lock name can be subscripted or unsubscripted. Lock subscripts have the same naming conventions and maximum length and number of levels as variable subscripts. In Caché, the following are all valid and unique lock names: a, a(1), A(1), ^a, ^a(1,2), ^A(1,1,1). For further details, see the Variables chapter of Using Caché ObjectScript.

**Note:** For performance reasons, it is recommended you specify lock names with subscripts whenever possible. For example, ^a (1) rather than ^a. Subscripted lock names are used in documentation examples.
Lock names can be local or global. A lock name such as $A(1)$ is a local lock name. It applies only to that process, but does apply across namespaces. A lock name that begins with a caret (^) character is a global lock name; the mapping for this lock follows the same mapping as the corresponding global, and thus can apply across processes, controlling their access to the same resource. (See Global Structure in Using Caché Globals.)

**Note:** Process-private global names can not be used as lock names. Attempting to use a process-private global name as a lock name performs no operation and completes without issuing an error.

A lock name can represent a local or global variable, subscripted or unsubscripted. It can be an implicit global reference, or an extended reference to a global on another computer. (See Global Structure in Using Caché Globals.)

The data resource corresponding to a lock name does not need to exist. For example, you may lock the lock name $^a(1,2,3)$ whether or not a global variable with the same name exists. Because the relationship between locks and data resources is an agreed-upon convention, a lock may be used to protect a data resource with an entirely different name.

**locktype**

A letter code specifying the type of lock to apply or remove. `locktype` is an optional argument; if you omit `locktype`, the lock type defaults to an exclusive non-escalating lock. If you omit `locktype`, you must omit the pound sign (#) prefix. If you specify `locktype`, the syntax for lock type is a mandatory pound sign (#), followed by quotation marks enclosing one or more lock type letter codes. Lock type letter codes can be specified in any order and are not case-sensitive. The following are the lock type letter codes:

- **S**: Shared lock

  Allows multiple processes to simultaneously hold nonconflicting locks on the same resource. For example, two (or more) processes may simultaneously hold shared locks on the same resource, but an exclusive lock limits the resource to one process. An existing shared lock prevents all other processes from applying an exclusive lock, and an existing exclusive lock prevents all other processes from applying a shared lock on that resource. However, a process can first apply a shared lock on a resource and then the same process can apply an exclusive lock on the resource, upgrading the lock from shared to exclusive. Shared and Exclusive lock counts are independent. Therefore, to release such a resource it is necessary to release both the exclusive lock and the shared lock. All locking and unlocking operations that are not specified as shared (“S”) default to exclusive.

  A shared lock may be incremental; that is, a process may issue multiple shared locks on the same resource. You may specify a shared lock as escalating (“SE”) when locking and unlocking. When unlocking a shared lock, you may specify the unlock as immediate (“SI”) or deferred (“SD”). To view the current shared locks with their increment counts for escalating and non-escalating lock types, refer to the system-wide lock table, described in the “Lock Management” chapter of Using Caché ObjectScript.

- **E**: Escalating lock

  Allows you to apply a large number of concurrent locks without overflowing the lock table. By default, locks are non-escalating. When applying a lock, you can use `locktype “E”` to designate that lock as escalating. When releasing an escalating lock, you must specify `locktype “E”` in the unlock statement. You can designate both exclusive locks and shared (“S”) locks as escalating.

  Commonly, you would use escalating locks when applying a large number of concurrent locks at the same subscript level. For example `LOCK ^mylock(1,1)#"E",^mylock(1,2)#"E",^mylock(1,3)#"E"...`

  The same lock can be concurrently applied as a non-escalating lock and as an escalating lock. For example, $^mylock(1,1)$ and $^mylock(1,1)#"E"$. Caché counts locks issued with `locktype “E”` separately in the lock table. For information on how escalating and non-escalating locks are represented in the lock table, refer to the “Lock Management” chapter of Using Caché ObjectScript.

  When the number of “E” locks at a subscript level reaches a threshold number, the next “E” lock requested for that subscript level automatically attempts to lock the parent node (the next higher subscript level). If it cannot, no escalation occurs. If it successfully locks the parent node, it establishes one parent node lock with a lock count corresponding to
the number of locks at the lower subscript level, plus 1. The locks at the lower subscript level are released. Subsequent “E” lock requests to the lower subscript level further increment the lock count of this parent node lock. You must unlock all “E” locks that you have applied to decrement the parent node lock count to 0 and de-escalate to the lower subscript level. The default lock threshold is 1000 locks; lock escalation occurs when the 1001st lock is requested.

Note that once locking is escalated, lock operations preserve only the number of locks applied, not what specific resources were locked. Therefore, failing to unlock the same resources that you locked can cause “E” lock counts to get out of sync.

In the following example, lock escalation occurs when the program applies the lock threshold + 1 “E” lock. This example shows that lock escalation both applies a lock on the next-higher subscript level and releases the locks on the lower subscript level:

```objectscript
Main
    TSTART
    SET thold=$SYSTEM.SQL.GetLockThreshold()
    WRITE "lock escalation threshold is ",thold,! 
    SET almost=thold-5
    FOR i=1:1:thold+5 { LOCK +dummy(1,i)#"E"
        IF i>almost {
            IF ^$LOCK("dummy(1,"_i_"),"OWNER") 'I' "" {WRITE "lower level lock applied at ",i,"th lock ",! }
            ELSEIF ^$LOCK("dummy(1)","OWNER") 'I' "" {WRITE "lock escalation",!
                WRITE "higher level lock applied at ",i,"th lock ",! 
                QUIT } 
        } 
        ELSE {WRITE "No locks applied",! }
    } 
    TCOMMIT
```

Note that only “E” locks are counted towards lock escalation. The following example applies both default (non-“E”) locks and “E” locks on the same variable. Lock escalation only occurs when the total number of “E” locks on the variable reaches the lock threshold:

```objectscript
Main
    TSTART
    SET thold=$SYSTEM.SQL.GetLockThreshold()
    WRITE "lock escalation threshold is ",thold,! 
    SET noE=17
    WRITE "setting ",noE," non-escalating locks",!
    FOR i=1:1:thold+noE { IF i < noE {LOCK +a(6,i)}
        ELSE {LOCK +a(6,i)#"E"}
            IF ^$LOCK("a(6)","OWNER") 'I' "" {WRITE "lock escalation on lock a(6,"i"),",! 
                QUIT } 
        } 
    } 
    TCOMMIT
```

Unlocking “E” locks is the reverse of the above. When locking is escalated, unlocks at the child level decrement the lock count of the parent node lock until it reaches zero (and is unlocked); these unlocks decrementing a count, they are not matched to specific locks. When the parent node lock count reaches 0, the parent node lock is removed and “E” locking de-escalates to the lower subscript level. Any subsequent locks at the lower subscript level create specific locks at that level.

The “E” locktype can be combined with any other locktype. For example, “SE”, “ED”, “EI”, “SED”, “SEI”. When combined with the “I” locktype it permits unlocks of “E” locks to occur immediately when invoked, rather than at the end of the current transaction. This “EI” locktype can minimize situations where locking is escalated.

Commonly, “E” locks are automatically applied for SQL INSERT, UPDATE, and DELETE operations within a transaction. However, there are specific limitations on SQL data definition structures that support “E” locking. Refer to the specific SQL commands for details.

- I: Immediate unlock
  Immediately releases a lock, rather than waiting until the end of a transaction:
Specifying “I” when unlocking a non-incremented (lock count 1) lock immediately releases the lock. By default, an unlock does not immediately release a non-incremented lock. Instead, when you unlock a non-incremented lock Caché maintains that lock in a delock state until the end of the transaction. Specifying “I” overrides this default behavior.

Specifying “I” when unlocking an incremented lock (lock count > 1) immediately releases the incremental lock, decrementing the lock count by 1. This is the same behavior as a default unlock of an incremented lock.

The “I” locktype is used when performing an unlock during a transaction. It has the same effect on Caché unlock behavior whether the lock was applied within the transaction or outside of the transaction. The “I” locktype performs no operation if the unlock occurs outside of a transaction. Outside of a transaction, an unlock always immediately releases a specified lock.

“I” can only be specified for an unlock operation; it cannot be specified for a lock operation. “I” can be specified for an unlock of a shared lock (#"SI") or an exclusive lock (#"I"). Locktypes “I” and “D” are mutually exclusive. “IE” can be used to immediately unlock an escalating lock.

This immediate unlock is shown in the following example:

```
TSTART
LOCK +^a(1)  // apply lock ^a(1)
LOCK -^a(1)  // remove (unlock) ^a(1)
// An unlock without a locktype defers the unlock
// of a non-incremented lock to the end of the transaction.
WRITE "Default unlock within a transaction.",!,"Go look at the Lock Table",!
HANG 10  // This HANG allows you to view the current Lock Table
LOCK +^a(1)  // reapply lock ^a(1)
LOCK -^a(1)#"I"  // remove (unlock) ^a(1) immediately
// this removes ^a(1) from the lock table immediately
// without waiting for the end of the transaction
WRITE "Immediate unlock within a transaction.",!,"Go look at the Lock Table",!
HANG 10  // This HANG allows you to view the current Lock Table
// while still in the transaction
TCOMMIT
```

- D: Deferred unlock

Controls when an unlocked lock is released during a transaction. The unlock state is deferred to the state of the previous unlock of that lock. Therefore, specifying locktype “D” when unlocking a lock may result in either an immediate unlock or a lock placed in delock state until the end of the transaction, depending on the history of the lock during that transaction. The behavior of a lock that has been locked/unlocked more than once differs from the behavior of a lock that has only been locked once during the current transaction.

The “D” unlock is only meaningful for an unlock that releases a lock (lock count 1), not an unlock that decrements a lock (lock count > 1). An unlock that decrements a lock is always immediately released.

“D” can only be specified for an unlock operation. “D” can be specified for a shared lock (#"SD") or an exclusive lock (#"D"). “D” can be specified for an escalating (“E”) lock, but, of course, the unlock must also be specified as escalating (“ED”). Lock types “D” and “I” are mutually exclusive.

This use of “D” unlock within a transaction is shown in the following examples. The HANG commands give you time to check the lock’s ModeCount in the Lock Table.

If the lock was only applied once during the current transaction, a “D” unlock immediately releases the lock. This is the same as “I” behavior. This is shown in the following example:

```
TSTART
LOCK +^a(1)  // Lock Table ModeCount: Exclusive
LOCK -^a(1)#"D"  // Lock Table ModeCount: null (immediate unlock)
HANG 10
TCOMMIT
```

If the lock was applied more than once during the current transaction, a “D” unlock reverts to the prior unlock state.

- If the last unlock was a standard unlock, the “D” unlock reverts unlock behavior to that prior unlock’s behavior — to defer unlock until the end of the transaction. This is shown in the following examples:
If the last unlock was an “I” unlock, the “D” unlock reverts unlock behavior to that prior unlock’s behavior — to immediately unlock the lock. This is shown in the following examples:

```
TSTART
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK -^a(1)      // Lock Table ModeCount: Exclusive
WRITE 1st unlock*,! HANG 5
LOCK -^a(1)#"I"  // Lock Table ModeCount: Exclusive
WRITE 2nd unlock*,! HANG 5
TCOMMIT
```

- If the last unlock was a “D” unlock, the “D” unlock follows the behavior of the last prior non-“D” lock:

```
TSTART
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK -^a(1)      // Lock Table ModeCount: Exclusive/2
WRITE 1st unlock*,! HANG 5
LOCK -^a(1)#"D"  // Lock Table ModeCount: null (immediate unlock)
WRITE 2nd unlock*,! HANG 5
TCOMMIT
```
timeout

The number of seconds to wait for a lock request to succeed before timing out. timeout is an optional argument. If omitted, the LOCK command waits indefinitely for a resource to be lockable; if the lock cannot be applied, the process will hang. The syntax for timeout is a mandatory colon (:), followed by an integer value or an expression that evaluates to an integer value. A value of zero permits one locking attempt before timing out. A negative number is equivalent to zero.

Commonly, a lock will wait if another process has a conflicting lock that prevents this lock request from acquiring (holding) the specified lock. The lock request waits until either a lock is released that resolves the conflict, or the lock request times out. Terminating the process also ends (deletes) pending lock requests. Lock conflict can result from many situations, not just one process requesting the same lock held by another process. A detailed explanation of lock conflict and lock request wait states is provided in the “Lock Management” chapter of Using Caché ObjectScript.

If you use timeout and the lock is successful, Caché sets the $TEST special variable to 1 (TRUE). If the lock cannot be applied within the timeout period, Caché sets $TEST to 0 (FALSE). Issuing a lock request without a timeout has no effect on the current value of $TEST. Note that $TEST can also be set by the user, or by a JOB, OPEN, or READ timeout.

The following example applies a lock on lock name ^abc(1,1), and unlocks all prior locks held by the process:

```
LOCK ^abc(1,1)
```

This command requests an exclusive lock: no other process can simultaneously hold a lock on this resource. If another process already holds a lock on this resource (exclusive or shared), this example must wait for that lock to be released. It can wait indefinitely, hanging the process. To avoid this, specifying a timeout value is strongly recommended:

```
LOCK ^abc(1,1):10
```

If a LOCK specifies multiple lockname arguments in a comma-separated list, each lockname resource may have its own timeout (syntax without parentheses), or all of the specified lockname resources may share a single timeout (syntax with parentheses).

- Without Parentheses: each lockname argument can have its own timeout. Caché parses this as multiple independent LOCK commands, so the timeout of one lock request does not affect the other lock arguments. Lock arguments are parsed in strict left-to-right order, with each lock request either completing or timing out before the next lock request is attempted.

- With Parentheses: all lockname arguments share a timeout. The LOCK must successfully apply all locks (or unlocks) within the timeout period. If the timeout period expires before all locks are successful, none of the lock operations specified in the LOCK command are performed, and control returns to the process.

Caché performs multiple operations in strict left-to-right order. Therefore, in LOCK syntax without parentheses, the $TEST value indicates the outcome of the last (rightmost) of multiple lockname lock requests.

In the following examples, the current process cannot lock ^a(1) because it is exclusively locked by another process. These examples use a timeout of 0, which means they make one attempt to apply the specified lock.

The first example locks ^x(1) and ^z(1). It sets $TEST=1 because ^z(1) succeeded before timing out:

```
LOCK +^x(1):0,+^a(1):0,+^z(1):0
```
The second example locks \(^x(1)\) and \(^z(1)\). It sets $TEST=0 because \(^a(1)\) timed out. \(^z(1)\) did not specify a timeout and therefore had no effect on $TEST:

\[
\text{LOCK } +^x(1):0,+^a(1):0,+^z(1)
\]

The third example applies no locks, because a list of locks in parentheses is an atomic (all-or-nothing) operation. It sets $TEST=0 because \(^a(1)\) timed out:

\[
\text{LOCK } (+^x(1),^a(1),^z(1)):0
\]

### Using the Lock Table to View and Delete Locks System-wide

Caché maintains a system-wide lock table that records all locks that are in effect and the processes that have applied them. The system manager can display the existing locks in the Lock Table or remove selected locks using the Management Portal interface or the \(^\text{LOCKTAB utility}\), as described in the “Lock Management” chapter of Using Caché ObjectScript. You can also use the \(\%\text{SYS.LockQuery}\) class to read lock table information. From the \(\%\text{SYS}\) namespace you can use the \(\text{SYS.Lock}\) class to manage the lock table.

You can use the Management Portal to view held locks and pending lock requests system-wide. Go to the Management Portal, select System Operation, select Locks, then select View Locks. For further details on the View Locks table refer to the “Lock Management” chapter of Using Caché ObjectScript.

You can use the Management Portal to remove (delete) locks currently held on the system. Go to the Management Portal, select System Operation, select Locks, then select Manage Locks. For the desired process (Owner) click either “Remove” or “Remove All Locks for Process”.

Removing a lock releases all forms of that lock: all increment levels of the lock, all exclusive, exclusive escalating, and shared versions of the lock. Removing a lock immediately causes the next lock waiting in that lock queue to be applied.

You can also remove locks using the \(\text{SYS.Lock.DeleteOneLock()}\) and \(\text{SYS.Lock.DeleteAllLocks()}\) methods.

Removing a lock requires WRITE permission. Lock removal is logged in the audit database (if enabled); it is not logged in cconsole.log.

### Incremental Locking and Unlocking

Incremental locking permits you to apply the same lock multiple times: to increment the lock. An incremented lock has a lock count of \(>1\). Your process can subsequently increment and decrement this lock count. The lock is released when the lock count decrements to 0. No other process can acquire the lock until the lock count decrements to 0. The lock table maintains separate lock counts for exclusive locks and shared locks, and for escalating and non-escalating locks of each type. The maximum incremental lock count is 32,766. Attempting to exceed this maximum lock count results in a <MAX LOCKS> error.

You can increment a lock as follows:

- **Plus sign**: Specify multiple lock operations on the same lock name with the plus sign lock operation indicator. For example: \(\text{LOCK } +^a(1) \text{ LOCK } +^a(1) \text{ LOCK } +^a(1) \text{ or LOCK } +^a(1),+^a(1),+^a(1) \text{ or LOCK } + (^a(1),^a(1),^a(1)).\) All of these would result in a lock table ModeCount of Exclusive/3. Using the plus sign is the recommended way to increment a lock.

- **No sign**: It is possible to increment a lock without using the plus sign lock operation indicator by specifying an atomic operation performing multiple locks. For example, \(\text{LOCK } (^a(1),^a(1),^a(1)) \text{ unlocks all prior locks and incrementally locks } ^a(1)\) three times. This too would result in a lock table ModeCount of Exclusive/3. While this syntax works, it is not recommended.

Unlocking an incremented lock when not in a transaction simply decrements the lock count. Unlocking an incremented lock while in a transaction has the following default behavior:
• Decrementing Unlocks: each decrementing unlock immediately releases the incremental unlock until the lock count is 1. By default, the final unlock puts the lock in delock state, deferring release of the lock to the end of the transaction. This is always the case when you delock with the minus sign lock operation indicator, whether or not the operation is atomic. For example: LOCK -^a(1) LOCK -^a(1) LOCK -^a(1) or LOCK -^a(1), -^a(1), -^a(1) or LOCK -(^a(1), ^a(1), ^a(1)). All of these begin with a lock table ModeCount of Exclusive/3 and end with Exclusive->Delock.

• Unlocking Prior Resources: an operation that unlocks all prior resources immediately puts an incremented lock into a delock state until the end of the transaction. For example, either LOCK x(3) (lock with no lock operation indicator) or an argumentless LOCK would have the following effect: the incremented lock would begin with a lock table ModeCount of Exclusive/3 and end with Exclusive/3->Delock.

Note that separate lock counts are maintained for the same lock as an Exclusive lock, a Shared lock, an Exclusive escalating lock and a Shared escalating lock. In the following example, the first unlock decrements four separate lock counts for lock ^a(1) by 1. The second unlock must specify all four of the ^a(1) locks to remove them. The HANG commands give you time to check the lock’s ModeCount in the Lock Table.

LOCK +(^a(1), ^a(1)#"E", ^a(1)#"S", ^a(1)#"SE")
LOCK +(^a(1), ^a(1)#"E", ^a(1)#"S", ^a(1)#"SE")
HANG 10
LOCK -(^a(1), ^a(1)#"E", ^a(1)#"S", ^a(1)#"SE")
HANG 10
LOCK -(^a(1), ^a(1)#"E", ^a(1)#"S", ^a(1)#"SE")

If you attempt to unlock a lock name that has no current locks applied, no operation is performed and no error is returned.

**Automatic Unlock**

When a process terminates, Caché performs an implicit argumentless LOCK to clear all locks that were applied by the process. It removes both held locks and lock wait requests.

**Locks on Global Variables**

Locking is typically used with global variables to synchronize the activities of multiple processes that may access these variables simultaneously. Global variables differ from local variables in that they reside on disk and are available to all processes. The potential exists, then, for two processes to write to the same global at the same time. In fact, Caché processes one update before the other, so that one update overwrites and, in effect, discards the other.

Global lock names begin with a caret (^) character.

To illustrate locking with global variables, consider the case in which two data entry clerks are concurrently running the same student admissions application to add records for newly enrolled students. The records are stored in a global array named ^student. To ensure a unique record for each student, the application increments the global variable ^index for each student added. The application includes the LOCK command to ensure that each student record is added at a unique location in the array, and that one student record does not overwrite another.

The relevant code in the application is shown below. In this case, the LOCK controls not the global array ^student but the global variable ^index. ^index is a scratch global that is shared by the two processes. Before a process can write a record to the array, it must lock ^index and update its current value (SET ^index=^index+1). If the other process is already in this section of the code, ^index will be locked and the process will have to wait until the other process releases the lock (with the argumentless LOCK command).

```
READ !,"Last name: ",!,,lname QUIT:lname="" SET lname=lname_","
READ !,"First name: ",!,,fname QUIT:fname="" SET fname=fname_","
READ !,"Middle initial: ",!,,minit QUIT:minit="" SET minit=minit_":"
READ !,"Student ID Number: ",!,,sid QUIT:sid=""
SET rec = lname_fname_minit_sid
LOCK ^index
SET ^index = ^index + 1
SET ^student(^index)=rec
LOCK
```
The following example recasts the previous example to use locking on the node to be added to the ^student array. Only the affected portion of the code is shown. In this case, the ^index variable is updated after the new student record is added. The next process to add a record will use the updated index value to write to the correct array node.

LOCK ^student(^index)
SET ^student(^index) = rec
SET ^index = ^index + 1
LOCK /* release all locks */

Note that the lock location of an array node is where the top level global is mapped. Caché ignores subscripts when determining lock location. Therefore, ^student(name) is mapped to the namespace of ^student, regardless of where the data for ^student(name) is stored.

**Locks in a Network**

In a networked system, one or more servers may be responsible for resolving locks on global variables.

You can use the `LOCK` command with any number of servers, up to 255. A `LOCK` command can specify a maximum of 8 remote systems, and a maximum of 20 locks for each remote system.

You can use ^$LOCK to list remote locks, but it cannot list the lock state of a remote lock.

Remote locks held by a client job on a remote server system are released when you call the ^RESJOB utility to remove the client job.

**Local Variable Locks**

The behavior of `LOCK` on local (non-careted) variables is the same on Caché and Open M [DTM]. The Config.Miscellaneous class provides a ZaMode property that you can use to set the locking mode system-wide for compatibility with the older DSM–11 ZALLOCATE (ZA) and ZDEALLOCATE (ZD) locking commands.

In Open M [DTM] local locks are always taken out in the manager's dataset regardless of the first character. If an application ported to Caché uses local locks that do not begin with a % character, the application may experience deadlock conditions if it uses the same, local, non-% lock names in different datasets. The locks now collide because they resolve to the same item. If this is part of a hierarchical locking system, a deadly embrace may occur that can hang an application.

The current behavior is:

- Local (non-careted) locks acquired in the context of a specific namespace, either because the default namespace is an explicit namespace or through an explicit reference to a namespace, are taken out in the manager's dataset on the local machine. This occurs regardless of whether the default mapping for globals is a local or a remote dataset.
- Local (non-careted) locks acquired in the context of an implied namespace or through an explicit reference to an implied namespace on the local machine, are taken out using the manager's dataset of the local machine. An implied namespace is a directory path preceded by two caret characters: "^^dir".

Referencing explicit and implied namespaces is further described in **Global Structure** in *Using Caché Globals*.

**See Also**

- $TEST special variable
- ^$LOCK structured system variable
- Lock Management in *Using Caché ObjectScript*
- Using ObjectScript for Transaction Processing in *Using Caché ObjectScript*
- The Monitoring Locks section of the “Monitoring Caché Using the Management Portal” chapter in *Caché Monitoring Guide*
- The article *Locking and Concurrency Control*
MERGE

Merges global nodes or subtrees from source into destination.

MERGE:pc mergeargument,...
M:pc mergeargument,...

where mergeargument is:

destination=source

Arguments

<table>
<thead>
<tr>
<th>pc</th>
<th>Optional — A postconditional expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>destination and source</td>
<td>Local variables, process-private globals, or globals to be merged. If specified as a class property, the source variable must be a multidimensional (subscripted) variable.</td>
</tr>
</tbody>
</table>

Description

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.

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. MERGE offers a one-command alternative to the current technique for doing subtree copy: a series of SET commands with $ORDER references.

MERGE issues a <COMMAND> error if the source and destination have a parent-child relationship.

MERGE Execution

The MERGE command can take longer than most other ObjectScript 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.

MERGE is a non-atomic operation. When executing MERGE while other processes are performing concurrent data modification operations, the contents of destination will be the state of the data at the time that MERGE was initiated, minus any variables that were KILLED at the time the MERGE operation concluded. Other data modifications that occurred during MERGE processing may not be reflected in the contents of destination.

Arguments

pc

An optional postconditional expression. Caché executes the MERGE command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

destination and source

Variables to be merged. Either variable can be a local variable, a process-private global, or a global. If destination is undefined, MERGE defines it and sets it to source. If source is undefined, MERGE completes successfully, but does not change destination.

You can specify multiple, comma-separated destination=source pairs. They are evaluated in left-to-right order.
A mergeargument can reference a destination=source pair by indirection. For example, MERGE @tMergeString.

A mergeargument can be an array passed by reference that specifies a variable number of parameters, such in myargs...

Examples

The following example copies a subtree from one global variable (^a) to another global variable (^b). In this case, the merge is being used to create a smaller global ^b, which contains only the ^a(1,1) subtree of the information in ^a.

```
SET ^a="cartoons"
SET ^a(1)="The Flintstones",^a(2)="The Simpsons"
SET ^a(1,1)="characters",^a(1,2)="place names"
SET ^a(1,1,1)="Flintstone family"
SET ^a(1,1,1,1)="Fred"
SET ^a(1,1,1,2)="Wilma"
SET ^a(1,1,2)="Rubble family"
SET ^a(1,1,2,1)="Barney"
SET ^a(1,1,2,2)="Betty"
MERGE ^b=^a(1,1)
WRITE ^b,!,^b(2),!,^b(2,1)," and ",^b(2,2)
```

The following example shows how a destination global variable looks after it has been merged with a subtree of a source global variable.

Suppose you execute the following:

```
KILL ^X,^Y
SET ^X(2,2)="first"
SET ^X(2,2,4)="second"
SET ^Y(3,6,7)="third"
SET ^Y(3,6,8)="fourth"
SET ^Y(3,6,7,8)="fifth"
SET ^Y(3,6,7,8,9)="sixth"
WRITE ^X(2,2),!,^X(2,2,4),!
WRITE ^Y(3,6,7),!,^Y(3,6,8),!
WRITE ^Y(3,6,7,8,4),!,^Y(3,6,7,8,9)
```

The following figure shows the resulting logical structure of ^X and ^Y.

```
Figure B–2: Initial Structure of ^X and ^Y
```

Consider the following MERGE command:

```
MERGE ^X(2,3)=^Y(3,6,7,8)
```

When you issue the previous statement, Caché copies part of ^Y into ^X(2,3). The global ^X now has the structure illustrated in the figure below.
**Notes**

**Naked Indicator**

When both `destination` and `source` are local variables, the naked indicator is not changed. If `source` is a global variable and `destination` is a local variable, then the naked indicator references `source`.

When both `source` and `destination` are global variables, the naked indicator is unchanged if `source` is undefined ($DATA(source)=0). In all other cases (including $DATA(source)=10), the naked indicator takes the same value that it would have if the `SET` command replaced the `MERGE` command and `source` had a value. For more details on the naked indicator, see Naked Global Reference in Using Caché Globals.

**Merge to Self**

When the `destination` and `source` are the same variable, no merge occurs. Nothing is recorded in the journal file. However, the naked indicator may be changed, based on the rules described in the previous section.

**Watchpoints**

The `MERGE` command supports watchpoints. If a watchpoint is in effect, Caché triggers that watchpoint whenever that `MERGE` alters the value of a watched variable. To set watchpoints, use the `ZBREAK` command.

**See Also**

- `ZBREAK` command
- Debugging chapter in Using Caché ObjectScript
- Global Structure chapter in Using Caché Globals
**NEW**

Creates empty local variable environment.

```
NEW: pc newargument,...
N: pc newargument,...
```

where `newargument` can be:

```
variable,...
(variable,...)
```

### Arguments

<table>
<thead>
<tr>
<th>pc</th>
<th>Optional — A postconditional expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>Optional — Name of variable(s) to be added to the existing local variable environment. The effect of a <strong>NEW</strong> on existing local variables depends on whether <code>variable</code> is enclosed in parentheses (exclusive <strong>NEW</strong>) or is not enclosed in parentheses (inclusive <strong>NEW</strong>). A <code>variable</code> must be a valid local variable name, but does not have to be a defined variable; specifying an undefined variable neither issues an error nor defines the variable.</td>
</tr>
</tbody>
</table>

### Description

The **NEW** command has two forms:

- Without an argument
- With an argument

**NEW** without an argument creates an empty local variable environment for a called subroutine or user-defined function. Existing local variable values are not available in this new local environment. They can be restored by returning to the previous local environment.

The action **NEW** with an argument performs depends on the argument form you use.

- **NEW variable** (inclusive **NEW**) retains the existing local variable environment and adds the specified variable(s) to it. If any of the specified local variables has the same name as an existing local variable, the old value for that named variable is no longer accessible in the current environment.
- **NEW (variable)** (exclusive **NEW**) replaces all existing variables in the local variable environment except the specified variable(s).

### NEW Restrictions

The **NEW** command (inclusive or exclusive) cannot be used on the following:

- **Globals**
- **Process-Private Globals**
- **Local variable subscripts**
- **Private variables**
- **Special variables, except $ESTACK, $ETRAP, $NAMESPACE, and $ROLES**

Attempting to use **NEW** in any of these contexts results in a `<SYNTAX>` error.
Arguments

pc
An optional postconditional expression. Caché executes the NEW command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

variable
Name of a single variable or a comma-separated list of variable names. You can specify only unsubscripted variable names, although you can NEW an entire array (that is, an array name without subscripts). You can specify undefined variable names or you can reuse the names of existing local variables. For an inclusive NEW, when you specify an existing local variable, Caché reinitializes that variable in the local environment, but saves its current value on the program stack and restores it after the subroutine or function terminates.

When a variable name or comma-separated list of variable names is enclosed in parentheses (exclusive NEW), Caché performs the opposite operation. That is, all local variables are reinitialized except the specified variable names, which retain their previous values. Caché saves the current values of all variables on the program stack and restores them after the subroutine or function terminates.

Examples

The following example illustrates an inclusive NEW, which keeps the existing local variables a, b, and c, and adds variables d and e, in this case, overlaying the prior value of d.

Main
SET a=7,b=8,c=9,d=10
WRITE !,"Before NEW:","a="a,"b="b,"c="c,"d="d
DO Sub1
WRITE !,"Returned to prior context:"
WRITE !,"a="a,"b="b,"c="c,"d="d
QUIT
Sub1
NEW d,e
SET d="big number"
WRITE !,"After NEW:","a="a,"b="b,"c="c,"d="d
QUIT

The following example illustrates an exclusive NEW, which removes all existing local variables except the specified variables a and c.

SET a=7,b=8,c=9,d=10
NEW (a,c)
WRITE

Notes

Where to Use NEW

NEW allows you to insulate the current process’s local variable environment from changes made by a subroutine, user-defined function, or XECUTE string. NEW is most frequently used within a subroutine called by the DO command.

The basic purpose of the NEW command is to redefine the local variable environment within a called subroutine or user-defined function. A subroutine or user-defined function called without parameter passing inherits its local variable environment from the calling routine. To redefine this environment for a subroutine or function, you can use NEW for all local variables (argumentless NEW), for named local variables (inclusive NEW) or for all local variables except the named variables (exclusive NEW).

Within a procedure, variables are either private or public:

- By default, local variables are private to that procedure. The procedure block uses private variables that do not interact with variables with the same names outside the procedure block. You cannot perform a NEW on a private variable;
attempting an inclusive or exclusive NEW on a private variable within a procedure block results in a <SYNTAX> error.

- When declaring a procedure, you can explicitly list local variables as public variables. You can perform a NEW on a public variable within a procedure block. This NEW only affects the variable value within that procedure. You can repeatedly NEW a public variable within a procedure.

For further details, refer to “Procedure Variables” in Using Caché ObjectScript.

Special considerations apply in the case of a subroutine called by the DO command with parameter passing. These considerations are described under "Subroutines with Parameter Passing".

NEW and KILL

Variables created by NEW do not require explicit and corresponding KILL commands. When a called subroutine or a user-defined function terminates, Caché executes an implicit KILL for each variable initialized by a NEW command within that subroutine or function.

Inclusive NEW

An inclusive NEW — NEW variable — retains the existing local variable environment and adds the specified variables to it. If an existing variable is named, the “new” variable replaces the existing variable, which is saved on the stack and then restored when the subroutine or function terminates.

Inclusive NEW does not restrict the number of variables you can specify as a comma-separated list. Inclusive NEW also does not limit the number of local variable environment levels (number of times NEW is issued).

In the following example, assume that the local variable environment of the calling routine (Main) consists of variables a, b, and c. When the DO calls Subr1, the NEW command redefines Subr1’s local variable environment to new variable c and add variable d. After the NEW, the subroutine’s environment consists of the existing variables a and b plus the new variables c and d. The variables a and b are inherited and retain their existing values. The new variables c and d are created undefined. Since c is the name of an existing local variable, the system saves the existing value on the stack and restores it when Subr1 QUITs. Note that the first SET command in Subr1 references a and b to assign a value to d. Note that variable c in this context is undefined.

```objectscript
Main
SET a=2,b=4,c=6
WRITE !,"c in Main before DO: ",c
DO Subr1(a,b,c)
WRITE !,"c in Main after DO: ",c
QUIT
```

```objectscript
Subr1(a,b,c)
NEW c,d
IF $DATA(c) {WRITE !,"c=",c}
ELSE {WRITE !,"c in Subr1 is undefined"}
SET d=a*b
SET c=d+2
WRITE !,"c in Subr1: ",c
QUIT
```

When executed, this code produces the following results:

c before DO: 6
c in Subr1 is undefined
c in Subr1: 16
c after DO: 6

The results are the same whether passing parameters by value, as in the previous example, or passing parameters by reference:
Main
  SET a=2,b=4,c=6
  WRITE !,"c in Main before DO: ",c
  DO Subr1(a,b,c)
  WRITE !,"c in Main after DO: ",c
  QUIT
Subr1(&a,&b,&c)
  NEW c,d
  IF $DATA(c) {WRITE !,"c=",c}
  ELSE {WRITE !,"c in Subr1 is undefined"}
  SET d=a*b
  SET c=d*2
  WRITE !,"c in Subr1: ",c
  QUIT

Note that variable $c$ is passed to Subr1 and then immediately redefined using **NEW**. In this case, passing variable $c$ was unnecessary; the program results are identical whether or not $c$ is passed. If you **NEW** any of the variables named in the subroutine’s formal parameter list, you render them undefined and make their passed values inaccessible.

**Exclusive NEW**

An exclusive **NEW** — NEW (var1, var2) — replaces the entire existing local variable environment except the specified variables. If an existing variable is named, it is retained and can be referenced in the new environment. However, any changes made to such a variable are reflected in the existing variable when the function or subroutine terminates.

An exclusive **NEW** can specify a maximum of 255 variables as a comma-separated list. Exceeding this number causes Caché to issue a `<SYNTAX>` error.

Exclusive **NEW** (NEW (x,y,z)) temporarily removes local variables from the current scope. This can affect local variables created by Caché objects. For example, Caché maintains %objcn which is the cursor pointer for Caché object queries. Removing this from the current scope can result in collisions with other internal structures. Therefore, do not use exclusive **NEW** in any context where it might affect system structures.

Attempting to issue more than 31 levels of exclusive **NEW** or argumentless **NEW** results in a `<MAXSCOPE>` error.

When using exclusive **NEW** in a **FOR** code block, you must specify the **FOR** count variable as an excluded variable. For further details, refer to the **FOR** command.

In the following example, assume that the local variable environment of the calling routine (Start) consists of variables $a$, $b$, and $c$. When the **DO** calls Subr1, the **NEW** command redefines Subr1’s local variable environment to exclude all variables except $c$ and $d$.

After the **NEW**, the subroutine’s environment consists only of the new variables $c$ and $d$. The new variable $c$ is retained from the calling routine’s environment and keeps its existing value. The new variable $d$ is created undefined.

The first **SET** command in Subr1 references $c$ to assign a value to $d$. The second **SET** command assigns a new value (24) to $c$. When the subroutine **QUITs**, $c$ will have this updated value (and not the original value of 6) in the calling routine’s environment.

Start
  SET a=2,b=4,c=6
  DO Subr1
  WRITE !,"c in Start: ",c
  QUIT
Subr1
  NEW (c,d)
  SET d=c+c
  SET c=d+2
  WRITE !,"c in Subr1: ",c
  QUIT

When executed, this code produces the following results:

- $c$ in Subr1: 24
- $c$ in Start: 24
**Argumentless NEW**

The argumentless **NEW** provides an empty local variable environment for a called subroutine or user-defined function. The existing local variable environment (in the calling routine) is saved and then restored when the subroutine or function terminates. Any variables created after the **NEW** are deleted when the subroutine or function terminates.

If a command follows the **NEW** on the same line, be sure to separate the **NEW** command from the command following it by (at least) two spaces.

Argumentless **NEW** should not be used within the body of a **FOR** loop or in a context in which it can affect Caché objects.

Attempting to issue more than 31 levels of exclusive **NEW** or argumentless **NEW** results in a <MAXSCOPE> error.

**Special Variables: $ESTACK, $ETRAP, $NAMESPACE, and $ROLES**

You cannot use **NEW** on most special variables; attempting to do so results in a <SYNTAX> error. There are four exceptions: $ESTACK, $ETRAP, $NAMESPACE, and $ROLES.

- **SETRAP**: When you issue the command **NEW** $ETRAP, the system creates a new context for error trapping. You can then set $ETRAP in this new context with the desired error trapping command(s). The $ETRAP value in the previous context is preserved. If you set $ETRAP without first issuing the **NEW** $ETRAP command, Caché sets $ETRAP to this value in all contexts. It is therefore recommended that you always **NEW** the $ETRAP special variable before setting it.

- **NAMESPACE**: When you issue the command **NEW** $NAMESPACE, the system creates a new namespace context. Within this context you can change the namespace. When you exit this context (with **QUIT**, for example) Caché reverts to the prior namespace.

**Subroutines with Parameter Passing**

If you call a subroutine with parameter passing, Caché issues an implicit **NEW** command for each of the variables named in the subroutine’s formal parameter list. It then assigns the values passed in the **DO** command’s actual parameter list (by value or by reference) to these variables.

If the **DO** command uses parameter passing by value and if the formal list names any existing local variables, Caché places the existing variables and their values on the stack. When the subroutine terminates (with either an explicit or an implicit **QUIT**), Caché issues an implicit **KILL** command for each of the formal list variables to restore them from the stack.

**See Also**

- **DO** command
- **QUIT** command
- **SET** command
- $ETRAP special variable
- $NAMESPACE special variable
OPEN

Acquires ownership of a device or file for input/output operations.

```
OPEN:pc device:(parameters):timeout:"mnespace",...
O:pc device:(parameters):timeout:"mnespace",...
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td><strong>Optional</strong> — A postconditional expression.</td>
</tr>
<tr>
<td>device</td>
<td>The device to be opened, specified by a device ID or a device alias. A device ID can be an integer (a device number), a device name, or the pathname of a sequential file. If a string, it must be enclosed with quotation marks. The maximum length of device is 256 characters.</td>
</tr>
<tr>
<td>parameters</td>
<td><strong>Optional</strong> — The list of parameters used to set device characteristics. The parameter list is enclosed in parentheses, and the parameters in the list are separated by colons. Parameters can either be positional (specified in a fixed order in the parameter list) or keyword (specified in any order). A mix of positional and keyword parameters is permitted. The individual parameters and their positions and keywords are highly device-dependent.</td>
</tr>
<tr>
<td>timeout</td>
<td><strong>Optional</strong> — The number of seconds to wait for the request to succeed, specified as an integer. Fractional seconds are truncated to the integer portion. If omitted, Caché waits indefinitely.</td>
</tr>
<tr>
<td>mnespace</td>
<td><strong>Optional</strong> — The name of the mnemonic space that contains the control mnemonics to use with this device, specified as a quoted string.</td>
</tr>
</tbody>
</table>

**Description**

Use the **OPEN** command to acquire ownership of a specified device (or devices) for input/output operations. An **OPEN** retains ownership of the device until ownership is released with the **CLOSE** command.

An **OPEN** command can be used to open multiple devices by using a comma to separated the specifications for each device. Within the specification of a device, its arguments are separated by using colons (:). If an argument is omitted, the positional colon must be specified; however, trailing colons are not required.

The **OPEN** command can be used to open devices such as terminal devices, magnetic tape devices, spool devices, TCP bindings, interprocess pipes, named pipes, and interjob communications.

The **OPEN** command can also be used to open a sequential file. The **device** argument specifies the file pathname as a quoted string. The **parameters** argument specifies the parameters governing the sequential file. These parameters can include the option of creating a new file if the specified file does not exist. Specifying the **timeout** argument, though optional, is strongly encouraged when opening a sequential file.

Sequential file open option defaults are set for the current process using the `%SYSTEM.Process` class `OpenMode()` and `FileMode()` methods, and system-wide by using the `Config.Miscellaneous` class `OpenMode` and `FileMode` properties. For much more details on opening sequential files, see **Sequential File I/O** in the **Caché I/O Device Guide**.

The **OPEN** command is not used to access a Caché database file.

On Windows, ObjectScript allocates each process an open file quota between database files and files opened with **OPEN**. When **OPEN** causes too many files to be allocated to **OPEN** commands, you receive a `<TOOMANYFILES>` error. The Caché maximum number of open files for a process is 1,024. The actual maximum number of open files for each process
is a platform-specific setting. For example, Windows defaults to a maximum of 998 open files per process. Consult the operating system documentation for your system.

**Arguments**

**pc**
An optional postconditional expression. Caché executes the OPEN command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the OPEN command if the postconditional expression is false (evaluates to zero). Only one postconditional is permitted, even if the OPEN command opens multiple devices or files. For further details, refer to Command Postconditional Expressions in *Using Caché ObjectScript*.

**device**
The device to be opened. You can specify the device using any of the following:

- Physical device number, specified as a positive integer. For example, 2 is always the spooler device. This is the only way to specify a magnetic tape device (numbers 47 through 62). This number is internal to Caché and is unrelated to device numbers assigned by the platform operating system.
- Device ID, specified as a quoted string. For example, "|TRM| : |4294318809". This value for the current device is found in the $IO special variable.
- Device alias, specified as a positive integer. A device alias refers to a physical device number.
- File pathname, specified as a quoted string. This is used for opening sequential files. A pathname can be canonical (c:\myfiles\testfile) or relative to the current directory (\myfiles\testfile).

The maximum length of device is 256 characters for Windows and UNIX®. See "Specifying the Device" for more information.

**parameters**
The list of parameters used to set operating characteristics of the device to be opened. The enclosing parentheses are required if there is more than one parameter. (It’s good programming practice to always use parentheses when you specify a parameter.) Note the required colon before the left parenthesis. Within the parentheses, colons are used to separate multiple parameters.

The parameters for a device can be specified using either positional parameters or keyword parameters. You can also mix positional parameters and keyword parameters within the same parameter list.

In most cases, specifying contradictory, duplicate, or invalid parameter values does not result in an error. Wherever possible, Caché ignores inappropriate parameter values and takes appropriate defaults.

If you do not specify a list of parameters, Caché uses the device’s default parameters. The default parameters for a device are configurable. Go to the Management Portal, select [System] > [Configuration] > [Device Settings] > [Devices] to display the current list of devices. For the desired device, click “edit” to display its Open Parameters: option. Specify this value in the same way you specify the OPEN command parameters, including the enclosing parentheses. For example, ("AVL":0:2048).

The available parameters are specific to the type of device that is being opened. For more information on device parameters, see I/O Devices and Commands in *Caché I/O Device Guide*.

**Positional Parameters**
Positional parameters must be specified in a fixed sequence in the parameter list. You can omit a positional parameter (and receive the default value), but you must retain the colon to indicate the position of the omitted positional parameter. Trailing colons are not required; excess colons are ignored. The individual parameters and their positions are highly device-dependent. There are two types of positional parameters: values and letter code strings.
A value can be an integer (for example, record size), a string (for example, host name), or a variable or expression that evaluates to a value.

A letter code string uses individual letters to specify device characteristics for the open operation. For most devices, this letter code string is one of the positional parameters. You can specify any number of letters in the string, and specify the letters in any order. Letter codes are not case-sensitive. A letter code string is enclosed in quotation marks; no spaces or other punctuation is allowed within a letter code string (exception: K and Y may be followed by a name delimited by backslashes: thus: K\name\). For example, when opening a sequential file, you might specify a letter code string of “ANDFW” (append to existing file, create new file, delete file, fix-length records, write access.) The position of the letter code string parameter, and the meanings of individual letters is highly device-dependent.

**Keyword Parameters**

Keyword parameters can be specified in any sequence in the parameter list. A parameter list can consist entirely of keyword parameters, or it can contain a mix of positional and keyword parameters. (Commonly, the positional parameters are specified first (in their correct positions) followed by the keyword parameters.) You must separate all parameters (positional or keyword) with a colon (:). A parameter list of keyword parameters has the following general syntax:

```
OPEN device:(/KEYWORD1=value1:/KEYWORD2=value2:.../KEYWORDn=valuen):timeout
```

The individual parameters and their positions are highly device-dependent. As a general rule, you can specify the same parameters and values using either a positional parameter or a keyword parameter. You can specify a letter code string as a keyword parameter by using the /PARAMS keyword.

**timeout**

The number of seconds to wait for the OPEN request to succeed. The preceding colon is required. timeout must be specified as an integer value or expression. If timeout is set to zero (0), OPEN makes a single attempt to open the file. If the attempt fails, the OPEN immediately fails. If the attempt succeeds it successfully opens the file. If timeout is not set, Caché will continue trying to open the device until the OPEN is successful or the process is terminated manually. If you use the timeout option and the device is successfully opened, Caché sets the $TEST special variable to 1 (TRUE). If the device cannot be opened within the timeout period, Caché sets $TEST to 0 (FALSE). Note that $TEST can also be set by the user, or by a JOB, LOCK, or READ timeout.

**mnespace**

The name of the mnemonic space that contains the device control mnemonics used by this device. By default, Caché provides two mnemonic spaces: ^%XMAG for magnetic tape devices, and ^%X364 (ANSI X3.64 compatible) for all other devices and sequential files. Default mnemonic spaces are assigned by device type.

Go to the Management Portal, select [System] > [Configuration] > [Device Settings] > [IO Settings]. View and edit the File, MagTape, Other, or Terminal mnemonic space setting.

A mnemonic space is a routine that contains entry points for the device control mnemonics used by READ and WRITE commands. The READ and WRITE commands invoke these device control mnemonics using the /mnemonic(params) syntax. These device control mnemonics perform operations such as moving the cursor to a specified screen location or rewinding a magnetic tape.

Use the mnespace argument to override the default mnemonic space assignment. Specify an ObjectScript routine that contains the control mnemonics entry points used with this device. The enclosing double quotes are required. Specify this option only if you plan to use device control mnemonics with the READ or WRITE command. If the mnemonic space does not exist, a <NOROUTINE> error is returned. For further details on mnemonic spaces, see I/O Devices and Commands in the Caché I/O Device Guide.

**Examples**

In the following example, the OPEN command attempts to acquire ownership of device 2 (the spooler). The first positional parameter (3) specifies the file number within the ^SPOOL global and the second positional parameter (12) specifies the
OPEN 2: (3:12)

In the following example, the OPEN command attempts to acquire ownership of the sequential file CUSTOMER within the timeout period of 10 seconds.

OPEN "\myfiles\customer":10

Note that because no parameters are specified, the parentheses are omitted, but the colon must be present.

The following example opens a sequential file named Seqtest; the letter code positional parameter is "NRW". The “N” letter code specifies that if the file does not exist, create a new sequential file with this name. The “R” and “W” letter codes specify that the file is being opened for reading and writing. The timeout is 5 seconds.

NEW $NAMESPACE
SET $NAMESPACE="%SYS"
SET dir=##class(%SYSTEM.Process).CurrentDirectory() ; determine Caché directory
SET seqfilename=dir_"Samples\Seqtest"
OPEN seqfilename:"NRW":5
WRITE !,"Opened a sequential file named Seqtest"
USE seqfilename
WRITE "a line of data for the sequential file"
CLOSE seqfilename:"D"
WRITE !,"Closed and deleted Seqtest"
QUIT

This example requires that UnknownUser have assigned the %DB_CACHESYS role.

Notes

Device Ownership and the Current Device

OPEN establishes ownership of the specified device. The process retains ownership of the device until the process either terminates or releases the device with a subsequent CLOSE command. While a device is owned by a process, no other process can acquire or use the device.

A process can own multiple devices at the same time. However, only one device can be the current device. You establish an owned device as the current device with the USE command. The ID of the current device is found in the $IO special variable.

A process always owns at least one device (designated as device 0), which is its principal device. This device is assigned to the process when it is started and is typically the terminal used to sign onto Caché. The ID of the principal device is found in the $PRINCIPAL special variable.

When a process terminates, Caché issues an implicit CLOSE for each of the devices owned by the process and returns them to the pool of available devices.

Changing Parameters for an Owned Device

To change the parameters for a device that is already owned by the process, you can:

- Close and then reopen the device with new parameter values.
- If the device is a terminal, TCP, or magnetic tape device, you can issue an OPEN with new parameter values on an already open device.

If you specify the device on another OPEN command, any device parameters set by the initial OPEN command remain in effect unless explicitly changed. Depending on the type of device, subsequent I/O may be different than if you had closed and then reopened the device.

For some devices, you can omit the parameters option and later set the desired characteristics with the parameters option on the USE command.
**Specifying the Device**

When you open a device, you can identify the device by supplying a device number or an alias assigned to the device.

**Using Physical Device Numbers**

Caché allows you to identify certain devices by supplying their system-assigned physical device numbers. All implementations of Caché recognize the following physical device numbers:

- 0 = The process’s principal device (usually the device at which you sign on).
- 2 = The spooler (to store output for later printing).
- 63 = The view buffer.

**OPEN 63** accepts a namespace, as shown in the following example:

```
OPEN 63: "SAMPLES"
```

If you specify a nonexistent namespace, Caché issues a <NAMESPACE> error. If you specify a namespace for which you do not have privileges, Caché issues a <PROTECT> error.

Device 3 is an invalid device; attempting to open it returns a <NOTOPEN> error without waiting for *timeout* expiration.

See About I/O Devices in *Caché I/O Device Guide* for more information about device numbers.

**Using a Device Alias**

An alias is an alternate numeric device ID. It must be a valid device number, it must be unique and cannot conflict with an assigned device number.

You can establish a numeric alias for a device. Go to the Management Portal, select [System] > [Configuration] > [Device Settings] > [Devices] to display the current list of devices and their aliases. For the desired device, click “edit” to edit its Alias: option.

After you have assigned an alias to a device, you can use the **OPEN** command or the **%IS** utility to open the device using this alias.

**Exceeding the Open File Quota**

Caché allocates each process’ open file quota between database files and files opened with **OPEN**. When **OPEN** causes too many files to be allocated to **OPEN** commands, you receive a <TOOMANYFILES> error. The Caché maximum number of open files for a process is 1,024. The actual maximum number of open files for each process is a platform-specific setting. For example, Windows defaults to a maximum of 998 open files per process.

**Default Record Length**

If the record size for sequential files is not specified in the **OPEN** command, Caché assumes a default record length of 32,767 characters regardless of whether *long strings* are enabled or not.

**See Also**

- **CLOSE** command
- **USE** command
- **$TEST** special variable
- **$IO** special variable
- **I/O Devices and Commands** in *Caché I/O Device Guide*
- Terminal I/O in *Caché I/O Device Guide*
• TCP Client/Server Communication in Caché I/O Device Guide
• Magnetic Tape I/O in Caché I/O Device Guide
• Sequential File I/O in Caché I/O Device Guide
• The Spool Device in Caché I/O Device Guide
• ^%IS global and %IS utility in Caché I/O Device Guide
QUIT

Terminates execution of a loop structure or a routine.

QUIT:pc expression
Q:pc expression
QUIT n
Q n

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>Optional</td>
<td>A postconditional expression.</td>
</tr>
<tr>
<td>expression</td>
<td>Optional</td>
<td>A value to return to the invoking routine; a valid expression.</td>
</tr>
<tr>
<td>n</td>
<td>Optional</td>
<td>Terminal prompt only: The number of program levels to clear; an expression that resolves to a positive integer.</td>
</tr>
</tbody>
</table>

Description

The QUIT command is used in two different contexts:

- QUIT in program code
- QUIT at the Terminal prompt

In Program Code

The QUIT command terminates execution of the current context, exiting to the enclosing context. When invoked in a routine, QUIT returns to the calling routine, or terminates the program if there is no calling routine. When invoked from within a FOR, DO WHILE, or WHILE flow-of-control structure, a TRY or CATCH block, or a legacy DO structure, the QUIT breaks out of the structure and continues execution with the next command outside of that structure.

The similar RETURN command terminates execution of a routine at any point, including from within a FOR, DO WHILE, or WHILE loop or nested loop structure, a TRY or CATCH block, or a legacy DO structure. RETURN always exits the current routine, returning to the calling routine or terminating the program if there is no calling routine.

The QUIT command has two forms:

- Without an argument
- With an argument

A postconditional is not considered an argument. The $QUIT special variable indicates whether or not an argumented QUIT command is required to exit from the current context. Two error codes are provided for this purpose: M16 “Quit with an argument not allowed” and M17 “Quit with an argument required.” For further details, see $ECODE and the ISO 11756-1999 standard M programming language error messages listing in the Caché Error Reference.

QUIT Without an Argument

QUIT without an argument exits from the current context without returning a value. It is used to terminate the execution level of a process started with a DO or XECUTE command, or to exit from a FOR, DO WHILE, or WHILE flow-of-control loop.

If DO, XECUTE, or an (unnested) flow-of-control loop command was issued from the Terminal prompt, QUIT returns control to the Terminal. If the terminated operation contains any NEW commands before QUIT, QUIT automatically KILLS the affected variables and restores them to their original values.
**QUIT With an Argument**

QUIT expression terminates a user-defined function or an object method and returns the value resulting from the specified expression. QUIT with an argument cannot be used to exit a routine from within a **FOR, DO WHILE, or WHILE** command loop. QUIT with an argument also cannot be used to exit from within a **TRY** block or a **CATCH** block.

If an argumented QUIT is invoked inside a subroutine, one of the following occurs:

- If an argumented QUIT is invoked inside a subroutine (instead of a function), the QUIT argument is evaluated (which may produce side effects or throw an error) and the argument result is discarded. Execution returns to the caller of the subroutine.
- If the subroutine was called by a **DO** command and is in the scope of that **DO** argument, then the QUIT command evaluates its argument (and any side effects of that evaluation occur), but it does not return the argument. For example, a subroutine called by **DO** that concludes with QUIT 4/0 generates a <DIVIDE> error. The same behavior occurs if the subroutine was called by **DO** terminated by an argumented QUIT, and the subroutine is terminated by an **$ETRAP**.

**At the Terminal Prompt**

Issuing a **QUIT** at the Terminal prompt clears entities from the program stack.

- **QUIT n** clears the specified number of levels from the program stack. You can use the **$STACK** special variable to determine the current number of levels on the program stack.
  
  You can specify n as a literal, a variable, or an arithmetic expression. If n is larger than the current number of levels, all levels are cleared from the program stack and no error is issued. A fractional number for n is truncated to its integer value. If n resolves to a negative non-zero integer, **QUIT** clears all levels from the program stack. If n resolves to 0 (zero), the system generates a <COMMAND> error.

- **QUIT** clears all levels from the program stack.

The **$STACK** special variable contains the current number of context frames on the call stack. This number is also displayed as part of the Terminal prompt.

The following example use **QUIT** with an integer argument to clear the specified number of levels from the program stack, then uses argumentless **QUIT** to clear all remaining levels:

```objectscript
USER 5f0>QUIT 1
USER 4d0>QUIT 2
USER 2d0>QUIT
USER>
```

For further details see **Processing Errors at the Terminal Prompt** in the “Error Processing” chapter of *Using Caché ObjectScript*.

**Arguments**

**pc**

An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to **Command Postconditional Expressions** in *Using Caché ObjectScript*. If the **QUIT** command takes no other arguments, there must be two or more spaces between the postconditional and the next command following it on the same line.
**expression**

Any valid ObjectScript expression. It can be used only within a user-defined function to return the evaluated result to the calling routine.

**Examples**

The following two examples contrast QUIT and RETURN behavior when issued from within a flow-of-control structure. **QUIT** exits the FOR loop, continuing with the remainder of MySubroutine before returning to MyRoutine. **RETURN** exits MySubroutine, returning to MyRoutine.

```objectscript
MyMain
    WRITE "In the main routine",!
    DO MySubroutine
        WRITE "Returned to main routine",!
        QUIT
    MySubroutine
        WRITE "In MySubroutine",!
        FOR i=1:1:5 {
            WRITE "FOR loop: ", i,!
            IF i=3 QUIT
            WRITE "  loop again",!
        }
        WRITE "MySubroutine line displayed with QUIT",!
        QUIT
MyMain
    WRITE "In the main routine",!
    DO MySubroutine
        WRITE "Returned to main routine",!
        QUIT
    MySubroutine
        WRITE "In MySubroutine",!
        FOR i=1:1:5 {
            WRITE "FOR loop: ", i,!
            IF i=3 RETURN
            WRITE "  loop again",!
        }
        WRITE "MySubroutine line not displayed with RETURN",!
        QUIT
```

In the following example, execution of the first QUIT command is controlled by a postconditional (:x>46). If the randomly generated number is greater than 46 Caché does not perform the Cube procedure; the first QUIT takes the postconditional, returning a string to the calling routine as `num`. If the randomly generated number is less than or equal to 46 the second QUIT returns the results of the expression `x*x*x` as `num`.

```objectscript
Main
    SET x = $RANDOM(99)
    WRITE "Number is: ", x,!
    SET num=##Cube(x)
    WRITE "Cube is: ", num
    QUIT
Cube(x)
    QUIT:x>46 "a six-digit number."
    WRITE "Calculating the cube",!
    QUIT x*x*x
```

The following two examples contrast QUIT and RETURN behavior with TRY and CATCH. The TRY block attempts a divide-by-zero operation, invoking the CATCH block; this CATCH block contains a nested TRY block which is exited by either a QUIT or a RETURN. (For the purpose of demonstration, these programs do not include the recommended code (QUIT or RETURN) to prevent “fall-through”.)

**QUIT** exits the nested TRY block to the enclosing block, continuing execution with the remainder of the CATCH block. When it completes the CATCH block it execute the fall-through line outside of the TRY/CATCH structures:
TRY {
  WRITE "In the TRY block",!
  SET x = 5/0
  WRITE "This line should never display"
}
CATCH exp1 {
  WRITE "In the CATCH block",!
  WRITE "Error Name: ", $ZCVT(exp1.Name, "O", "HTML"),!
  TRY {
    WRITE "In the nested TRY block",!
    QUIT
  }!
  CATCH exp2 {
    WRITE "In the nested CATCH block",!
    WRITE "Error Name: ", $ZCVT(exp1.Name, "O", "HTML"),!
  }
  WRITE "QUIT displays this outer CATCH block line",!
}
WRITE "QUIT displays this outer CATCH block line",!
WRITE "fall-through at the end of the program"

RETURN exits the routine. It therefore exits the nested TRY block and any enclosing blocks, and does not execute the fall-through line outside of the TRY/CATCH structures:

TRY {
  WRITE "In the TRY block",!
  SET x = 5/0
  WRITE "This line should never display"
}
CATCH exp1 {
  WRITE "In the CATCH block",!
  WRITE "Error Name: ", $ZCVT(exp1.Name, "O", "HTML"),!
  TRY {
    WRITE "In the nested TRY block",!
    RETURN
  }!
  CATCH exp2 {
    WRITE "In the nested CATCH block",!
    WRITE "Error Name: ", $ZCVT(exp1.Name, "O", "HTML"),!
  }
  WRITE "RETURN does not display this outer CATCH block line",!
}
WRITE "QUIT displays this outer CATCH block line",!
WRITE "fall-through at the end of the program"

In this example, the argument of the QUIT command is an object method. Caché terminates execution of the method and returns control to the calling routine.

QUIT inv.TotalNum()

Notes

QUIT Restores Variables

If a terminated process contains any NEW commands before QUIT, QUIT automatically KILLS the affected variables and restores them to their original values.

QUIT and Flow-of-Control Structures

A QUIT can be used to break out of a FOR loop, a DO WHILE loop, or a WHILE loop. Execution continues with the command that follows the end of the loop structure. If these loop structures are nested, the QUIT breaks out of the inner loop from which it was called to the next enclosing outer loop.

If a QUIT command is encountered within an IF code block (or an ELSEIF code block or an ELSE code block) the QUIT behaves as a regular QUIT command, as if the code block did not exist:

- If the IF code block is nested within a loop structure (such as a FOR code block), the QUIT exits the loop structure block and continues execution with the command that follows the loop structure code block.
- If the IF code block is not nested within a loop structure, the QUIT terminates the current routine.

A QUIT can be used to break out of a legacy DO structure (an argumentless DO followed by lines of code each of which is prefixed by a period). Execution continues with the command that follows the argumentless DO structure code block.
This argumentless **DO** syntax is considered obsolete and should not be used for new coding. For further details, see Argumentless DO (legacy version).

**Use with Indefinite FOR Loop**

An indefinite **FOR** loop is a **FOR** without an argument; unless broken out of, it will loop infinitely. To control an indefinite **FOR** loop, you must include within the loop structure either a **QUIT** with a postconditional, or an **IF** command that invokes a **QUIT**. The **QUIT** within the **IF** terminates the enclosing **FOR** loop. Without one of these **QUITs**, the loop will be an infinite loop and will execute endlessly.

In the following example, the indefinite **FOR** loop is exited using a **QUIT** with a postconditional. The loop continues to execute as long as the user enters some value in response to the "Text =" prompt. Only when the user enters a null string (that is, just presses Return or Enter) is **QUIT** executed and the loop terminated.

```objectscript
Main
   FOR { 
      READ !,"Text =",var1
      QUIT:var1=""  DO Subr1
   }
Subr1
   WRITE "Thanks for the input", !
   QUIT
```

This command requires at least two spaces between the postconditional on the **QUIT** command and the following **DO** command on the same line. This is because Caché treats postconditionals as **command modifiers**, not as arguments.

**Implicit QUIT**

In the following cases, a **QUIT** command is not required, because Caché automatically issues an implicit **QUIT** to prevent execution "falling through" to a separate unit of code.

- Caché executes an implicit **QUIT** at the end of a routine.
- Caché executes an implicit **QUIT** when it encounters a **label** with parameters. A **label** with parameters is defined as one with parentheses, even if the parentheses contain zero parameters. All procedures begin with a **label** with parameters (even if no parameters are defined). Many subroutines and functions also have a **label** with parameters.

You can, of course, code an explicit **QUIT** in any of these circumstances.

**Behavior with DO**

When encountered in a subroutine called by the **DO** command, **QUIT** terminates the subroutine and returns control to the command following the **DO** command.

**Behavior with XECUTE**

When encountered in a line of code that is being executed, **QUIT** terminates execution of the line and returns control to the command following the **XECUTE** command. No argument is allowed.

**Behavior with User-Defined Functions**

When encountered in a user-defined function, **QUIT** terminates the function and returns the value that results from the specified expression. The expression argument is required.

In their use, user-defined functions are similar to **DO** commands with parameter passing. They differ from such **DO** commands, however, in that they return the value of an expression directly, rather than through a variable. To invoke a user-defined function, use the form:

```objectscript
$\$name(parameters)
```

where **name** is the name of the function. It can be specified as **label**, ~**routine**, or **label**~**routine**.
parameters is a comma-separated list of parameters to be passed to the function. The label associated with the function must also have a parameter list. The parameter list on the invoked function is known as the actual parameter list. The parameter list on the function label is known as the formal parameter list.

In the following example, the FOR loop uses a READ command to first acquire the number to be squared and store it in the num variable. (Note the two spaces after the argumentless FOR and the postconditional QUIT.) It then uses a WRITE command to invoke the Square standard function, with num specified as the function parameter.

The only code for the function is the QUIT command followed by an expression to calculate the square. When it encounters the QUIT command, Caché evaluates the expression, terminates the function, and returns the resulting value directly to the WRITE command. The value of num is not changed.

```
Test  WRITE "Calculate the square of a number",!
    FOR   {
        READ !,"Number:",num QUIT:num="
        WRITE !,$$Square(num),!
        QUIT
    }

Square(val)
    QUIT val*val
```

**Using QUIT for Program Readability**

Caché executes an implicit QUIT at the end of each routine, but you can include it explicitly to improve program readability.

**See Also**

- DO command
- DO WHILE command
- FOR command
- NEW command
- RETURN command
- WHILE command
- XECUTE command
- ZQUIT command
- $ESTACK special variable
- $QUIT special variable
- $STACK special variable
READ

Accepts input and stores it in a variable.

```
READ:pc readargument,...
R:pc readargument,...
```

where `readargument` is:

```
fchar
prompt
variable:timeout
*variable:timeout
variable#length:timeout
```

### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pc</code></td>
<td><code>Optional</code> — A postconditional expression.</td>
</tr>
</tbody>
</table>
| `fchar`  | `Optional` — One or more format control characters. Permitted characters are !, #, ?, and /.
| `prompt` | `Optional` — A string literal that provides a prompt or message for user input. Enclose in quotation marks. |
| `variable` | The variable to receive the input data. Can be a local variable, a process-private global, or a global. May be unsubscripted or subscripted. |
| `length` | `Optional` — The number of characters to accept, specified as an integer, or an expression or variable that evaluates to an integer. The preceding `#` symbol is mandatory. |
| `timeout` | `Optional` — The number of seconds to wait for the request to succeed, specified as an integer. Fractional seconds are truncated to the integer portion. The preceding colon (`:`) is mandatory. If omitted, Caché waits indefinitely. |

You can specify more than one `fchar` or `prompt` argument by separating the arguments with commas.

### Description

The `READ` command accepts input from the current device. The current device is established using the `OPEN` and `USE` commands. The `$IO` special variable contains the device ID of the current device. By default, the current device is the user terminal.

The `READ` command suspends program execution until it either receives input from the current device or times out. For this reason, the `READ` command should not be used in programs executed as background (non-interactive) jobs if the current device is the user terminal.

The `variable` argument receives the input characters. `READ` first defines `variable`, if it is undefined, or clears it if it has a previous value. Therefore, if no data is input for `variable` (for example, if the `READ` times out before any characters are entered) `variable` is defined and contains the null string. This is also true if the only character entered is a terminator character (for example, pressing the <Enter> key from the user terminal). For the effects of an interrupt (for example, <Ctrl-C>) see below.

Note that for fixed-length and variable-length reads, `variable` does not store the terminator character used to terminate the read operation. Single-character reads handle `variable` differently; for single-character read use of `variable`, see below.
If you specify the optional timeout value, a READ can time out before all characters are input. If a READ times out, those characters input before the timeout are stored in variable. Entering a terminator character is not necessary in this case; the characters entered before the timeout are transferred to variable, and the READ terminates, setting $TEST equal to 0.

There are three types of READ operations: variable-length read, fixed-length read, and single-character read. All of these can be specified with or without a timeout argument. A single READ command can include multiple READ operations in any combination of these three types. Each read operation is executed independently in left-to-right sequence. A READ command can also contain any number of comma-separated prompt and fchar arguments.

The three types of READ operations are as follows:

- A variable-length read has the following format:
  ```plaintext
  READ variable
  ```
  A variable-length read accepts any number of input characters and stores them in the specified variable. Input is concluded by a terminator character. For a terminal, this terminator is usually supplied by pressing the <Enter> key. The input characters, but not the terminator character, are stored in variable.

- A fixed-length read has the following format:
  ```plaintext
  READ variable#length
  ```
  A fixed-length read accepts a maximum of length input characters and stores them in the specified variable. Input concludes automatically when the specified number of characters is input, or when a terminator character is encountered. For example, entering two characters in a four-character fix-length read, and then pressing the <Enter> key. The input characters, but not the terminator character (if any), are stored in variable.

- A single-character read has the following format:
  ```plaintext
  READ *variable
  ```
  A single-character read accepts a single input character and stores its ASCII numeric value equivalent in the specified variable. It stores the character itself in the $ZB and $KEY special variables. Input concludes automatically when a single character is input. A terminator character is considered a single-character input, and is stored as such. If the optional timeout argument is specified, and a timeout occurs, the timeout sets variable to –1.

### Arguments

- **pc**
  An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

- **fchar**
  Any of the following format control codes. When used with user input from the keyboard, these controls determine where a specified prompt or the user input area will appear on the screen.
  
  - ! starts a new line. You may specify multiple exclamation points
  - # starts a new page. On a terminal, it clears the current screen and starts at the top of the new screen.
  - ?n positions at the nth column location, where n is a positive integer.

  /keyword(parameters) A device control mnemonic. Performs a device-specific operation, such as positioning the cursor on a video terminal or rewinding a magnetic tape. The slash character (/) is followed by a keyword, which is optionally followed by one or more parameters enclosed in parentheses. Multiple parameters are separated by commas. The keyword is an entry point label into the current device’s mnemonic space routine.

  You can establish the default mnemonic space for a device type in either of the following ways:
Go to the Management Portal, select [System] > [Configuration] > [Device Settings] > [IO Settings]. View and edit the File, MagTape, Other, or Terminal mnemonic space setting.

Include the /mnemonic space parameter in the OPEN or USE command for the device.

You can specify multiple format controls. For example: #!!!?20 means to start at the top of a new page (or screen), go down three lines, and then position to column 20. You can intersperse format control characters with other comma-separated READ arguments. For example:

```objectscript
READ #!,"Please enter","!,"your name: ",,","THANK YOU"
```

Displays something like the following:

```objectscript
> 
Please enter your name: FRED 
THANK YOU 
>
```

**prompt**

A string literal that provides a prompt or message for user input with the terminal keyboard. Generally, a prompt argument is followed by a variable, so that the user input area follows the displayed literal. You can specify a multi-line prompt or message by using a comma-separated series of prompt and fchar arguments.

**variable**

The local variable, process-private global, or global that is to receive the input data. It can be either unsubscripted or subscripted. If a specified variable does not already exist, Caché defines it at the beginning of the READ operation. If a specified variable is defined and has a value, Caché clears this value at the beginning of the READ operation.

When you input characters, they are stored in variable as they are input. If the optional timeout argument is specified, and the read operation is interrupted by a timeout, the characters typed up to that point are stored in variable. (However, note the behavior of variable upon a encountered a <Ctrl-C> interrupt, as described below.)

Nonprinting characters (such as <Tab>) are stored in variable. A terminator character can be used to conclude any type of read operation. For example, from a terminal, you press the <Enter> key to conclude a read operation. This terminator character is not stored in variable for a variable-length or fixed-length read. The terminator character is stored in variable for a single-character read.

**length**

A positive integer specifying the maximum number of characters to accept for a fixed-length read. The READ completes either when the specified number of characters is input, or when it encounters a terminator character. This argument is optional, but if specified the preceding # symbol is required.

Specifying zero or a negative number results in a <SYNTAX> error. However, leading zeros and the fractional portion of a number are ignored, and the integer portion used. You can specify length as a variable or an expression that resolve to an integer.

Note that READ a#1 and READ *a can both be used to input a single character. However the value stored in variable is different: a#1 stores the input character in variable a; *a stores the ASCII numeric value for the input character in variable a; both store the input character in the $ZB special variable. These two types of single-character input also differ in how they handle terminator characters and how they handle a timeout.

**timeout**

The number of seconds to wait for the request to succeed. This argument is optional, but if specified, the preceding colon is required. You must specify timeout as an integer or an expression that evaluates to an integer. The timeout argument sets the $TEST special variable as follows:
• **READ** with timeout argument completes successfully (does not time out): $TEST$ set to 1 (TRUE).

• **READ** with timeout argument times out: $TEST$ set to 0 (FALSE).

• **READ** with *no* timeout argument: $TEST$ remains set to its previous value.

Note that $TEST$ can also be set by the user, or by a **LOCK**, **OPEN**, or **JOB** timeout.

If the timeout period expires before the **READ** completes and some characters have been input (for a variable-length or fixed-length reads) the input characters are stored in *variable*. If no characters have been input (for a variable-length or fixed-length reads), Caché defines *variable* (if necessary) and sets it to the null string. If no character has been input for a single-character **READ**, Caché defines *variable* (if necessary) and sets it to –1.

### Examples

The following example uses the variable-length form of **READ** to acquire any number of characters from the user. The format control `!` starts the prompt on a new line.

```objectscript
READ !,”Enter your last name: “, lname
```

The following example uses the single-character form of **READ** to acquire one character from the user and store it as its ASCII numeric value.

```objectscript
READ !,”Enter option number (1,2,3,4): “, *opt
WRITE !,”ASCII value input= “, opt
WRITE !,”Character input= “, $KEY
```

The following example uses the fixed-length form of **READ** to acquire exactly three characters from the user.

```objectscript
READ !,”Enter your 3-digit area code: “, area#3
```

The following example prompts for three parts of a name: a fixed-length given name (*gname*) of up to 12 characters, a fixed-length (one-character) middle initial (*iname*), and a family name (*fname*) of any length. The *gname* and *iname* variables are coded to time out after 10 seconds:

```objectscript
READ "Given name: “, gname#12:10, !,
"Middle initial: “, iname#1:10, !,
"Family name: “, fname
WRITE $TEST
```

A timeout of a read operation causes the **READ** command to proceed to the next read operation. The first two read operations set $TEST$ whether or not they time out. The third read operation does not set $TEST$, so the value of $TEST$ in this example reflects the result (success or timeout) of the second read operation.

The following example uses indirection to dynamically change the prompt associated with the **READ** command:

```objectscript
PromptChoice
READ "Type 1 for numbers or 2 for names: “, p, #!!!!
IF p'=1,p'=2 {WRITE !,"Invalid input" RETURN }
ELSE {DO DataInput(p) }
DataInput(dtype)
SET MESS(1)="**ENTER A NUMBER:**
SET MESS(2)=**ENTER A NAME:**
SET x=1
READ !,@MESS(dtype), val(x)
IF val(x)="" {WRITE !,"Goodbye" RETURN }
ELSE {
  IF dtype=1,1=$ISVALIDNUM(val(x)) { WRITE !,"You input number: “,val(x), ! }
  ELSEIF dtype=2 { WRITE !,"You input string: “,val(x), ! }
  ELSE { WRITE !,"That is not a number", ! }
  SET x=x+1
  DO DataInput(dtype)
}
```

The following example sets the length of a fixed-length read based on the number of digits of the first number input:
READ "ENTER LARGEST INTEGER (and press Return): ",val(1)
SET ibuf=$LENGTH(val(1))
WRITE !,"Your largest number is: ",val(1),!
DO OtherNums(ibuf)

OtherNums(digits)
SET x=2
READ !,"ENTER NEXT INTEGER: ",val(x)#digits
IF val(x)="" { WRITE !,"Goodbye" RETURN |
ELSEIF val(x)>val(1) { WRITE !,"Number is too big",!
DO OtherNums(digits) }
ELSE { WRITE !,"You input: ",val(x),!
SET x=x+1
DO OtherNums(digits) }

Notes

READ Uses the Current Device

READ inputs character-oriented data from the current I/O device. You must open a device with the OPEN command, then establish it as the current device with the USE command. Caché maintains the current device ID in the $IO special variable.

While the most common use for READ is to acquire user input from the keyboard, you can also use it to input characters from any byte-oriented device, such as a magnetic tape, a sequential disk file, or a communications buffer.

Read Line Recall

Read line recall mode permits a READ command on a terminal device to receive as its input a previously input line. This recalled input line can then be edited. The user must interactively conclude the input of a recalled line in the same way that user-specified input is concluded. Caché supports read line recall for both variable-length terminal reads (READ variable) and fixed-length terminal reads (READ variable#length). Caché does not support read line recall for single-character terminal reads (READ *variable). To activate read line recall for the current process, use the LineRecall() method of the %SYSTEM.Process class. To set the system-wide read line recall default, use the LineRecall property of the Config.Miscellaneous class. You can also set the OPEN and USE protocols for terminals, as described in the Terminal I/O chapter of the Caché I/O Device Guide.

READ Terminators

Caché terminates a read operation when the input string reaches the specified length (for single-character READ and fixed-length READ). For a variable-length READ, Caché terminates reading if the input string reaches the maximum string length for the current process.

Caché also terminates reading when it encounters certain terminator characters. The terminators are determined by the device type. For example, with terminals, the default terminators are RETURN (also known as the <Enter> key) (ASCII 13), LINE FEED (ASCII 10), and ESCAPE (ASCII 27).

You can modify the terminator default when you issue an OPEN or USE command for a device. OPEN and USE allow you to specify a terminator parameter value. See the Terminal I/O chapter of the Caché I/O Device Guide for OPEN and USE protocols for terminals. See the I/O Devices and Commands chapter of the Caché I/O Device Guide for details about terminators based on device type.

Caché does not store the input terminator with the input value for variable-length and fixed-length reads; it records it in the $KEY and $ZB special variables. Caché does store the input terminator (if specified) as the input value for a single-character read.

Timeout and the $ZA, $ZB, and $TEST Special Variables

Caché records the completion status of a READ in the $TEST, $ZA, and $ZB special variables, as follows:
**$ZB and $KEY**

The `$ZB` and `$KEY` special variables return the exact same value for every type of read, except one. When you perform a fixed-length read and input the specified number of characters, the `READ` completes without a terminator. In this case, `$ZB` contains the last character input (the terminating character), and `$KEY` contains the null string (there being no terminator character).

**Interrupts**

If there is a pending `CTRL-C` interrupt when `READ` is invoked, `READ` dismisses this interrupt before reading from the terminal.

If a read in progress is interrupted by a `CTRL-C` interrupt, `variable` reverts to its previous state. For example, if you input several characters for a read operation, and then issue a `CTRL-C`, `variable` reverts to its state before the read operation. That is, if it was undefined, it remains undefined; if it had a previous value, it contains the previous value. This behavior is completely different than a read operation that times out. A read timeout retains the new state of `variable`, including any characters input before the timeout occurred. If a `READ` command contains multiple read operations, the interrupt affects only the read operation in progress. To commit or revert multiple read operations as a unit, use transaction processing.

For information on enabling and disabling `CTRL-C` interrupts, refer to the `BREAK flag` command and the `$ZJOB` special variable.

**Reading from Non-Keyboard Devices**

As described earlier, `READ` can be used to acquire input from any character-oriented device. This includes devices such as magnetic tapes and sequential disk files, as well terminal keyboards. However, you must first establish the device to read from as the current device with the `OPEN` and `USE` commands.

With non-keyboard devices, you can use any of the three available forms (variable-length, single-character, and fixed-length). The choice of which form to use in any given case depends on the type of terminator available. With fixed-length `READ`, Caché treats terminators encountered within the input string the same as any other character.
For example, if you are reading from a device that presents data in a line-oriented format with CARRIAGE RETURN/LINE FEED as the line terminator, you can use the variable-length form. In this case, Caché reads each line into variable in its entirety, terminating input only when it reaches the Return (ASCII code 13) at the end of the line. (Remember, from the user input examples shown previously, that <Return> is the input terminator.)

On the other hand, if you are reading from a magnetic tape that presents records as a series of fixed-length fields, you would use the fixed-length (variable#length) form. For example, assume that you have a mag tape that uses a record format consisting of four fields of up to 8, 12, 4, and 6 characters, respectively. You might use code similar to the following to read in the data:

```objectscript
READ field1#8,field2#12,field3#4,field4#6
```

In this case, the #n value sets the input terminator for each field.

Which terminator is used for a given device can be set by the device parameters that you specify for that device on the OPEN or USE command.

When reading block-oriented data from magnetic tape, the $ZB special variable contains the number of bytes remaining in the I/O buffer. Its function is entirely different than its use when reading character-oriented data. $ZB does not contain the read terminator character or the last input character during block-oriented I/O. Refer to $ZB for further details.

**Reading Nonprinting Characters**

Nonprinting characters are characters outside the standard range of ASCII printable characters. In other words, they are characters whose ASCII codes are less than ASCII 32 or greater than ASCII 127. They are characters that have no single key equivalents on a standard keyboard.

The characters whose codes are less than ASCII 32 are usually used for control operations. They can be entered only in conjunction with the Ctrl key. For example, ETX (ASCII 3) is entered as <Ctrl-C> and is used to assert a BREAK when entered from the keyboard.

The characters whose codes are greater than ASCII 127 are usually used for graphics operations. As a rule, they cannot be entered from the keyboard, but they can be read from other types of devices. For example, ASCII 179 produces the vertical line character.

You can use the READ command to input nonprinting characters as well as the standard ASCII printable characters. However, you must include code to handle the escape sequence used for each such character. An escape sequence is a sequence of characters that starts with the Esc character (ASCII 27). For example, you can code a READ so that the user is allowed to press a function key as a valid input response. Pressing the function key produces an escape sequence that can be different for different types of terminals.

ObjectScript supports input of escape sequences by storing them in the $ZB and $KEY special variables, rather than in the specified variable. For example, for a function key press, Caché stores the Esc code (ASCII 27) in $ZB and $KEY. To handle an escape sequence, you must include code that tests the current value in $ZB or $KEY after each READ because subsequent reads update these special variables and overwrite any previous value. ($ZB and $KEY are very similar, but not identical; see $KEY for details.) To display a nonprinting character, such as an escape sequence, use the ZZDUMP command or the $ASCII function.

**Sequential File End-of-File**

The behavior of READ when encountering an end-of-file for a sequential file depends on the system-wide default. Go to the Management Portal, select [System] > [Configuration] > [Compatibility Settings]. View and edit the current setting of SetZEOF. This option controls the behavior when Caché encounters an unexpected end-of-file when reading a sequential file. When set to “true”, Caché sets the $ZEOF special variable to indicate that you have reached the end of the file. When set to “false”, Caché issues an <ENDOFFILE> error. The default is “false”.

To change this end-of-file behavior for the current process, use the SetZEOF() method of the %SYSTEM.Process class. To set the system-wide default for end-of-file behavior, use the SetZEOF property of the Config.Miscellaneous class.
Default Record Length

If the number of characters to read is not specified, Caché assumes a default length of 32,767 characters, regardless of whether long strings are enabled or not.

See Also

- OPEN command
- WRITE command
- ZZDUMP command
- USE command
- $KEY special variable
- $TEST special variable
- $ZA special variable
- $ZB special variable
- $ZEOF special variable
- Terminal I/O in Caché I/O Device Guide
- Magnetic Tape I/O in Caché I/O Device Guide
- Sequential File I/O in Caché I/O Device Guide
**RETURN**

Terminates execution of a routine.

**RETURN**:pc expression  
**RET**:pc expression

### Arguments

<table>
<thead>
<tr>
<th>pc</th>
<th>Optional — A postconditional expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>Optional — An ObjectScript expression.</td>
</tr>
</tbody>
</table>

### Description

The `RETURN` command is used to terminate execution of a routine. In many contexts it is a synonym for the `QUIT` command. `RETURN` and `QUIT` differ when issued from within a `FOR`, `DO WHILE`, or `WHILE` flow-of-control structure, or a `TRY` or `CATCH` block.

- You can use `RETURN` to terminate execution of a routine at any point, including from within a `FOR`, `DO WHILE`, or `WHILE` loop or nested loop structure. `RETURN` always exits the current routine, returning to the calling routine or terminating the program if there is no calling routine. `RETURN` always behaves the same, regardless of whether it is issued from within a code block. This includes a `TRY` block or a `CATCH` block.

- In contrast, `QUIT` exits only the current structure when issued from within a `FOR` loop, a `DO WHILE` loop, a `WHILE` loop, or a `TRY` or `CATCH` block. `QUIT` exits the structure block and continues execution of the current routine with the next command outside of that structure block. `QUIT` exits the current routine when issued outside of a block structure or from within an `IF`, `ELSEIF`, or `ELSE` code block.

The `RETURN` command has two forms:

- Without an argument
- With an argument

A postconditional is not considered an argument. The `$QUIT` special variable indicates whether or not an argumented `RETURN` command is required to exit from the current context. Two error codes are provided for this purpose: M16 “Quit with an argument not allowed” and M17 “Quit with an argument required.” For further details, see `$ECODE` and the ISO 11756-1999 standard M programming language error messages listing in the *Caché Error Reference*.

### RETURN Without an Argument

`RETURN` without an argument exits from the current context without returning a value. It is used to terminate the execution level of a process started with a `DO` or `XECUTE` command.

If `DO` or `XECUTE` is invoked from the Terminal prompt, `RETURN` returns control to the Terminal prompt. If the terminated process contains any `NEW` commands before `RETURN`, `RETURN` automatically `KILLs` the affected variables and restores them to their original values.

### RETURN With an Argument

`RETURN` *expression* terminates a user-defined function or an object method and returns the value resulting from the specified expression. `RETURN` with an argument can be used to exit a routine from within a `FOR`, `DO WHILE`, or `WHILE` command loop, or from within a `TRY` block or a `CATCH` block.

If an argumented `RETURN` is invoked inside a subroutine, one of the following occurs:
If an argumented RETURN is invoked inside a subroutine (instead of a function), the RETURN argument is evaluated (which may produce side effects or throw an error) and the argument result is discarded. Execution returns to the caller of the subroutine.

If the subroutine was called by a DO command and is in the scope of that DO argument, then the RETURN command evaluates its argument (and any side effects of that evaluation occur), but it does not return the argument. For example, a subroutine called by DO that concludes with RETURN 4/0 generates a <DIVIDE> error. The same behavior occurs if the subroutine was called by DO terminated by an argumented RETURN, and the subroutine is terminated by an SETRAP.

Arguments

pc
An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript. If the RETURN command takes no other arguments, there must be two or more spaces between the postconditional and the next command following it on the same line.

eexpression
Any valid ObjectScript expression. It can be used only within a user-defined function to return the evaluated result to the calling routine.

Examples

The following two examples contrast RETURN and QUIT behavior when issued from within a flow-of-control structure. RETURN exits MySubroutine, returning to MyRoutine. QUIT exits the FOR loop, continuing with the remainder of MySubroutine before returning to MyRoutine:

MyMain
WRITE "In the main routine",!
DO MySubroutine
  WRITE "Returned to main routine",!
  QUIT
MySubroutine
WRITE "In MySubroutine",!
FOR i=1:1:5 {
  WRITE "FOR loop:",i,!
  IF i=3 RETURN
  WRITE "  loop again",!
}
WRITE "MySubroutine line not displayed with RETURN",!
QUIT

MyMain
WRITE "In the main routine",!
DO MySubroutine
  WRITE "Returned to main routine",!
  QUIT
MySubroutine
WRITE "In MySubroutine",!
FOR i=1:1:5 {
  WRITE "FOR loop:",i,!
  IF i=3 QUIT
  WRITE "  loop again",!
}
WRITE "MySubroutine line displayed with QUIT",!
QUIT

In the following example, execution of the first RETURN command is controlled by a postconditional (:x>46). If the randomly generated number is greater than 46 Caché does not perform the Cube procedure; the first RETURN takes the postconditional, returning a string to the calling routine as num. If the randomly generated number is less than or equal to 46 the second RETURN returns the results of the expression x*x*x as num.
Main
SET x = $RANDOM(99)
WRITE "Number is: ",x,!
SET num=$$Cube(x)
WRITE "Cube is: ",num
QUIT
Cube(x) RETURN:x>46 "a six-digit number."
WRITE "Calculating the cube",!
RETURN x*x*x

The following two examples contrast QUIT and RETURN behavior with TRY and CATCH. The TRY block attempts a divide-by-zero operation, invoking the CATCH block; this CATCH block contains a nested TRY block which is exited by either a QUIT or a RETURN. (For the purpose of demonstration, these programs do not include the recommended code (QUIT or RETURN) to prevent “fall-through”.)

RETURN exits the routine. It therefore exits the nested TRY block and any enclosing blocks, and does not execute the fall-through line outside of the TRY/CATCH structures:

TRY {
    WRITE "In the TRY block",!
    SET x = 5/0
    WRITE "This line should never display"
}
CATCH exp1 {
    WRITE "In the CATCH block",!
    WRITE "Error Name: ",$ZCVT(exp1.Name,"O","HTML"),!
    TRY {
        WRITE "In the nested TRY block",!
        RETURN
    }
    CATCH exp2 {
        WRITE "In the nested CATCH block",!
        WRITE "Error Name: ",$ZCVT(exp2.Name,"O","HTML"),!
    }
    WRITE "RETURN does not display this outer CATCH block line",!
}
WRITE "fall-through at the end of the program"

QUIT exits the nested TRY block to the enclosing block, continuing execution with the remainder of the CATCH block. When it completes the CATCH block it execute the fall-through line outside of the TRY/CATCH structures:

TRY {
    WRITE "In the TRY block",!
    SET x = 5/0
    WRITE "This line should never display"
}
CATCH exp1 {
    WRITE "In the CATCH block",!
    WRITE "Error Name: ",$ZCVT(exp1.Name,"O","HTML"),!
    TRY {
        WRITE "In the nested TRY block",!
        QUIT
    }
    CATCH exp2 {
        WRITE "In the nested CATCH block",!
        WRITE "Error Name: ",$ZCVT(exp2.Name,"O","HTML"),!
    }
    WRITE "QUIT displays this outer CATCH block line",!
}
WRITE "fall-through at the end of the program"

In this example, the argument of the RETURN command is an object method. Caché terminates execution of the method and returns control to the calling routine.

RETURN inv.TotalNum()

Notes

**RETURN Restores Variables**

If a terminated process contains any NEW commands before RETURN, RETURN automatically KILLS the affected variables and restores them to their original values.
Implicit RETURN

In the following cases, a RETURN command is not required, because Caché automatically issues an implicit RETURN to prevent execution "falling through" to a separate unit of code.

- Caché executes an implicit RETURN at the end of a routine.
- Caché executes an implicit RETURN when it encounters a label with parameters. A label with parameters is defined as one with parentheses, even if the parentheses contain zero parameters. All procedures begin with a label with parameters (even if no parameters are defined). Many subroutines and functions also have a label with parameters.

You can, of course, code an explicit RETURN in any of these circumstances.

Behavior with DO

When encountered in a subroutine called by the DO command, RETURN terminates the subroutine and returns control to the command following the DO command.

Behavior with XECUTE

When encountered in a line of code that is being executed, RETURN terminates execution of the line and returns control to the command following the XECUTE command. No argument is allowed.

Behavior with User-Defined Functions

When encountered in a user-defined function, RETURN terminates the function and returns the value that results from the specified expression. The expression argument is required.

In their use, user-defined functions are similar to DO commands with parameter passing. They differ from such DO commands, however, in that they return the value of an expression directly, rather than through a variable. To invoke a user-defined function, use the form:

$\text{name(parameters)}$

where \text{name} is the name of the function. It can be specified as \text{label}, \text{^routine}, or \text{label^routine}.

\text{parameters} is a comma-separated list of parameters to be passed to the function. The \text{label} associated with the function must also have a parameter list. The parameter list on the invoked function is known as the \text{actual parameter list}. The parameter list on the function label is known as the \text{formal parameter list}.

Clearing Levels from the Program Stack

Each invocation of RETURN removes a context frame from the call stack for your process. The $\text{STACK}$ special variable contains the current number of context frames on the call stack.

You can use RETURN from the Terminal prompt to clear some or all levels from the program stack. The following example clears the top two levels from the stack:

RETURN 2

The argumentless RETURN clears all the levels from the stack.

Using RETURN for Program Readability

Caché executes an implicit RETURN at the end of each routine, but you can include it explicitly to improve program readability.

See Also

- DO command
- DO WHILE command
ObjectScript Commands

- FOR command
- NEW command
- QUIT command
- WHILE command
- XECUTE command
- ZQUIT command
- $QUIT special variable
- $STACK special variable
**SET**

Assigns a value to a variable.

```
SET:pc setargument,...
S:pc setargument,...
```

where `setargument` can be:

```
variable=value
(variable-list)=value
```

### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pc</code></td>
<td>Optional — A postconditional expression.</td>
</tr>
<tr>
<td><code>variable</code></td>
<td>The variable to set to the corresponding value. <code>variable</code> can be a local variable, a process-private global, a global, an object property, or a special variable. (Not all special variables can be set by an application; see documentation of individual special variables.)</td>
</tr>
<tr>
<td><code>variable-list</code></td>
<td>A comma-separated list, enclosed in parentheses, that consists of one or more <code>variable</code> arguments. All of the <code>variable</code> arguments in <code>variable-list</code> are assigned the same value.</td>
</tr>
<tr>
<td><code>value</code></td>
<td>A literal value, or any valid ObjectScript expression that evaluates to a value. Can be a JSON object or JSON array.</td>
</tr>
</tbody>
</table>

### Description

The **SET** command assigns a value to a variable. It can set a single variable, or set multiple variables using any combination of two syntactic forms. It can assign values to variables by specifying a comma-separated list of `variable=value` pairs. For example:

```
SET a=1,b=2,c=3
WRITE a,b,c
```

There is no restriction in the number of assignments you can perform with a single invocation of **SET** `a=value,b=value,c=value,...`

If a specified variable does not exist, **SET** creates it and assigns the value. If a specified variable exists, **SET** replaces the previous value with the specified value. Because **SET** executes in left-to-right order, you can assign a value to a variable, then assign that variable to another variable:

```
SET a=1,b=a
WRITE a,b
```

A value can be a string, a numeric, a JSON object, a JSON array, or an expression that evaluates to one of these values. To define an “empty” variable, you can set the variable to the empty string ("" ) value.

### Setting Multiple Variables to the Same Value

You can use **SET** to assign the same value to multiple variables by specifying a comma-separated list of variables enclosed in parentheses. For example:

```
SET (a,b,c)=1
WRITE a,b,c
```

You can combine the two **SET** syntactic forms in any combination. For example:
The maximum number of assignments you can perform with a single invocation of \texttt{SET (a,b,c,...)=value} is 128. Exceeding this number results in a \texttt{<SYNTAX>} error.

You cannot use \texttt{SET (a,b,c,...)=value} syntax to assign a value to a function that uses relative offset syntax: the \texttt{$EXTRACT}, \texttt{$PIECE}, or \texttt{$LIST} function. In relative offset syntax, an asterisk represents the end of a string, and \texttt{*-n} and \texttt{*+n} represent a relative offset from the end of the string. You must use \texttt{SET a=value,b=value,c=value,...} syntax when setting one of these functions using relative offset.

### Arguments

**pc**

An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in \textit{Using Caché ObjectScript}.

**variable**

The variable to receive the value resulting from the evaluation of \texttt{value}. It can be a local variable, a process-private global, a global variable, or a special variable. A local variable, process-private global, or global variable can be either subscripted or unsubscripted (see SET and Subscripts for further details). A global variable can be specified with extended global reference (see Global Structure in \textit{Using Caché Globals}).

Local variables, process-private globals, and special variables are specific to the current process; they are mapped to be accessible from all namespaces. A global variable persists after the process that created it terminates.

A global is specific to the namespace in which it was created. By default, a \texttt{SET} assigns a global in the current namespace. You can use \texttt{SET} to define a global (\texttt{^myglobal}) in another namespace by using syntax such as the following: \texttt{SET ^["Samples"]myglobal="Ansel Adams"}. If you specify a nonexistent namespace, Caché issues a \texttt{<NAMESPACE>} error. If you specify a namespace for which you do not have privileges, Caché issues a \texttt{<PROTECT>} error, followed by the global name and database path, such as the following: \texttt{<PROTECT> ^myglobal,c:\intersystems\cache\mgr\}.

A variable can be a piece or segment of a variable as specified in the argument of a \texttt{$PIECE} or \texttt{$EXTRACT} function. A variable can be represented as an object property using \texttt{obj.property} or .. property syntax, or by using the \texttt{$PROPERTY} function. You can set an i%property instance property reference using the following syntax:

\begin{verbatim}
SET i%propname = "abc"
\end{verbatim}

\texttt{SET} accepts a variable name of any length, but it truncates a long variable name to 31 characters before assigning it a value. If a variable name is not unique within the first 31 characters this name truncation can cause unintended overwriting of variable values, as shown in the following example:

\begin{verbatim}
SET abcdefghijklmnopqrstuvwxyz2abc="30 characters"
SET abcdefghijklmnopqrstuvwxyz2abcd="31 characters"
SET abcdefghijklmnopqrstuvwxyz2abcde="32 characters"
SET abcdefghijklmnopqrstuvwxyz2abcdef="33 characters"
WRITE !,abcdefghijklmnopqrstuvwxyz2abc             // returns "30 characters"
WRITE !,abcdefghijklmnopqrstuvwxyz2abcd            // returns "31 characters"
WRITE !,abcdefghijklmnopqrstuvwxyz2abcde           // returns "32 characters"
WRITE !,abcdefghijklmnopqrstuvwxyz2abcdef           // returns "33 characters"
\end{verbatim}

Special variables are, by definition, set by system events. You can use \texttt{SET} to assign a value to certain special variables. However, most special variables cannot be assigned a value using \texttt{SET}. See the reference pages for individual special variables for further details.

Refer to the “Variables” chapter of \textit{Using Caché ObjectScript} for further details on variable types and naming conventions.
**value**

A literal value, a defined variable, or any valid ObjectScript expression. Usually a *value* is a numeric or string expression. A *value* can be a JSON object or JSON array.

- A numeric value is converted to canonical form before assignment: leading and trailing zeros, a plus sign or a trailing decimal point are removed. Conversion from scientific notation and evaluation of arithmetic operations are performed.
- A string value is enclosed in quotation marks. A string is assigned unchanged, except that doubled quotation marks within the string are converted to a single quotation mark. The null string (""") is a valid *value*.
- A numeric value enclosed in quotation marks is not converted to canonical form and no arithmetic operations are performed before assignment.
- If a relational or logical expression is used, Caché assigns the truth value (0 or 1) resulting from the expression.
- Object properties and object methods that return a value are valid expressions. Use the relative dot syntax (..) for assigning a property or method value to a variable.

**JSON Values**

You can use the `SET` command to set a variable to a JSON object or a JSON array. For a JSON object, the *value* is a JSON object delimited by curly braces. For a JSON array, the *value* is a JSON array delimited by square brackets.

Within these delimiters, the literal values are JSON literals, not ObjectScript literals. An invalid JSON literal generates a `<SYNTAX>` error.

- **String literal:** You must enclose a JSON string in double quotes. To specify certain characters as literals within a JSON string, you must specify the \ escape character, followed by the literal. If a JSON string contains a double-quote literal character this character is written as ". JSON string syntax provides escapes for double quote ("), backslash (\), and slash (/). Line space characters can also be escaped: backspace (\b), formfeed (\f), newline (\n), carriage return (\r), and tab (\t). Any Unicode character can be represented by a six character sequence: a backslash, followed by lowercase letter u, followed by four hexadecimal digits. For example, `\u0022` specifies a literal double quote character; `\u03BC` specifies the Greek lowercase letter Mu.

- **Numeric literal:** JSON does not convert numbers to ObjectScript canonical form. JSON has its own conversion and validation rules: Only a single leading minus sign is permitted; a leading plus sign is not permitted, multiple leading signs are not permitted. The “E” scientific notation character is permitted, but not evaluated. Leading zeros are not permitted; trailing zeros are preserved. A decimal separator must have a digit character on both sides of it. Therefore, the JSON numerics 0, 0.0, 0.4, and 0.400 are valid. A negative sign on a zero value is preserved.

For IEEE floating-point numbers, additional rules apply. Refer to the `$DOUBLE` function for details.

JSON fractional numbers are stored in a different format than ObjectScript numbers. ObjectScript floating point fractional numbers are rounded when they reach their maximum precision, and trailing zeros are removed. JSON packed BCD fractional numbers allow for greater precision, and trailing zeros are retained. This is shown in the following example:

```
SET jarray=[1.23456789123456789876000, (1.23456789123456789876000)]
WRITE jarray.%ToJSON()
```

- **Special values:** JSON supports the following special values: `true`, `false`, and `null`. These are literal values that must be specified as an unquoted literal in lowercase letters. These JSON special values cannot be specified using a variable, or specified in an ObjectScript expression.

- **ObjectScript:** To include an ObjectScript literal or expression within a JSON array element or a JSON object value, you must enclose the entire string in parentheses. You cannot specify ObjectScript in a JSON object key. ObjectScript and JSON use different escape sequence conventions. To escape a double quote character in ObjectScript, you double it. In the following example, a JSON string literal and an ObjectScript string literal are specified in a JSON array:

```
SET jarray=("This is a "good" JSON string", ("This is a "good" ObjectScript string"))
WRITE jarray.%ToJSON()
```
The following JSON array example specifies an ObjectScript local variable and performs ObjectScript numeric conversion to canonical form:

```
SET str="This is a string"
SET jarray=[(str),(--0007.000)]
WRITE jarray.%ToJSON()
```

The following example specifies an ObjectScript function in a JSON object value:

```
SET jobj={"firstname":"fred","namelen":($LENGTH("fred"))}
WRITE jobj.%ToJSON()
```

**JSON Object**

A *value* can be a JSON object delimited by curly braces. The *variable* is set to an oref, such as the following:

```
3@%Library.DynamicObject
```

You can resolve this oref to the JSON object value using the %ToJSON() function. This is shown in the following example:

```
SET jobj={"firstname":"Fred"}
WRITE "JSON object reference = ",jobj,!
WRITE "key value = ",jobj.%Get("firstname"),!
WRITE "JSON object value = ",jobj.%ToJSON()
```

A valid JSON object has the following format:

- Begins with an open curly brace, ends with a close curly brace. The empty object `{}` is a valid JSON object.
- Within the curly braces, a key:value pair or a comma-separated list of key:value pairs. Both the key and the value components are JSON literals, not ObjectScript literals.
- The key component must be a JSON quoted string literal. It cannot be an ObjectScript literal or expression enclosed in parentheses.
- The value component can be a JSON string or a JSON numeric literal. These JSON literals follow JSON validation criteria. A value component can be an ObjectScript literal or expression enclosed in parentheses. The value component can be specified as a defined variable specifying a string, a numeric, a JSON object, or a JSON array. The value component can contain nested JSON objects or JSON arrays. A value component can also be one of the following three JSON special values: true, false, null, specified as an unquoted literal in lowercase letters; these JSON special values cannot be specified using a variable.

The following are all valid JSON objects:

```
{"name":"Fred"},{"name":"Fred","city":"Bedrock"},
{"bool":true},{"1":true,"0":false,"Else":null},
{"name":{"fname":"Fred","lname":"Flintstone"},"city":"Bedrock"},
{"name":{"Fred","Wilma","Barney"},"city":"Bedrock"}.
```

A JSON object can specify a null property name and assign it a value, as shown in the following example:

```
SET jobj={}
SET jobj."=""This is the "null" property value"
WRITE "JSON null property object value = ",jobj.%ToJSON()
```

Note that the returned JSON string uses the JSON escape sequence (\") for a literal double quote character.

For further details, refer to “Creating and Modifying Dynamic Entities” in Using JSON in Caché.

**JSON Array**

A *value* can be a JSON array delimited by square brackets. The *variable* is set to an oref, such as the following:

```
1@%Library.DynamicArray
```

This oref can be resolved to the JSON array value using the %ToJSON() function. This is shown in the following example:

```
SET jary=["Fred","Wilma","Barney"]
WRITE "JSON array reference = ",jary,!
WRITE "JSON array value = ",jary.%ToJSON()
```
A valid JSON array has the following format:

- Begins with an open square bracket, ends with a close square bracket. The empty array \[\] is a valid JSON array.
- Within the square brackets, an element or a comma-separated list of elements. Each array element can be a JSON string or JSON numeric literal. These JSON literals follow JSON validation criteria. An array element can be an ObjectScript literal or expression enclosed in parentheses. An array element can be specified as a defined variable specifying a string, a numeric, a JSON object, or a JSON array. An array element can contain one or more JSON objects or JSON arrays. An array element can also be one of the following three JSON special values: true, false, null, specified as an unquoted literal in lowercase letters; these JSON special values cannot be specified using a variable.

The following are all valid JSON arrays: [1], [5,7,11,13,17], ["Fred","Wilma","Barney"], [true,false], ["Bedrock",["Fred","Wilma","Barney"]], [{"name":"Fred"},{"name":"Wilma"}], [{"name":"Fred","city":"Bedrock"},{"name":"Wilma","city":"Bedrock"}], [{"names":['Fred','Wilma','Barney']}].

For further details, refer to “Creating and Modifying Dynamic Entities” in Using JSON in Caché.

**Examples**

The following example specifies multiple arguments for the same SET command. Specifically, the command assigns values to three variables. Note that arguments are evaluated in left-to-right order.

```
SET var1=12,var2=var1*3,var3=var1+var2
WRITE "var1=",var1,"var2=",var2,"var3=",var3
```

The following example shows the (variable-list)=value form of the SET command. It shows how to assign the same value to multiple variables. Specifically, the command assigns the value 0 to three variables.

```
SET (sum,count,average)=0
WRITE "sum=",sum,"count=",count,"average=",average
```

The following example sets a subscripted global variable in a different namespace using extended global reference.

```
NEW $NAMESPACE
SET $NAMESPACE="%SYS"
SET ^["user"]nametest(1)="fred"
NEW $NAMESPACE
SET $NAMESPACE="USER"
WRITE ^nametest(1)
KILL ^nametest
```

**SET Command with Objects**

The following example contains three SET commands: the first sets a variable to an oref (object reference); the second sets a variable to the value of an object property; the third sets an object property to a value:

```
SET myobj=##class(%SQL.Statement).%New()
SET dmode=myobj.%SelectMode
WRITE "Default select mode=",dmode,"!
SET myobj.%SelectMode=2
WRITE "Newly set select mode=",myobj.%SelectMode
```

Note that dot syntax is used in object expressions; a dot is placed between the object reference and the object property name or object method name.

To set a variable with an object property or object method value for the current object, use the double-dot syntax:

```
SET x=..LastName
```

If the specified object property does not exist, Caché issues a <PROPERTY DOES NOT EXIST> error. If you use double-dot syntax and the current object has not been defined, Caché issues a <NO CURRENT OBJECT> error.

For further details, refer to Object-Specific ObjectScript Features in Using Caché Objects.
When using **SET** with objects, do not perform multiple assignments. For example, avoid statements such as:

```objectscript
// Avoid this syntax:
SET (a.Name,b.Name)=z
```

Instead, issue a separate **SET** command for each assignment, as shown in the following example:

```objectscript
SET a.Name=z
SET b.Name=z
```

The following command sets `x` to the value returned by the **GetNodeName()** method:

```objectscript
SET x=##class(%SYS.System).GetNodeName()
WRITE "the current system node is: ",{x}
```

A **SET** command for objects can take an expression with cascading dot syntax, as shown in the following examples:

```objectscript
SET x=patient.Doctor.Hospital.Name
```

In this example, the patient.Doctor object property references the Hospital object, which contains the Name property. Thus, this command sets `x` to the name of the hospital affiliated with the doctor of the specified patient. The same cascading dot syntax can be used with object methods.

A **SET** command for objects can be used with system-level methods, such as the following data type property method:

```objectscript
SET x=patient.NameIsValid(Name)
```

In this example, the **NameIsValid()** method returns its result for the current patient object. **NameIsValid()** is a boolean method generated for data type validation of the Name property. Thus, this command sets `x` to 1 if the specified name is a valid name, and sets `x` to 0 if the specified name is not a valid name.

### SET Using an Object Method

You can specify an object method on the left side of a **SET** expression. The following example specifies the **%Get()** method:

```objectscript
SET obj=##class(test).%New()  // Where test is class with a multidimensional property md
SET myarray={obj}
SET index=0,subscript=2
SET myarray.%Get(index).md(subscript)="value"
IF obj.md(2)="value" {WRITE "success"}
ELSE {WRITE "failure"}
```

### Notes

Each variable assignment can be a local variable, a process-private global, or a global, the `$PIECE` function, the `$EXTRACT` function, and certain special variables, including `$ECODE`, `$ETRAP`, `$DEVICE`, `$KEY`, `$TEST`, `$X`, and `$Y`.

If the target variable does not already exist, **SET** creates it and then assigns the value. If it does exist, **SET** replaces the existing value with the assigned value.

### SET and Subscripts

You can set individual subscripted values (array nodes) for a local variable, process-private global, or a global. You can set subscripts in any order. If the variable subscript level does not already exist, **SET** creates it and then assigns the value. Each subscript level is treated as an independent variable; only those subscript levels set are defined. For example:

```objectscript
KILL myarray
SET myarray{1,1,1}="Cambridge"
WRITE 1,myarray{1,1,1}
SET myarray{1}="address"
WRITE 1,myarray{1}
```

In this example, the variables myarray{1,1,1} and myarray{1} are defined and contain values. However, the variables myarray and myarray{1,1} are not defined, and return an `<UNDEFINED>` error when invoked.
By default, you cannot set a null subscript. For example, `SET ^x("") = 123` results in a `<SUBSCRIPT>` error. However you can set `%SYSTEM.Process.NullSubscripts()` method to allow null subscripts for global and process-private global variables. You cannot set a null subscript for a local variable.

The maximum length of a subscript is 511 characters. Exceeding this length results in a `<SUBSCRIPT>` error. The maximum number of subscript levels for a local variable is 255. The maximum number of subscript levels for a global variable depends on the subscript level names, and may exceed 255 levels. Attempting to set a local variable to more than 255 subscript levels (either directly or by indirection) results in a `<SYNTAX>` error. For further information on subscripted variables, refer to Global Structure in *Using Caché Globals*.

**Order of Evaluation**

Caché evaluates the arguments of the `SET` command in strict left-to-right order. For each argument, it performs the evaluation in the following sequence:

1. Evaluates occurrences of indirection or subscripts to the left of the equal sign in a left-to-right order to determine the variable name(s). For more information, refer to Indirection in *Using Caché ObjectScript*.
2. Evaluates the expression to the right of the equal sign.
3. Assigns the expression to the right of the equal sign to the variable name or references to the left of the equal sign.

**Transaction Processing**

A `SET` of a global variable is journaled as part of the current transaction; this global variable assignment is rolled back during transaction rollback. A `SET` of a local variable or a process-private global variable is not journaled, and thus this assignment is unaffected by a transaction rollback.

**Defined and Undefined Variables**

Most Caché commands and functions require that a variable be defined before it is referenced. By default, attempting to reference an undefined variable generates an `<UNDEFINED>` error. Attempting to reference an undefined object generates a `<PROPERTY DOES NOT EXIST>` or `<METHOD DOES NOT EXIST>` error. Refer to `$ZERROR` for further details on these error codes.

You can change Caché behavior when referencing an undefined variable by setting the `%SYSTEM.Process.Undef()` method.

The `READ` command and the `$INCREMENT` function can reference an undefined variable and assign a value to it. The `$DATA` function can take an undefined or defined variable and return its status. The `$GET` function returns the value of a defined variable; optionally, it can also assign a value to an undefined variable.

**SET with $PIECE and $EXTRACT**

You can use the `$PIECE` and `$EXTRACT` functions with `SET` on either side of the equals sign. For detailed descriptions, refer to `$PIECE` and `$EXTRACT`.

When used on the right side of the equals sign, `$PIECE` and `$EXTRACT` extract a substring from a variable and assign its value to the specified variable(s) on the left side of the equals sign. `$PIECE` extracts a substring using a specified delimiter, and `$EXTRACT` extracts a substring using a character count.

For example, assume that variable `x` contains the string "HELLO WORLD". The following commands extract the substring "HELLO" and assign it to variables `y` and `z`, respectively:

```plaintext
SET x="HELLO WORLD"
SET y=$PIECE(x," ",1)
SET z=$EXTRACT(x,1,5)
WRITE "x=",x," y=",y," z=",z
```
When used on the left side of the equals sign, $PIECE and $EXTRACT insert the value from the expression on the right side of the equals sign into the specified portion of the target variable. Any existing value in the specified portion of the target variable is replaced by the inserted value.

For example, assume that variable x contains the string "HELLO WORLD" and that variable y contains the string "HI THERE". In the command:

```objectscript
SET x="HELLO WORLD"
SET y="HI THERE"
SET $PIECE(x," ",2)=$EXTRACT(y,4,9)
WRITE "x="x,
```

The $EXTRACT function extracts the string " THERE" from variable y and the $PIECE function inserts it into variable x at the second field position, replacing the existing string "WORLD". Variable x now contains the string "HELLO THERE".

If the target variable does not exist, the system creates it and pads it with delimiters (in the case of $PIECE) or with spaces (in the case of $EXTRACT) as needed.

In the following example, SET $EXTRACT is used to insert the value of z into strings x and y, overwriting the existing values:

```objectscript
SET x="HELLO WORLD"
SET y="OVER EASY"
SET z="THERE"
SET $EXTRACT(x,7,11)=z
SET $EXTRACT(y,*-3,*)=z
WRITE "edited x=",x,!
WRITE "edited y=",y
```

Variable x now contains the string "HELLO THERE" and y contains the string "OVER THERE". Note that because one of the SET $EXTRACT operations in this example uses a negative offset (*-3) these operations must be done as separate sets. You cannot set multiple variables with a single SET using enclosing parentheses if any of the variables uses negative offset.

In the following example, assume that the global array ^client is structured so that the root node contains the client’s name, with subordinate nodes containing the street address and city. For example, ^client(2,1,1) would contain the city address for the second client stored in the array.

Assume further that the city node (x,1,1) contains field values identifying the city, state abbreviation, and ZIP code (postal code), with the comma as the field separator. For example, ^client(2,1,1) would contain the city address for the second client stored in the array.

The three SET commands in the following code each use the $PIECE function to assign a specific portion of the array node value to the appropriate local variable. Note that in each case $PIECE references the comma (",") as the string separator.

```objectscript
ADDRESSPIECE
SET "client(2,1,1)="Cambridge,MA,02142"
SET city=$PIECE("client(2,1,1),","",1)
SET state=$PIECE("client(2,1,1),","",2)
SET zip=$PIECE("client(2,1,1),","",3)
WRITE "City is ",city,!
   "State or Province is ",state,!
   "Postal code is ",zip
QUIT
```

The $EXTRACT function could be used to perform the same operation, but only if the fields were fixed length and the lengths were known. For example, if the city field was known to contain only up to 9 characters and the state and ZIP fields were known to contain only 2 and 5 characters, respectively, the SET commands could be coded with the $EXTRACT function as follows:
Notice the gaps between 9 and 11 and 12 and 14 to accommodate the comma field separators.

The following example replaces the first substring in A (originally set to 1) with the string "abc".

StringPiece
SET A="1^2^3^4^5^6^7^8^9"
SET $PIECE(A,"^")="abc"
WRITE !,"A=",A
QUIT

A="abc^2^3^4^5^6^7^8^9"

The following example uses $EXTRACT to replace the first character in A (again, a 1) with the string "abc".

StringExtract
SET A="123456789"
SET $EXTRACT(A)="abc"
WRITE !,"A=",A
QUIT

A="abc23456789"

The following example replaces the third through sixth pieces of A with the string "abc" and replaces the first character in the variable B with the string "abc".

StringInsert
SET A="1^2^3^4^5^6^7^8^9"
SET B="123"
SET ($PIECE(A,"^",3,6),$EXTRACT(B))="abc"
WRITE !,"A=",A!,"B=",B
QUIT

A="1^2^abc^7^8^9"

B="abc23"

The following example sets $X, $Y, $KEY, and the fourth piece of a previously unde

SetVars
SET ($X,$Y,$KEY,$PIECE(A,"^",4))=20,X=$X,Y=$Y,K=$KEY
WRITE !,"A=",A!,"K=",K!,"X=",X!,"Y=",Y
QUIT

A="^^^20" K="20" X=20 Y=20

The following example sets $ECODE and $ETRAP to the null string. Then, the example sets the local variables EC and ET to the values of $ECODE and $ETRAP.

SetEvars
SET ($ECODE,$ETRAP)="",EC=$ECODE,ET=$ETRAP
WRITE "EC=",EC!,"ET=",ET
QUIT

EC=""
ET=""
**SET with $LIST and $LISTBUILD**

The $LIST functions create and manipulate lists. They encode the length (and type) of each element within the list, rather than using an element delimiter. They then use the encoded length specifications to extract specified list elements during list manipulation. Because the $LIST functions do not use delimiter characters, the lists created using these functions should not be input to $PIECE or other character-delimiter functions.

When used on the *right side of the equal sign*, these functions return the following:

### $LIST

$LIST returns the specified element of the specified list.

### $LISTBUILD

$LISTBUILD returns a list containing one element for each argument given.

When used on the *left side of the equal sign*, in a SET argument, these functions perform the following tasks:

### $LIST

Replaces the specified element(s) with the value given on the right side of the equal sign.

```
SET A=$LISTBUILD("red","blue","green","white")
WRITE "Created list A=",LISTTOSTRING(A),!
SET $LIST(A,2)="yellow"
WRITE "Edited list A=",LISTTOSTRING(A)
```

```
SET A=$LISTBUILD("red","blue","green","white")
WRITE "Created list A=",LISTTOSTRING(A),!
SET $LIST(A,*-1,*)=$LISTBUILD("yellow")
WRITE "Edited list A=",LISTTOSTRING(A)
```

You cannot use parentheses with SET $LIST to assign the same value to multiple variables.

### $LISTBUILD

Extracts several elements of a list in a single operation. The arguments of $LISTBUILD are variables, each of which receives an element of the list corresponding to their position in the $LISTBUILD parameter list. Variable names may be omitted for positions that are not of interest.

In the following example, $LISTBUILD (on the right side of the equal sign) is first used to return a list. Then $LISTBUILD (on the left side of the equal sign) is used to extract two items from that list and set the appropriate variables.

```
SetListBuild
SET J=$LISTBUILD("red","blue","green","white")
SET $LISTBUILD(A,,B)=J
WRITE ",A=",A,"B=",B
```

In this example, A="red" and B="green".

### See Also

- $LIST function
- $LISTBUILD function
- $EXTRACT function
- $PIECE function
- $X special variable
- $Y special variable
TCOMMIT

Marks the successful completion of a transaction.

<table>
<thead>
<tr>
<th>Argument</th>
<th>pc</th>
<th>Optional — A postconditional expression.</th>
</tr>
</thead>
</table>

**Description**

TCOMMIT marks the successful end of a transaction initiated by the corresponding TSTART.

TCOMMIT decrements the value of the $TLEVEL special variable. Caché terminates the transaction only if $TLEVEL goes to 0. Usually this is when TCOMMIT has been called as many times as TSTART. Changes made during nested transactions are not committed until $TLEVEL = 0.

Calling TCOMMIT when $TLEVEL is already 0 results in a <COMMAND> error. This can occur if you issue a TCOMMIT when no transaction is in progress, when the number of TCOMMIT commands is larger than the number of TSTART commands, or following a TROLLBACK command. The corresponding $ZERROR value consists of <COMMAND>, the location of the error (for example +3^mytest), and the data literal *NoTransaction.

**Argument**

pc

An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

**Examples**

You use TCOMMIT with the TROLLBACK and TSTART commands. See TROLLBACK and TSTART for examples of how to use these transaction processing commands together.

**Notes**

**Nested TSTART / TCOMMIT**

Caché supports the nesting of the TSTART/TCOMMIT commands, so that modules can issue their TSTART/TCOMMIT pairs correctly, independent of any other TSTART/TCOMMIT issued in the modules that called them or in the modules they call. The current nesting level of the transaction is tracked by the special variable $TLEVEL. The transaction is committed when the outermost matching TCOMMIT is issued; that is, when $TLEVEL goes back to 0.

You can roll back individual nested transactions by calling TROLLBACK 1 or roll back all current transactions by calling TROLLBACK. TROLLBACK rolls back the whole transaction that is in effect — no matter how many levels of TSTART were issued — and sets $TLEVEL to 0.

**Network Transactions**

To synchronize the completion of transactions over a network, use the ZSYNC command.
**Synchronous Commit**

A **TCOMMIT** command requests a flush of the journal data involved in that transaction to disk. Whether to wait for this disk write operation to complete is a configurable option.

Go to the Management Portal, select [System] > [Configuration] > [Compatibility Settings]. View and edit the current setting of **SynchCommit**. When set to “true”, TCOMMIT does not complete until the journal data write operation completes. When set to “false”, TCOMMIT does not wait for the write operation to complete. The default is “false”. A restart is required for a change to the **SynchCommit** setting to take effect.

**SQL and Transactions**

ObjectScript and SQL transaction commands are fully compatible and interchangeable, with the following exception:

ObjectScript **TSTART** and SQL **START TRANSACTION** both start a transaction if no transaction is current. However, **START TRANSACTION** does not support nested transactions. Therefore, if you need (or may need) nested transactions, it is preferable to start the transaction with **TSTART**. If you need compatibility with the SQL standard, use **START TRANSACTION**.

ObjectScript transaction processing provides limited support for nested transactions. SQL transaction processing supplies support for savepoints within transactions.

**See Also**

- **TROLLBACK** command
- **TSTART** command
- **STLEVEL** special variable
- Using ObjectScript for Transaction Processing in *Using Caché ObjectScript*
THROW

Explicitly throws an exception to the next exception handler.

| THROWS oref |

**Argument**

| oref | Optional — An object reference (OREF) that is thrown to an exception handler. Optional, but highly recommended. |

**Description**

The `THROW` command explicitly throws an exception. An exception can be a system error, a `%Status` exception, or a user-defined exception. It throws this exception as an object reference (OREF) that inherits from the `%Exception.AbstractException` object. The `THROW` command throws this exception to the next exception handler.

There are two ways to use `THROW` *oref*:

- **TRY/CATCH**: Use `THROW oref` to explicitly signal an exception from within a `TRY` block of code, transferring execution from the `TRY` block to its corresponding `CATCH` block exception handler.
- **Other Exception Handlers**: Use `THROW oref` to explicitly signal an exception when not in a `TRY` block. This triggers the current exception handler (for example, `$ZTRAP`), where the *oref* can be retrieved from the `$THROWOBJ` special variable.

**Note**: Use of `THROW` without an argument is deprecated and not recommended for new code.

**System Errors**

Caché issues a system error when a runtime error occurs, such as referencing an undefined variable. A system error generates a `%Exception.SystemException` object reference, sets the *oref* properties `Code`, `Name`, `Location`, and `Data`, and also sets the `$ZERROR` and `$ECODE` special variables, and transfers control to the next error handler. This error handler can be a `CATCH` exception handler, or a `$ZTRAP` or `$ETRAP` error handler. A system error is an implicit error, which does not use a `THROW`.

You can use `THROW` within an error handler to throw a system error object to a further error handler. This is known as re-signalling a system error.

A `THROW` passes control up the execution stack to the next error handler. If the exception is an `%Exception.SystemException` object, the next error handler can be any type (`CATCH`, `$ZTRAP`, or `$ETRAP`). Otherwise, there must be a `CATCH` to handle the exception or Caché generates a `<THROW>` error.

**Argument**

*oref*

A reference to an exception object, which is an instance of any class that inherits from `%Exception.AbstractException`. A exception object for a system error is an instance of the class `%Exception.SystemException`. A user-specified exception object can be a `%Status` exception object (%Exception.StatusException), a general exception object (%Exception.General), or SQL exception object (%Exception.SQL). The creation and population of a user exception object is the responsibility of the programmer.

For information on OREFs, see “OREF Basics” in *Using Caché Objects*. 

Caché ObjectScript Reference 143
THROW from a TRY Block

THROW oref can be issued from a TRY block to its corresponding CATCH block. This explicitly signals a user-defined exception. This transfers execution from a TRY block to its corresponding CATCH block. The thrown oref is set as the CATCH block’s exceptionvar argument.

To issue an argumented THROW from a CATCH exception handler, you can either throw to a non-CATCH exception handler, or you can nest a TRY block (and associated nested CATCH block) within the CATCH exception handler, and issue the THROW from this nested TRY block.

%Status Exceptions and User-Defined Exceptions

To trap a %Status exception or a user-defined exception, specify an object based on the %Exception.AbstractException object as the oref argument. Define an exception class, then create an instance of the class using %New() and supply the exception information. These types of exceptions must be handled by a CATCH exception handler. If no CATCH exists, the system generates a <THROW> error.

A user-defined exception does not change the value of $ZERROR or $ECODE. In order to use either of these special variables, your program must explicitly set them using the SET command.

General Exception

Caché supplies a general exception that you can supply as the THROW argument. This is the %Exception.General subclass of the %Exception.AbstractException abstract class. Its use is shown in the following example:

```objectscript
TRY {
    WRITE "In the TRY block",!!
    SET mygenex = ##class(%Exception.General).%New("My exception","999",,
        "My own special exception")
    THROW mygenex
    WRITE "This shouldn't display",!
}
CATCH stuff {
    WRITE "In the CATCH block",!
    WRITE stuff.Name,!
    WRITE stuff.Code,!
    WRITE stuff.Data,!
    WRITE "End of the CATCH block",!
    RETURN
}
```

THROW when not in a TRY Block

If you issue a THROW outside of a TRY block, Cache generates a <THROW> error, such as the following:

```
<THROW>+3^myprog *%Exception.General MyErr 999 My user-defined error.This use of THROW is useful for re-signalling an error.
```

The object reference (oref) specified in the THROW is stored in the $THROWOBJ special variable. For example, 9@%Exception.General. The $THROWOBJ value is cleared by the next successful THROW operation, or by SET $THROWOBJ="".

In the following example, a THROW throws an exception to a $ZTRAP exception handler:

```objectscript
MainRou
    WRITE "In the Main Routine",!!
    SET $ZTRAP="ErrRou"
    SET mygenex = ##class(%Exception.General).%New("My exception","999",,
        "My own special exception")
    THROW mygenex
    WRITE "This shouldn't display",!
    RETURN
```
**THROW without an Argument**

Argumentless THROW re-signals the current system error, transferring control to the next exception handler. The current system error is the error referenced by the $ZERROR special variable. Thus, an argumentless THROW is equivalent to the command ZTRAP $ZERROR.

Use of argumentless THROW is not recommended, because which system error is the current system error may change. For instance, this would occur if the error handler changes the $ZERROR value, or if the error handler itself generates a system error. It is therefore preferable to explicitly specify the system error to be thrown to the next exception handler by using THROW oref.

**Examples**

Note that $ZCVT(myerr.Name,"O","HTML") is used in these examples because Caché error names are enclosed in angle brackets and these examples are run from a web browser. In most other contexts, myerr.Name will return the desired value.

The following example uses an instance of the %Exception.General class to throw a user-defined exception. It generates birth dates in the TRY block; if it generates a birth date that is in the future, it uses THROW to issue a general exception, passing the OREF of the user-defined exception to a general-purpose CATCH block. (You may have to run this example more than once to generate a date that throws an exception):

```
TRY {
    WRITE "In the TRY block",!
    SET badDOB=#class(%Exception.General).%New("<BAD DOB>"","999","Birth date is in the future")
    FOR x=1:1:20 { SET rndDOB = $RANDOM(7)_$RANDOM(10000)
        IF rndDOB > $HOROLOG { WRITE !,"Birthdate ",$ZDATE(rndDOB,1,,4)," is invalid"
            THROW badDOB }
        ELSE { WRITE "Birthdate ",$ZDATE(rndDOB,1,,4)," is valid",! }
    }
} CATCH err {
    WRITE !,"In the CATCH block"
    WRITE !,"Error code=",err.Code
    WRITE !,"Error name=",$ZCVT(err.Name,"O","HTML")
    WRITE !,"Error data=",err.Data
    RETURN
}
```

The following example can issue either of two THROW commands with a user-defined argument. $SRANDM picks which THROW to issue (random values 0 or 1) or to not issue a THROW (random value 2). Note that code execution continues after the TRY / CATCH block pair, unless block execution ends with a RETURN command:

```
TRY {
    SET errdatazero="this is the zero error"
    SET errdataone="this is the one error"
    /* Error Randomizer */
    SET test=$RANDOM(3)
    WRITE "Error test is ",test,!
    IF test=0 {
        WRITE !,"Throwing exception 998",!
        THROW #class(Sample.MyException).%New("TestZeroError",998,,errdatazero)
        THROW myvar
    }
    ELSEIF test=1 {
        WRITE !,"Throwing exception 999",!
        THROW #class(Sample.MyException).%New("TestOneError",999,,errdataone)
    }
    ELSE { WRITE !,"No THROW error this time" }
} CATCH exp {
    WRITE !,"This is the exception handler"
```
WRITE !,"Error code="",exp.Code
WRITE !,"Error name="",exp.Name
WRITE !,"Error data="",exp.Data
RETURN
}
WRITE !,"Execution after TRY block continues here"

The following example shows the use of THROW with a system error. THROW is commonly used in a CATCH exception handler to forward the system error to another handler. This may occur when the system error received is an unexpected type of system error. Note that this requires nesting a TRY block (and corresponding CATCH block) within the CATCH block. It is used to THROW the system error to the nested CATCH block. This is shown in the following example, which calls Calculate to perform a division operation and return the answer. There are three possible outcomes: If \( y \) = any non-zero number, the division operation succeeds and no CATCH block code is executed. If \( y=0 \) (or any nonnumeric string), the division operation attempts to divide by zero, throwing a system error to its CATCH block; this is caught by the calcerr exception handler, which “corrects” this error and returns a value of 0. If, however, \( y \) is not defined (\$NEW \( y \)), calcerr catches an unexpected system error, and throws this error to the myerr exception handler. To demonstrate these three possible outcomes, this sample program uses \$RANDOM to set the divisor \( (y) \):

```
Randomizer
SET test=\$RANDOM(3)
IF test=0 { SET y=0 }
ELSEIF test=1 { SET y=7 }
ELSEIF test=2 { \$NEW y }
/* Note: if test=2, y is undefined */
Main
SET x=4
TRY {
  SET result=\$Calculate(x,y)
  WRITE !,"Calculated value="",result
}
CATCH myerr {
  WRITE !,"this is the exception handler"
  WRITE !,"Error code="",myerr.Code
  WRITE !,"Error name="\$ZCVT(myerr.Name,"O","HTML")
  WRITE !,"Error data="",myerr.Data
}
QUIT

Calculate(arg1,arg2) PUBLIC {
  TRY {
    SET answer=arg1/arg2
  }
  CATCH calcerr {
    WRITE "In the CATCH Block",!
    TRY {
      IF calcerr.Name="<DIVIDE>" {
        WRITE !,"handling zero divide error"
        SET answer=0 }
      ELSE {
        THROW calcerr
      }
    RETURN
  }
  CATCH {
    WRITE "Unexpected error",!
    WRITE "Error name="\$ZCVT(myerr.Name,"O","HTML")",!
  }
}
QUIT answer
```

See Also

- CATCH command
- TRY command
- ZTRAP command
- \$ETRAP special variable
- \$THROWOBJ special variable
- \$ZERROR special variable
- \$ZTRAP special variable
• Error Processing in *Using Caché ObjectScript*
**TROLLBACK**

Rolls back an unsuccessful transaction.

```
TROLLBACK:pc
TRO:pc
TROLLBACK:pc 1
TRO:pc 1
```

**Arguments**

<table>
<thead>
<tr>
<th>pc</th>
<th>Optional — A postconditional expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Optional — The integer 1. Rolls back one level of nesting. Must be specified as a literal.</td>
</tr>
</tbody>
</table>

**Description**

**TROLLBACK** terminates the current transaction and restores all journaled database values to the values they held at the start of the transaction. **TROLLBACK** has two forms:

- **TROLLBACK** rolls back all transactions in progress (no matter how many levels of **TSTART** were issued) and resets `$TLEVEL` to 0.
- **TROLLBACK 1** rolls back the current level of nested transactions (the one initiated by the most recent **TSTART**) and decrements `$TLEVEL` by 1. The 1 argument must be the literal number 1; it cannot be a variable or an expression that resolve to 1. Numbers other than 1 are not supported.

You can determine the level of nested transactions from the `$TLEVEL` special variable. Calling **TROLLBACK** when `$TLEVEL` is 0 has no effect.

You can use the `GetImageJournalInfo()` method of the %SYS.Journal.System class to search the journal file for **TSTART** commands, and thus identify open transactions. A **TSTART** increments `$TLEVEL` and writes a journal file record: either a “BT” (Begin Transaction) record if `$TLEVEL` was zero, or a “BTL” (Begin Transaction with Level) record if `$TLEVEL` was greater than 0. Use the `Sync()` method of the %SYS.Journal.System class to flush the journal buffer following a successful rollback operation.

**TROLLBACK** disables Ctrl-C interrupts for the duration of the rollback operation.

**What Is and Isn’t Rolled Back**

**TROLLBACK** rolls back all journaled operations. These include the following:

- **TROLLBACK** rolls back most changes to global variables, including **SET** and **KILL** operations. Global variables underlie many Caché operations. Local variables are not reverted by a rollback operation.
  
  **TROLLBACK** rolls back changes to bit string values in global variables during a transaction. However, a rollback operation does not return the global variable bit string to its previous internal string representation.

- **TROLLBACK** rolls back insert, update, and delete changes to SQL data.

However, not all changes made by an application are journaled:

- **TROLLBACK** does not roll back changes to local variables or process-private globals.
- **TROLLBACK** does not roll back changes to special variables, such as `$TEST`.
- **TROLLBACK** does not roll back changes to the current namespace.
- **TROLLBACK** does not roll back **LOCK** command lock or unlock operations.
- **TROLLBACK** does not roll back $INCREMENT (or $ZINCREMENT) changes to global variables.
- **TROLLBACK** does not roll back $SEQUENCE changes to global variables.

Caché treats a **SET** or **KILL** of a global variable as a journaled transaction event; rolling back the transaction reverses these operations. Caché does not treat a **SET** or **KILL** of a local variable or a process-private global variable as a journaled transaction event; rolling back the transaction has no effect on these operations. By default, a **SET** or **KILL** of a global variable is immediately visible by other processes while the transaction is in progress. To prevent the **SET** or **KILL** of a global variable invoked within a transaction from being seen by other users before the transaction has been committed, you must coordinate access to the global variable via the **LOCK** command.

**Transaction Rollback Logging**

If an error occurs during a roll back operation, Caché issues a **<ROLLFAIL>** error message, and logs an error message in the cconsole.log operator console log file. You can use the Management Portal **System Operation** option to view cconsole.log:

[System] > [System Logs] > [View Console Log].

By default, the operator console log file is cconsole.log in the Caché system management directory (Mgr). This default is configurable. Go to the Management Portal **System Administration** option, select **Configuration**, then **Additional Settings**, then **Advanced Memory** ([System] > [Configuration] > [Advanced Memory Settings]). View and edit the current setting of **ConsoleFile**. By default this setting is blank, routing console messages to cconsole.log in the MGR directory. If you change this setting, you must restart Caché for this change to take effect.

**<ROLLFAIL> Errors**

If **TROLLBACK** cannot successfully roll back the transaction, a **<ROLLFAIL>** error occurs. The process behavior depends on the setting of the system-wide journal configuration setting flag **Freeze on error** ([System] > [Configuration] > [Journal Settings]):

- If **Freeze on error** is not set (the default), the process gets a **<ROLLFAIL>** error. The transaction is closed and any locks retained for the transaction are released. This option trades data integrity for system availability.
- If **Freeze on error** is set, the process halts and the clean job daemon (CLNDMN) retries rolling back the open transaction. During the CLNDMN retry period, locks retained for the transaction are intact and, as a result, the system might hang. This option trades system availability for data integrity.

For further details, refer to **Journal IO Errors** in the *Caché Data Integrity Guide*.

When a **<ROLLFAIL>** occurs, the Caché %msg records both the **<ROLLFAIL>** error itself, and the previous error that caused the roll back. For example, attempting to update a date with an out-of-range value and then failing roll back might return the following %msg:

SQLCODE = -105  %msg = Unexpected error occurred:  **<ROLLFAIL>**%0Ac+1^dpv during TROLLBACK. Previous error: SQLCODE=-105, %msg='Field 'Sample.Person.DOB' (value '5888326') failed validation'.

A **<ROLLFAIL>** occurs upon transaction rollback if within the transaction a global accessed a remote database, and then the program explicitly dismounted that remote database.

A **<ROLLFAIL>** occurs upon transaction rollback if the process disabled journaling before making database changes and an error occurred that invoked transaction rollback. A **<ROLLFAIL>** **does not** occur upon transaction rollback if the process disabled journaling after all database changes had been made but before issuing the **TROLLBACK** command. Instead, Caché temporarily enables journaling for the duration of the rollback operation. Upon completion of the rollback operation Caché again disables journaling.

**SQL and Transactions**

ObjectScript and SQL transaction commands are fully compatible and interchangeable, with the following exception:

ObjectScript **TSTART** and SQL **START TRANSACTION** both start a transaction if no transaction is current. However, **START TRANSACTION** does not support nested transactions. Therefore, if you need (or may need) nested transactions,
it is preferable to start the transaction with TSTART. If you need compatibility with the SQL standard, use START TRANSACTION.

ObjectScript transaction processing provides limited support for nested transactions. SQL transaction processing supplies support for savepoints within transactions.

**Purging Cached Queries**

If during a transaction you call the Purge() method of %SYSTEM.Selenium class to purge cached queries, the cached queries are permanently deleted. A subsequent TROLLBACK will not restore purged cached queries.

**Globals and TROLLBACK 1**

TROLLBACK 1 rolls back and restores all globals changed within its nested transaction. However, if globals are changed that are mapped to a remote system that does not support nested transactions, these changes are treated as occurring at the outermost nested level. Such globals are only rolled back when a rollback resets $TLEVEL to 0, either by calling TROLLBACK or by calling TROLLBACK 1 when $TLEVEL=1.

**Locks and TROLLBACK 1**

TROLLBACK 1 does not restore locks established during its nested transaction to their prior state. All locks established during a transaction remain in the lock table until the transaction is concluded by a TROLLBACK to level 0 or a TCOMMIT. At that point Caché releases all locks created during the nested transaction, and restores all preexisting locks to their state before TSTART.

A TCOMMIT of a nested transaction does not release the corresponding locks, so a subsequent TROLLBACK can effect locks in a committed sub-transaction.

**Arguments**

**pc**

An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

**Examples**

The following example uses a single-level transaction to transfer a random amount of money from one account to another. If the transfer amount is more than the available balance, the program uses TROLLBACK to roll back the transaction:

```objectscript
SetupBankAccounts
  SET num=12345
  SET ^CHECKING(num,"balance")=500.99
  SET ^SAVINGS(num,"balance")=100.22
  IF $DATA(^NumberOfTransfers)=0 {SET ^NumberOfTransfers=0}
BankTransfer
  WRITE "Before transfer:","Checking=$",^CHECKING(num,"balance")," Savings=$",^SAVINGS(num,"balance"),!
  // Transfer funds from one account to another
  SET transfer=$RANDOM(1000)
  WRITE "transfer amount $",transfer,!
  DO CkToSav(num,transfer)
    IF ok=1 {WRITE "sucessful transfer",!,"Number of transfers to date=",^NumberOfTransfers,!}
    ELSE {WRITE "*** INSUFFICIENT FUNDS ***",!}
  WRITE "After transfer:","Checking=$",^CHECKING(num,"balance")," Savings=$",^SAVINGS(num,"balance"),!
RETURN
CkToSav(acct,amt)
  TSTART
  SET ^CHECKING(acct,"balance") = ^CHECKING(acct,"balance") - amt
  SET ^SAVINGS(acct,"balance") = ^SAVINGS(acct,"balance") + amt
  SET ^NumberOfTransfers=^NumberOfTransfers + 1
  IF ^CHECKING(acct,"balance") > 0 {TCOMMIT SET ok=1 QUIT:ok}
  ELSE {TROLLBACK SET ok=0 QUIT:ok}
```

150 Caché ObjectScript Reference
The following example shows the effects of TROLLBACK on nested transactions. Each TSTART increments $TLEVEL and sets a global. Issuing a TCOMMIT on the inner nested transaction decrements $TLEVEL, but the commitment of changes made in a nested transaction is deferred. In this case, the subsequent TROLLBACK on the outer transaction rolls back all changes made, including those in the inner “committed” nested transaction.

```
SET ^a(1)="[- - -]",^b(1)="[- - -]"
WRITE !,"level:","$TLEVEL," ",^a(1)," ",^b(1)
TSTART
LOCK +^a(1)
SET ^a(1)="hello"
WRITE !,"level:","$TLEVEL," ",^a(1)," ",^b(1)
TSTART
LOCK +^b(1)
SET ^b(1)="world"
WRITE !,"level:","$TLEVEL," ",^a(1)," ",^b(1)
TCOMMIT
WRITE !,"After TCOMMIT"
WRITE !,"level:","$TLEVEL," ",^a(1)," ",^b(1)
TROLLBACK
WRITE !,"After TROLLBACK"
WRITE !,"level:","$TLEVEL," ",^a(1)," ",^b(1)
QUIT
```

The following example shows how TROLLBACK rolls back global variables, but not local variables:

```
SET x="default",^y="default"
WRITE !,"level:","$TLEVEL"
WRITE !,"local:","x," global:"","^y
TSTART
SET x="first",^y="first"
WRITE !,"TSTART level:","$TLEVEL"
WRITE !,"local:","x," global:"","^y
TSTART
SET x=x_" second",^y="y_" second"
WRITE !,"TSTART level:","$TLEVEL"
WRITE !,"local:","x," global:"","^y
TSTART
SET x=x_" third",^y="y_" third"
WRITE !,"TSTART level:","$TLEVEL"
WRITE !,"local:","x," global:"","^y
TROLLBACK
WRITE !,"After Rollback:"
WRITE !,"TROLLBACK level:","$TLEVEL"
WRITE !,"local:","x," global:"","^y
```

The following example shows how $INCREMENT changes to a global are not rolled back.

```
SET ^x=-1,^y=0
WRITE !,"level:","$TLEVEL"
WRITE !,"Increment:","$INCREMENT(^x)," Add:"","^y
TSTART
SET ^y="y+1"
WRITE !,"level:","$TLEVEL"
WRITE !,"Increment:","$INCREMENT(^x)," Add:"","^y
TSTART
SET ^y="y+1",^z="z_" second"
WRITE !,"level:","$TLEVEL"
WRITE !,"Increment:","$INCREMENT(^x)," Add:"","^y
TSTART
SET ^y="y+1",^z="z_" third"
WRITE !,"level:","$TLEVEL"
WRITE !,"Increment:","$INCREMENT(^x)," Add:"","^y
TROLLBACK
WRITE !,"After Rollback"
WRITE !,"level:","$TLEVEL"
WRITE !,"Increment:","^x," Add:"","^y
```

See Also

- TCOMMIT command
- TSTART command
- $TLEVEL special variable
- Using ObjectScript for Transaction Processing in Using Caché ObjectScript
TRY

Identifies a block of code to monitor for errors during execution.

```objectscript
TRY {
. . .
}
```

Description

The **TRY** command takes no arguments. It is used to identify a block of ObjectScript code statements enclosed in curly braces. This block of code is protected code for structured exception handling. If an exception occurs within this block of code, Caché sets `$ZERROR` and `$ECODE`, then transfers execution to an exception handler, identified by the **CATCH** command.

An exception may occur as a result of a runtime error, such as attempting to divide by 0, or it may be explicitly propagated by issuing a **THROW** command. If no error occurs, execution continues with the next ObjectScript statement after the **CATCH** block of code.

A **TRY** block must be immediately followed by a **CATCH** block. You cannot specify either executable code statements or a label between the closing curly brace of the **TRY** code block and the **CATCH** command. However, you can specify comments between the **TRY** and block and its **CATCH** block. Only one **CATCH** block is permitted for each **TRY** block. However, it is possible to nest paired **TRY/CATCH** blocks, such as the following:

```objectscript
TRY {
   /* TRY code */
   TRY {
      /* nested TRY code */
   }
   CATCH {
      /* nested CATCH code */
   }
}
CATCH {
   /* CATCH code */
}
```

Commonly, an ObjectScript program consists of multiple **TRY** blocks, each **TRY** block immediately followed by its associated **CATCH** block.

**QUIT and RETURN**

You exit a **TRY** block using **QUIT** or **RETURN**. **QUIT** exits the current block structure and continues execution with the next command outside of that block structure. For example, if you are within a nested **TRY** block, issuing a **QUIT** exits that **TRY** block to the enclosing block structure. Issuing a **QUIT** command within a **TRY** block transfers execution to the first code line after the corresponding **CATCH** block. You cannot use an argumented **QUIT** to exit a **TRY** block; attempted to do so results in a compile error. To exit a routine completely from within a **TRY** block, issue a **RETURN** statement.

In rare circumstances, a **TRY** block **QUIT** or **RETURN** command may generate an exception. This could happen if the **TRY** created a new context and then deleted some aspect of the old context; attempting to revert to the old context would cause an exception. A **TRY** block **QUIT** or **RETURN** exception does not invoke the **CATCH** block exception handler.

**$ZTRAP and $ETRAP**

The **TRY** and **CATCH** commands perform error handling within an execution level. When an exception occurs within a **TRY** block, Caché normally executes the **CATCH** block of exception handler code that immediately follows the **TRY** block. This is the preferred error handling behavior.

You cannot set **$ZTRAP** or **$ETRAP** within a **TRY** block. However, you can set **$ZTRAP** or **$ETRAP** within an argumentless **DO** block within the **TRY** block. This is because the lines of the **DO** block are in their own execution level and can thus have their own error handler.
If a \$ZTRAP was set before entering the TRY block and an exception occurs within the TRY block, Caché takes the CATCH block rather than the \$ZTRAP.

If $ETRAP was set before entering the TRY block and an exception occurs within the TRY block, Caché may take SETRAP rather than CATCH unless you forestall this possibility. If both $ETRAP and CATCH are present when an exception occurs, Caché executes the error code (CATCH or $ETRAP) that applies to the current execution level. Because $ETRAP is intrinsically not associated with an execution level, Caché assumes that it is associated with the current execution level unless you specify otherwise. You must NEW $ETRAP before setting $ETRAP to establish a level marker for SETRAP, so that Caché will correctly take CATCH as the current level exception handler, rather than $SETRAP. Otherwise, a system error (including a system error thrown by the THROW command) may take the \$ETRAP exception handler.

**GOTO and DO**

You can use a GOTO or DO command to enter a TRY block at a label within the TRY block. If an exception occurs later in the TRY block, the CATCH block exception handler is taken, just as if you had entered the TRY block at the TRY keyword. However, for clarity of coding, entering a TRY block using GOTO or DO should be avoided.

You can, of course, issue a GOTO from within a TRY block or a CATCH block.

Using a GOTO or DO to enter a CATCH block is strongly discouraged.

**DO Within a TRY Block**

When using a TRY statement, a THROW causes a search of the frame stack trying to find the appropriate CATCH block. When the frame stack indicates execution within a TRY block then execution will resume at the corresponding CATCH block. However, Caché must remove any "local" calls within the current TRY block before executing the CATCH block.

If a TRY block contains a DO statement that results in a reentry to that TRY block, one of two things may happen:

A "local" DO call (DO call that remains within the current TRY block): If the previous frame stack entry is a DO call located in the same TRY block, that DO is assumed to be a "local" subroutine call within the current TRY block. In this case, the CATCH is not immediately entered, but instead the frame stack is popped (possibly removing some recently allocated NEW variables) and the search resumes at the DO call in the current TRY block. If the new previous frame stack entry is not a DO from inside the current TRY block then the corresponding CATCH block is entered. However, if the previous frame stack entry is another DO in the same TRY then the frame stack is popped again (along with recently allocated NEW variables). This operation continues until the previous frame stack entry is not a DO, at which point the CATCH block is entered.

A “recursive” DO call (DO call inside a TRY block that leaves the TRY block but later execution reenters that TRY block): When searching for a CATCH block, if the previous frame stack entry is a DO inside the current TRY block, but the target label of that previous stack frame is not within the current TRY block (including any nested TRY blocks) then the frame stack is not popped (and no recently allocated local variables are popped) and the CATCH block is immediately entered. Note that if that CATCH block does another THROW then it is possible that the current CATCH block will be reentered because the recursive DO frame is still on the frame stack.

**Examples**

The examples in this section show runtime errors (%Exception.SystemException errors). For examples of user-specified exceptions invoked by issuing a THROW, refer to the THROW and CATCH commands.

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 is skipped over. In the second example, the code fails with an \$UNDEFINED error, and execution is passed to the CATCH exception handler.

**TRY succeeds. CATCH block is skipped. Execution continues with the 2nd TRY block:**

```ObjectScript
TRY {
  WRITE "1st TRY block",!
  SET \$a="fred"
  WRITE "$a is a defined variable",!
  SET \$a=x
```
TRY {
WRITE !,"2nd TRY block",!
WRITE "This is where the code falls through",!
WRITE "$ZERROR: ",$ZERROR,!
WRITE "$ECODE: ",$ECODE
} CATCH exp2 {
WRITE !,"This is the 2nd CATCH exception handler",!
}

TRY fails. Execution continues with the CATCH block. CATCH block ends with RETURN, so 2nd TRY block is not executed:

TRY {
WRITE !,"1st TRY block",!
KILL x
WRITE "x is an undefined variable",!
SET a=x
} CATCH exp {
WRITE !,"This is the CATCH exception handler",!
IF 1=exp.%IsA("%Exception.SystemException") {
WRITE "System exception",!
WRITE "Name: ",$ZCVT(exp.Name,"O","HTML"),!
WRITE "Location: ",exp.Location,!
WRITE "Code: ",exp.Code,!
WRITE "Data: ",exp.Data,!!
} ELSE { WRITE "not a system exception",!!
WRITE "$ZERROR: ",$ZERROR,!
WRITE "$ECODE: ",$ECODE
} RETURN
} TRY {
WRITE !,"2nd TRY block",!
WRITE "This is where the code falls through",!
WRITE "$ZERROR: ",$ZERROR,!
WRITE "$ECODE: ",$ECODE
} CATCH exp2 {
WRITE !,"This is the 2nd CATCH exception handler",!
}

TRY quits. In the following example, the CATCH block is not executed because execution of the TRY block is ended by either a QUIT or a RETURN, not an error. If RETURN, program execution stops. If QUIT, program execution continues with the 2nd TRY block:

TRY {
WRITE !,"1st TRY block",!
KILL x
WRITE "x is an undefined variable",!
SET decide=$RANDOM(2)
IF decide=0 { WRITE "issued a QUIT",!
QUIT }
IF decide=1 { WRITE "issued a RETURN",!
RETURN }
WRITE "This should never display",!
SET a=x
} CATCH exp {
WRITE !,"This is the CATCH exception handler",!
IF 1=exp.%IsA("%Exception.SystemException") {
WRITE "System exception",!
WRITE "Name: ",$ZCVT(exp.Name,"O","HTML"),!
WRITE "Location: ",exp.Location,!
WRITE "$ZERROR: ",$ZERROR,!
WRITE "$ECODE: ",$ECODE
}
WRITE "Code: ", exp.Code, !
WRITE "Data: ", exp.Data, !!
}
ELSE { WRITE "not a system exception",!!
WRITE "$ZERROR: ",$ZERROR, !
WRITE "$ECODE: ",$ECODE
RETURN
}
TRY {
WRITE !,"2nd TRY block",!
WRITE "This is where the code falls through",!
WRITE "$ZERROR: ",$ZERROR, !
WRITE "$ECODE: ",$ECODE
}
CATCH exp2 {
WRITE !,"This is the 2nd CATCH exception handler",!
}

See Also

- **CATCH** command
- **THROW** command
- **$ETRAP** special variable
- **Error Processing** in *Using Caché ObjectScript*
**TSTART**

Marks the beginning of a transaction.

<table>
<thead>
<tr>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
</tr>
</tbody>
</table>

**Description**

TSTART marks the beginning of a transaction. Following TSTART, database operations are journaled to enable a subsequent TCOMMIT or TROLLBACK command.

TSTART increments the value of the $TLEVEL special variable. A $TLEVEL value of 0 indicates that no transaction is in effect. The first TSTART begins a transaction and increments $TLEVEL to 1. Subsequent TSTART commands can create nested transactions, further incrementing $TLEVEL.

Not all operations that occur within a transaction can be rolled back. For example, setting global variables within a transaction can be rolled back; setting local variables within a transaction cannot be rolled back. Refer to Using ObjectScript for Transaction Processing in Using Caché ObjectScript for further details.

By default, a lock issued within a transaction will be held until the end of the transaction, even if the lock is released within the transaction. This default can be overridden when setting the lock. Refer to the LOCK command for more details.

**Nested Transactions**

If you issue a TSTART within a transaction it begins a nested transaction. Issuing a TSTART increments the $TLEVEL value, indicating the number of levels of transaction nesting. You end a nested transaction by issuing either a TCOMMIT to commit the nested transaction, or a TROLLBACK 1 to roll back the nested transaction. Ending a nested transaction decrements the $TLEVEL value by 1.

- Issuing a TROLLBACK 1 for a nested transaction rolls back changes made in that nested transaction and decrements $TLEVEL. You can issue a TROLLBACK to roll back the whole transaction, no matter how many levels of TSTART were issued.

- Issuing a TCOMMIT for a nested transaction decrements $TLEVEL, but the actual commitment of the nested transaction is deferred. Changes made during a nested transaction are only irreversibly committed when the outermost transaction is committed; that is, when a TCOMMIT decrements the $TLEVEL value to 0.

You can use the GetImageJournalInfo() method of the %SYS.Journal.System class to search the journal file for TSTART commands, and thus identify open transactions. A TSTART writes either a “BT” (Begin Transaction) journal file record if $TLEVEL was zero, or a “BTL” (Begin Transaction with Level) journal file record if $TLEVEL was greater than 0.

The maximum number of levels of nested transactions is 255. Attempting to exceed this nesting levels limit results in a TRANSACTION LEVEL> error.

**SQL and Transactions**

ObjectScript and SQL transaction commands are fully compatible and interchangeable, with the following exception:

ObjectScript TSTART and SQL START TRANSACTION both start a transaction if no transaction is current. However, START TRANSACTION does not support nested transactions. Therefore, if you need (or may need) nested transactions, it is preferable to start the transaction with TSTART. If you need compatibility with the SQL standard, use START TRANSACTION.
ObjectScript transaction processing provides limited support for nested transactions. SQL transaction processing supplies support for savepoints within transactions.

**Argument**

`pc`

An optional postconditional expression. Caché executes the TSTART command if the postconditional expression is true and does not execute the TSTART command if the postconditional expression is false. For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

**Examples**

The following example uses a single-level transaction to transfer a random amount of money from one account to another. If the transfer amount is more than the available balance, the program rolls back the transaction:

```
SetupBankAccounts
SET num=12345
SET ^CHECKING(num,"balance")=500.99
SET ^SAVINGS(num,"balance")=100.22
IF $DATA(^NumberOfTransfers)=0 {SET ^NumberOfTransfers=0}

BankTransfer
WRITE "Before transfer:","Checking=$",^CHECKING(num,"balance")," Savings=$",^SAVINGS(num,"balance"),!
// Transfer funds from one account to another
SET transfer=$RANDOM(1000)
WRITE "transfer amount $",transfer,!
DO CkToSav(num,transfer)
IF ok=1 {WRITE "successful transfer","Number of transfers to date=",^NumberOfTransfers,!
ELSE {WRITE "**** INSUFFICIENT FUNDS ****",!
WRITE "After transfer:","Checking=$",^CHECKING(num,"balance")," Savings=$",^SAVINGS(num,"balance"),!
RETURN

CkToSav(acct,amt)
TSTART
SET ^CHECKING(acct,"balance") = ^CHECKING(acct,"balance") - amt
SET ^SAVINGS(acct,"balance") = ^SAVINGS(acct,"balance") + amt
SET ^NumberOfTransfers=^NumberOfTransfers + 1
IF ^CHECKING(acct,"balance") > 0 {TCOMMIT SET ok=1 QUIT:ok}
ELSE {TROLLBACK SET ok=0 QUIT:ok}
```

The following examples use TSTART to create nested transactions. They show three scenarios for rollback of nested transactions:

Roll back the innermost transaction, commit the middle transaction, commit the outermost transaction:

```
KILL `a`, `b`, `c`
TSTART SET `a`=1 WRITE "tlevel=",$TLEVEL,!
TSTART SET `b`=2 WRITE "tlevel=",$TLEVEL,!
TSTART SET `c`=3 WRITE "tlevel=",$TLEVEL,!
ROLLBACK 1 WRITE "tlevel=",$TLEVEL,!
TCOMMIT WRITE "tlevel=",$TLEVEL,!
TCOMMIT WRITE "tlevel=",$TLEVEL,!
IF $DATA(`a`) {WRITE "`a`=",`a`!} ELSE {WRITE "`a` is undefined",!}
IF $DATA(`b`) {WRITE "`b`=",`b`!} ELSE {WRITE "`b` is undefined",!}
IF $DATA(`c`) {WRITE "`c`=",`c`!} ELSE {WRITE "`c` is undefined",!}
```

Commit the innermost transaction, roll back the middle transaction, commit the outermost transaction:

```
KILL `a`, `b`, `c`
TSTART SET `a`=1 WRITE "tlevel=",$TLEVEL,!
TSTART SET `b`=2 WRITE "tlevel=",$TLEVEL,!
TSTART SET `c`=3 WRITE "tlevel=",$TLEVEL,!
TCOMMIT WRITE "tlevel=",$TLEVEL,!
ROLLBACK 1 WRITE "tlevel=",$TLEVEL,!
TCOMMIT WRITE "tlevel=",$TLEVEL,!
TCOMMIT WRITE "tlevel=",$TLEVEL,!
IF $DATA(`a`) {WRITE "`a`=",`a`!} ELSE {WRITE "`a` is undefined",!}
IF $DATA(`b`) {WRITE "`b`=",`b`!} ELSE {WRITE "`b` is undefined",!}
IF $DATA(`c`) {WRITE "`c`=",`c`!} ELSE {WRITE "`c` is undefined",!}
```

Commit the innermost transaction, commit the middle transaction, roll back the outermost transaction:
KILL ^a,^b,^c
TSTART SET ^a=1 WRITE "tlevel=",$TLEVEL,!
TSTART SET ^b=2 WRITE "tlevel=",$TLEVEL,!
TSTART SET ^c=3 WRITE "tlevel=",$TLEVEL,!
TCOMMIT WRITE "tlevel=",$TLEVEL,!
TCOMMIT WRITE "tlevel=",$TLEVEL,!
TROLLBACK 1 WRITE "tlevel=",$TLEVEL,!
IF $DATA(^a) {WRITE "^a=",^a,!} ELSE {WRITE "^a is undefined",!}
IF $DATA(^b) {WRITE "^b=",^b,!} ELSE {WRITE "^b is undefined",!}
IF $DATA(^c) {WRITE "^c=",^c,!} ELSE {WRITE "^c is undefined",!}

Note that in this third case, TROLLBACK 1 and TROLLBACK would have the same result, because both would decrement $TLEVEL to 0.

See Also

- TCOMMIT command
- TROLLBACK command
- $TLEVEL special variable
- Using ObjectScript for Transaction Processing in Using Caché ObjectScript
USE

Establishes a device as the current device.

USE:pc useargument,...
U:pc useargument,...

where useargument can be:

device:(parameters):"mnespace"

Arguments

| pc | Optional — A postconditional expression. |
| device | The device to be selected as the current device, specified by a device ID or a device alias. A device ID can be an integer (a device number), a device name, or the pathname of a sequential file. If a string, it must be enclosed with quotation marks. |
| parameters | Optional — The list of parameters used to set device characteristics. The parameter list is enclosed in parentheses, and the parameters in the list are separated by colons. Parameters can either be positional (specified in a fixed order in the parameter list) or keyword (specified in any order). A mix of positional and keyword parameters is permitted. The individual parameters and their positions and keywords are highly device-dependent. |
| mnespace | Optional — The name of the mnemonic space that contains the control mnemonics to use with this device, specified as a quoted string. |

Description

USE device establishes the specified device as the current device. The process must have already established ownership of the device with the OPEN command.

The current device remains current until you issue another USE command to select another owned device as the current device or until the process terminates.

The USE command can establish as the current device such devices as terminal devices, magnetic tape devices, spool devices, TCP bindings, interprocess pipes, named pipes, and inter-job communications. The USE command can also be used to open a sequential file. The device argument specifies the file pathname as a quoted string.

The parameters available with the USE command are highly device-dependent. In many cases the available parameters are the same as those available with the OPEN command; however, some device parameters can only be set using the OPEN command, and other can only be set using the USE command.

The USE command can specify more than one useargument, separated by commas. However, you can only have one current device at a time. If you specify more than one useargument, the device specified in the last useargument becomes the current device. This form of USE may be used to set parameters for several devices, and then establish the last-named device as the current device.

Arguments

pc

An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.
**device**

The device to be selected as the current device. Specify the same device ID (or other device identifier) as you specified in the corresponding OPEN command. For more information on specifying devices, refer to the OPEN command.

**parameters**

The list of parameters used to set operating characteristics of the device to be used as the current device. The enclosing parentheses are required if there is more than one parameter. (It is good programming practice to always use parentheses when you specify a parameter.) Note the required colon before the left parenthesis. Within the parentheses, colons are used to separate multiple parameters.

The parameters for a device can be specified using either positional parameters or keyword parameters. You can also mix positional parameters and keyword parameters within the same parameter list.

In most cases, specifying contradictory, duplicate, or invalid parameter values does not result in an error. Wherever possible, Caché ignores inappropriate parameter values and takes appropriate defaults.

The available parameters are, in many cases, the same as those supported for the OPEN command. For sequential files, TCP devices, and interprocess communication pipes some parameters can only be set with the OPEN command; for sequential files some parameters can only be set with the USE command. USE parameters are specific to the type of device that is being selected and to the particular implementation. The USE command keyword parameters are listed by device type in I/O Devices and Commands in the Caché I/O Device Guide.

If you do not specify a list of USE parameters, Caché uses the device’s default OPEN parameters. The default parameters for a device are configurable. Go to the Management Portal, select [System] > [Configuration] > [Device Settings] > [Devices] to display the current list of devices. For the desired device, click “edit” to display its Open Parameters: option. Specify this value in the same way you specify the OPEN command parameters, including the enclosing parentheses. For example, (“AVL”:0:2048).

**Positional Parameters**

Positional parameters must be specified in a fixed sequence in the parameter list. You can omit a positional parameter (and receive the default value), but you must retain the colon to indicate the position of the omitted positional parameter. Trailing colons are not required; excess colons are ignored. The individual parameters and their positions are highly device-dependent.

There are two types of positional parameters: values and letter code strings.

A value can be an integer (for example, record size), a string (for example, host name), or a variable or expression that evaluates to a value.

A letter code string uses individual letters to specify device characteristics for the open operation. For most devices, this letter code string is one of the positional parameters. You can specify any number of letters in the string, and specify the letters in any order. Letter codes are not case-sensitive. A letter code string is enclosed in quotation marks; no spaces or other punctuation is allowed within a letter code string (exception: K and Y may be followed by a name delimited by backslashes: thus: K\name\). For example, when opening a sequential file, you might specify a letter code string of “ANDFW” (append to existing file, create new file, delete file, fix-length records, write access.) The position of the letter code string parameter, and the meanings of individual letters is highly device-dependent.

**Keyword Parameters**

Keyword parameters can be specified in any sequence in the parameter list. A parameter list can consist entirely of keyword parameters, or it can contain a mix of positional and keyword parameters. (Commonly, the positional parameters are spec-
USE device:(/KEYWORD1=value1:/KEYWORD2=value2:.../KEYWORDn=valuen):"mnespace"

The individual parameters and their positions are highly device-dependent. As a general rule, you can specify the same parameters and values using either a positional parameter or a keyword parameter. You can specify a letter code string as a keyword parameter by using the /PARAMS keyword.

**mnespace**

The name of the mnemonic space that contains the device control mnemonics used by this device. By default, Caché provides two mnemonic spaces: ^%XMAG for magnetic tape devices, and ^%X364 (ANSI X3.64 compatible) for all other devices and sequential files. Default mnemonic spaces are assigned by device type.

Go to the Management Portal, select [System] > [Configuration] > [Device Settings] > [IO Settings]. View and edit the File, MagTape, Other, or Terminal mnemonic space setting.

A mnemonic space is a routine that contains entry points for the device control mnemonics used by **READ** and **WRITE** commands. The **READ** and **WRITE** commands invoke these device control mnemonics using the `/mnemonic(params)` syntax. These device control mnemonics perform operations such as moving the cursor to a specified screen location or rewinding a magnetic tape.

Use the **mnespace** argument to override the default mnemonic space assignment. Specify an ObjectScript routine that contains the control mnemonics entry points used with this device. The enclosing double quotes are required. Specify this option only if you plan to use device control mnemonics with the **READ** or **WRITE** command. If the mnemonic space does not exist, Caché issues a <NOROUTINE> error. For further details on mnemonic spaces, see I/O Devices and Commands in the Caché I/O Device Guide.

**Examples**

In this example, the **USE** command sets the sequential file "STUDENTS" as the current device and sets the file pointer so that subsequent reads begin at offset 256 from the start of the file.

```
USE "STUDENTS":256
```

**Notes**

**Device Ownership**

Device ownership is established with the **OPEN** command. The only exception is the principal device, which is assigned to the process and is usually the terminal at which you sign on. If the device specified in the **USE** command is not owned by the process, Caché issues a <NOTOPEN> error message.

**The Current Device**

The current device is the device used for I/O operations by the **READ** and **WRITE** commands. The **READ** command acquires input from the current device and the **WRITE** command sends output to the current device.

Caché maintains the ID of the current device in the $IO special variable. If the **USE** request is successful, Caché sets $IO to the ID of the specified device. The **GetType()** method of the %Library.Device class returns the device type of the current device.

**The Principal Device**

The special device number 0 (zero) refers to the principal device. Each process has one principal device. Caché maintains the ID of the principal device in the $PRINCIPAL special variable. The principal device is automatically opened when you start up Caché. Initially, the principal device ($PRINCIPAL) and the current device ($IO) are the same.
After you issue a **USE** command, your current device (`$IO`) is normally the one named in the last **USE** command you executed.

While many processes can have the same principal device, only one at a time can own it. After a process successfully issues an **OPEN** command for a device, no other process can issue **OPEN** for that device until the first process releases it, either by explicitly issuing a **CLOSE** command, by halting, or because that user ends the session.

Although you can issue **OPEN** and **USE** for a device other than your principal device from the Terminal prompt, each time Caché returns to the > prompt, it implicitly issues **USE 0**. To continue using a device other than 0, you must issue a **USE** command in each line you enter at the > prompt.

Your principal device automatically becomes your current device when you do any of the following:

- Log on.
- Issue a **USE 0** command.
- Cause an error when an error trap is not set.
- Close the current device.
- Return to the Terminal prompt.
- Exit Caché by issuing a **HALT** command.

**USE 0** implies an **OPEN** command to the principal device. If another process owns the device, this process hangs on the implicit **OPEN** as it does when it encounters any **OPEN**.

Although **USE 0** implies **OPEN 0** for the principal device, issuing a **USE** command for any other device that the process does not own (due to a previous **OPEN** command) produces a <NOTOPEN> error.

**Note:** While most Caché platforms allow you to close your principal input device, Caché for UNIX® does not. Therefore, when a job that is the child of another job tries to perform I/O on your login terminal, it hangs until you log off Caché. At that time, the output may or may not appear.

**Using the Null Device on UNIX®**

When you issue an **OPEN** and **USE** command to the null device (/dev/null on UNIX®), Caché treats the null device as a dummy device. Subsequent **READ** commands immediately return a null string (""). Subsequent **WRITE** commands immediately return success. No actual data is read or written. On systems based on UNIX®, the device /dev/null bypasses the UNIX® open, write, and read system calls entirely.

Processes started by other processes with the **JOB** command have a principal device of /dev/null by default.

If you open /dev/null other than within Caché for example, by redirecting Caché output to /dev/null from the UNIX® shell the UNIX® system calls operate as they do for any other device.

**See Also**

- **OPEN** command
- **CLOSE** command
- `$IO` special variable
- `$PRINCIPAL` special variable
- **I/O Devices and Commands** in *Caché I/O Device Guide*
- **Terminal I/O** in *Caché I/O Device Guide*
- **TCP Client/Server Communication** in *Caché I/O Device Guide*
• Magnetic Tape I/O in *Caché I/O Device Guide*
• Sequential File I/O in *Caché I/O Device Guide*
• The Spool Device in *Caché I/O Device Guide*
The **VIEW** command reads and writes database blocks and writes locations in memory. **VIEW** has two argument forms:

- **VIEW block** transfers data between the Caché database and memory.
- **VIEW offset:mode:length:newvalue** places `newvalue` in the memory location identified by `offset`, `mode`, and `length`.

You can examine data in memory with the `$VIEW` function.

**Note:** InterSystems recommends that you avoid use of the **VIEW** command. When used in any environment, it can corrupt memory structures.

### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pc</code></td>
<td>Optional — A postconditional expression.</td>
</tr>
<tr>
<td><code>block</code></td>
<td>A block location, specified as an integer.</td>
</tr>
<tr>
<td><code>offset</code></td>
<td>An offset, in bytes, from a base address within the memory region specified by <code>mode</code>.</td>
</tr>
<tr>
<td><code>mode</code></td>
<td>The memory region whose base address will be used to calculate the data to be modified.</td>
</tr>
<tr>
<td><code>length</code></td>
<td>The length of the data to be modified.</td>
</tr>
<tr>
<td><code>newvalue</code></td>
<td>The replacement value to be stored at the memory location.</td>
</tr>
</tbody>
</table>

### Description

The **VIEW** command reads and writes database blocks and writes locations in memory. **VIEW** has two argument forms:

- **VIEW block** transfers data between the Caché database and memory.
- **VIEW offset:mode:length:newvalue** places `newvalue` in the memory location identified by `offset`, `mode`, and `length`.

You can examine data in memory with the `$VIEW` function.

**Note:** InterSystems recommends that you avoid use of the **VIEW** command. When used in any environment, it can corrupt memory structures.

### Arguments

#### `pc`

An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

#### `block`

A block location, specified as an integer. If `block` is a positive integer, **VIEW** reads that number block into the view buffer. If `block` is a negative integer, **VIEW** writes the block currently in the view buffer to that block address. The `block` and the `offset:mode:length:newvalue` arguments are mutually exclusive.

If the block is already in a memory buffer, the current contents of the buffer will be copied.

Block location 0 is not a valid location. Attempting to specify **VIEW 0** results in a `<BLOCKNUMBER>` error.
offset

An offset, in bytes, from a base address within the memory region specified by mode.

mode

The memory region whose base address will be used to calculate the data to be modified. See Modifying Data in Memory for a description of the possible values.

length

The length of the data to be modified.

Specify the number of bytes as an integer from 1 to 4, or 8. You can also use the letters C or P to indicate the size of an address field (pointer) on the current platform.

If newvalue defines a string, specify the number of bytes as a negative integer, counting from 1. If the length of newvalue exceeds this number, Caché ignores the extraneous characters. If the length of newvalue is less than this number, Caché stores the supplied characters and leaves the rest of the memory location unchanged.

To store a byte value in reverse order (low-order byte at lowest address) append the letter O to the length number and enclose both in double quotes.

newvalue

The replacement value to be stored at the memory location.

Examples

The following example reads the sixth block from the Caché database into the view buffer:

```
VIEW 6
```

The following example writes the view buffer back to the sixth block of the Caché database, presumably after the data has been modified:

```
VIEW -6
```

The following example copies the string "WXYZ" into four bytes starting at offset ADDR in the view buffer. The expression $VIEW(ADDR, 0, -4) would then result in the value "WXYZ":

```
VIEW ADDR:0:-4:"WXYZ"
```

Notes

Use VIEW with Caution

Use the VIEW command with caution. It is usually used for debugging and repair of Caché databases and Caché system information. It is easy to corrupt memory or your Caché database by using VIEW incorrectly.

VIEW Usage Restricted

The VIEW command is a restricted system capability. It is a protected command because the invoked code is located in the CACHESYS database. For further details, refer to the “CACHESYS Special Capabilities” in the Assets and Resources chapter of the Caché Security Administration Guide.

The View Buffer

When used to read and write database buffers, the VIEW command works with the view buffer (device 63). The view buffer is a special memory area that you must open before you can perform any VIEW operations.
When you open the view buffer (with the OPEN command), you indicate the Caché database (CACHE.DAT) to be associated with the view buffer. Using the VIEW command, you can then read individual blocks from the Caché database into the view buffer.

After reading a block into the view buffer, you can use the $VIEW function to examine the data. Or, you can use the VIEW command to modify the data. If you modify the data, you can use the VIEW command again to write the modified block back to the Caché database.

**Reading and Writing Data in a Caché Database**

Before you can read and write data blocks in a Caché database with VIEW, you must first use the OPEN command to open the view buffer.

1. Open the view buffer. The view buffer is designated as device number 63. Hence the command is:

   OPEN 63:location

   where location is the namespace that contains the CACHE.DAT file to be associated with the view buffer. The location is implementation specific. The OPEN 63 command creates the view buffer by allocating a region of system memory whose size is equal to the block size used by the Caché database.

2. Use the VIEW block form to read in a block from the associated Caché database. Specify block as a positive integer. For example:

   VIEW 4

   This example reads the fourth block from the Caché database into the view buffer. Because the size of the view buffer equals the block size used in the Caché database, the view buffer can contain only one block at any given time. As you read in subsequent blocks, each new block overwrites the current block. To determine which blocks to read in from the Caché database, you should be familiar with the structure of the file.

3. Examine the data in the block with the $VIEW function or modify it with the VIEW command.

4. If you changed any of the data in the view buffer, write it back to the Caché database. To write data, use the VIEW block form but specify a negative integer for block. The block number usually matches the number of the current block in the view buffer, but it does not have to. The specified block number identifies which block in the file will be replaced (overwritten) by the block in the view buffer. For example, VIEW -5 replaces the fifth block in the Caché database with the current block in the view buffer.

5. Close the view buffer using CLOSE 63.

**Transferring a Block between Caché Databases**

When you open the view buffer, Caché does not automatically clear the existing block. This allows you to transfer a block of data from one Caché database to another using the following sequence:

1. Use OPEN 63 and specify the namespace that contains the first Caché database.
2. Use VIEW to read the desired block from the file into the view buffer.
3. If necessary, use VIEW to modify the data in the view buffer.
4. Use OPEN 63 again and specify the namespace that contains the second Caché database.
5. Use VIEW to write the block from the view buffer to the second Caché database.
6. Use CLOSE 63 to close the view buffer.

**Modifying Data in Memory**

In addition to reading and writing data from a Caché database, the VIEW command allows you to modify data in memory either in the view buffer or in other system memory areas.
To modify data, use the following form:

```
VIEW offset:mode:length:newvalue
```

All four arguments are required.

You modify data by storing a new value into a memory location, which is specified as a byte offset from the base address indicated by `mode`. You specify the amount of memory affected in the `length` argument.

The possible values for `mode` are shown in the following table:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Memory Management Region</th>
<th>Base Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>$JOB &gt; 0</td>
<td>Address space of process $n$, where $n$ is the value of <code>$JOB</code> for that process, a process ID (pid).</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>The view buffer</td>
<td>Beginning of view buffer</td>
</tr>
<tr>
<td>-1</td>
<td>The process's partition</td>
<td>Beginning of partition</td>
</tr>
<tr>
<td>-2</td>
<td>The system table</td>
<td>Beginning of system table</td>
</tr>
<tr>
<td>-3</td>
<td>The process's address space</td>
<td>0</td>
</tr>
<tr>
<td>-6</td>
<td>Reserved for InterSystems use</td>
<td></td>
</tr>
<tr>
<td>-7</td>
<td>Used only by the integrity checking utility</td>
<td>Special. See the Caché High Availability Guide.</td>
</tr>
</tbody>
</table>

**See Also**

- OPEN command
- CLOSE command
- $VIEW function
WHILE

Executes code while a condition is true.

**WHILE** expression,... {  
  code  
}  

**Arguments**

<table>
<thead>
<tr>
<th>expression</th>
<th>A test condition. You can specify one or more comma-separated test conditions, all of which must be TRUE for execution of the code block.</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>A block of ObjectScript commands enclosed in curly braces.</td>
</tr>
</tbody>
</table>

**Description**

WHILE tests expression and, if expression evaluates to TRUE, it then executes the block of code (one or more commands) between the opening and closing curly braces. WHILE can execute a block of code repeatedly, as long as expression evaluates to TRUE. If expression is not TRUE, the block of code within the curly braces is not executed, and the next command following the closing curly brace ( } ) is executed.

Programmers must be careful to avoid a WHILE infinite loop. WHILE 1 { ... } (or any other WHILE expression that can never evaluate as false) performs an infinite loop, and therefore the loop code must issue a QUIT to exit the loop.

An opening or closing curly brace may appear on its own code line or on the same line as a command. An opening or closing curly brace may even appear in column 1 (though this is not recommended). It is a recommended programming practice to indent curly braces to indicate the beginning and end of a nested block of code. No whitespace is required before or after an opening curly brace. No whitespace is required before a closing curly brace, including a curly brace that follows an argumentless command. There is only one whitespace requirement for curly braces: a closing curly brace must be separated from the command that follows it by a space, tab, or line return.

The block of code within the curly braces can consist of one or more ObjectScript commands and function calls. This block of code may span several lines. Indents, line returns, and blank spaces are permitted within the block of code. Commands within this code block and arguments within commands may be separated by one or more blank spaces or line returns.

**Arguments**

expression

A boolean test condition. It can take the form of a single expression or a comma-separated list of expressions. Caché executes the WHILE loop if it evaluates expression as TRUE (any non-zero numeric value). Commonly expression is a condition test, such as x<10 or "apple"="apple", but any value that evaluates to a non-zero number is TRUE. For example 7, 0.01, “700”, “7dwarves” all evaluate to TRUE. Any value that evaluates to zero is FALSE. For example, 0, -0, and any non-numeric string all evaluate to FALSE.

For an expression list, Caché evaluates the individual expressions in left-to-right order. It stops evaluation if it encounters an expression that evaluates to 0 (FALSE). Any expressions to the right of an expression that evaluates to FALSE are not validated or tested.

If all expressions evaluate to a non-zero numeric value (TRUE), Caché executes the WHILE loop code block. As long as expression evaluates to TRUE, Caché continues to execute the WHILE loop repeatedly, testing expression at the top of each loop. If any expression evaluates to FALSE, Caché executes the next line of code after the WHILE closing curly brace.
Examples

The following example performs a **WHILE** loop a specified number of times. It tests the expression before executing the loop:

```objectscript
Mainloop
  SET x=1
  WHILE x<10 {
    WRITE !, "Looping", x
    SET x=x+1
  }
  WRITE !,"DONE"
QUIT
```

The following pair of examples perform two expression tests. The two tests are separated by a comma. If both tests evaluate to true, it executes **WHILE** loop. Thus, these programs either return all of the items in a list, or a specified sample size of the items in a list:

```objectscript
SET mylist=$LISTBUILD("a","b","c","d","e")
SET ptr=0,sampcnt=1,sampmax=4
WHILE 1=$LISTNEXT(mylist,ptr,value),sampcnt<sampmax {
  WRITE value," is item ",sampcnt,!
  SET sampcnt=sampcnt+1
}
IF sampcnt<sampmax {WRITE "This is the whole list"} ELSE {WRITE "This is a ",sampcnt-1," item sample of the list"}

SET mylist=$LISTBUILD("a","b","c","d","e")
SET ptr=0,sampcnt=1,sampmax=10
WHILE 1=$LISTNEXT(mylist,ptr,value),sampcnt<sampmax {
  WRITE value," is item ",sampcnt,!
  SET sampcnt=sampcnt+1
}
IF sampcnt<sampmax {WRITE "This is the whole list"} ELSE {WRITE "This is a ",sampcnt-1," item sample of the list"}
```

Notes

**WHILE and FOR**

You can use either a **FOR** or a **WHILE** to perform the same operation: loop until an event causes execution to break out of the loop. However, which loop construct you use has consequences for performing single-step (**BREAK "S+"** or **BREAK "L+"**) debugging on the code module.

A **FOR** loop pushes a new level onto the stack. A **WHILE** loop does not change the stack level. When debugging a **FOR** loop, popping the stack from within the **FOR** loop (using **BREAK "C" GOTO** or **QUIT 1**) allows you to continue single-step debugging with the command immediately following the end of the **FOR** command construct. When debugging a **WHILE** loop, issuing a using **BREAK "C" GOTO** or **QUIT 1** does not pop the stack, and therefore single-step debugging does not continue following the end of the **WHILE** command. The remaining code executes without breaking.

For further details, refer to the **BREAK** command and **Debugging with BREAK** in the “Command-line Routine Debugging” chapter of *Using ObjectScript*.

**WHILE and DO WHILE**

The **WHILE** command tests **expression** before executing the loop. The **DO WHILE** command executes the loop once and then tests **expression**.

**WHILE and CONTINUE**

Within the code block of a **WHILE** command, encountering a **CONTINUE** command causes execution to immediately jump back to the **WHILE** command. The **WHILE** command then evaluates its **expression** test condition, and, based on that evaluation, determines whether to re-execute the code block loop. Thus, the **CONTINUE** command has exactly the same effect on execution as reaching the closing curly brace of the code block.
WHILE, QUIT, and RETURN

The QUIT command within the code block ends the WHILE loop and transfers execution to the command following the closing curly brace, as shown in the following example:

Testloop
  SET x=1
  WHILE x < 10
  |   WRITE !,"Looping",x
  |   QUIT:x=5
  |   SET x=x+1
  | }
  WRITE !,"DONE"

This program writes Looping1 through Looping5 and then DONE.

WHILE code blocks may be nested. That is, a WHILE code block may contain another flow-of-control loop (another WHILE, or a FOR or DO WHILE code block). A QUIT in an inner nested loop breaks out of the inner loop, to the next enclosing outer loop. This is shown in the following example:

Nestedloops
  SET x=1,y=1
  WHILE x<6 {
    WRITE "outer loop ",!
    WHILE y<100 {
      WRITE "inner loop 
      WRITE " y="y,!
      QUIT:y=7
      SET y=y+2
    }
    WRITE "back to outer loop x="x,!!
    SET x=x+1
  }
  WRITE "Done"

You can use RETURN to terminate execution of a routine at any point, including from within a WHILE loop or nested loop structure. RETURN always exits the current routine, returning to the calling routine or terminating the program if there is no calling routine. RETURN always behaves the same, regardless of whether it is issued from within a code block.

WHILE and GOTO

A GOTO command within the block of code may direct execution to a label outside the loop, terminating the loop. A GOTO command within the block of code may direct execution to a label within the same block of code; this label may be in a nested code block.

A GOTO command should not direct execution to a label within another code block. While such a construct may execute, it is considered “illegal” because it defeats the test condition for the code block it is entering.

The following forms of GOTO are legal:

mainloop ; GOTO to outside of the code block
  WHILE 1=1 {
    WRITE !,"In an infinite WHILE loop"
    GOTO label1
    WRITE !,"This should not display"
  }
  WRITE !,"This should not display"
label1
  WRITE !,"Went to label1 and quit"

mainloop ; GOTO to elsewhere within the same code block
  SET x=1
  WHILE x<3 {
    WRITE !,"In the WHILE loop"
    GOTO label1
    WRITE !,"This should not display"
  }
  WRITE !,"Still in the WHILE loop after GOTO"
  SET x=x+1
  WRITE !,"x="x
}
WRITE !,"WHILE loop done"
mainloop ; GOTO from an inner to an outer nested code block
SET x=1,y=1
WHILE x<6 {
    WRITE !,"Outer loop",!
    SET x=x+1
    label1
    WRITE "outer loop iteration ",x-1,!
    WHILE y<4 {
        WRITE !," Inner loop iteration ",y,!
        SET y=y+1
        WRITE " return to "
        GOTO label1
        WRITE " This should not display",!
    }
    WRITE "Inner loop completed",!
}
WRITE "All done"

mainloop ; GOTO from an outer to an inner nested code block
SET x=1,y=1
WHILE x<6 {
    WRITE !,"Outer loop",!
    SET x=x+1
    WRITE "outer loop iteration ",x-1,!
    WRITE "Jumping into the "
    GOTO label1
    WRITE "This should not display",!
    WHILE y<4 {
        WRITE !," Inner loop iteration ",y,!
        SET y=y+1
    }
    WRITE "inner loop ",!
    WRITE "Inner loop completed",!
}
WRITE "All done"

The following forms of GOTO may execute, but they are considered “illegal” because they defeat (ignore) the condition test for the block that the GOTO enters into:

mainloop ; GOTO into a code block
SET x=1
WRITE "Jumped into the "
GOTO label1
WHILE x>1,x<6 {
    WRITE "Top of WHILE loop x="x,!
label1
    WRITE "Bottom of WHILE loop x="x,!!
    SET x=x+1
}

mainloop ; GOTO from a code block into an IF clause block
SET x=1
WHILE x<6 {
    WRITE !,"WHILE loop interation="x,!
    SET x=x+1
    GOTO label1
    WRITE "This should never display",!
    IF x#2 { WRITE "in the IF clause",!
    label1
        WRITE "GOTO entry into the IF clause"!,!
        WRITE x," is an odd number",!
    }
    ELSE {WRITE "in the ELSE clause",!
            WRITE x," is an even number",!
        WRITE "Bottom of WHILE loop",!
    }
    WRITE "All done"
}

See Also
• DO WHILE command
• FOR command
• IF command
• CONTINUE command
ObjectScript Commands

- **GOTO** command
- **QUIT** command
- **RETURN** command
WRITE

Displays output to current device.

```plaintext
WRITE:pc writeargument,...
W:pc writeargument,...
```

where `writeargument` can be:

- `expression`
- `f`
- `*integer`
- `*-integer`

### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pc</code></td>
<td>Optional — A postconditional expression.</td>
</tr>
<tr>
<td><code>expression</code></td>
<td>Optional — The value to write to the output device. Any valid ObjectScript expression, including literals, variables, object methods, and object properties that evaluates to either a numeric or a quoted string.</td>
</tr>
<tr>
<td><code>f</code></td>
<td>Optional — One or more format control characters that position the output on the target device. Format control characters include !, #, ?, and /.</td>
</tr>
<tr>
<td><code>*integer</code></td>
<td>Optional — An integer code representing a character to write to the output device. For ASCII, integers in the range 0 to 255; for Unicode, integers in the range 0 to 65534. Any valid ObjectScript expression that evaluates to an integer in the appropriate range. The asterisk is mandatory.</td>
</tr>
<tr>
<td><code>*-integer</code></td>
<td>Optional — A negative integer code specifying a device control operation. The asterisk is mandatory.</td>
</tr>
</tbody>
</table>

### Description

The `WRITE` command displays the specified output on the current I/O device. (To set the current I/O device, use the `USE` command, which sets the value of the `SIO` special variable.) `WRITE` has two forms:

- **WRITE without an argument**
- **WRITE with arguments**

#### Argumentless `WRITE`

Argumentless `WRITE` lists the names and values of all defined local variables. It does not list process-private globals, global variables, or special variables. It lists defined local variables one variable per line in the following format:

```
varname1=value1
varname2=value2
```

Argumentless `WRITE` displays local variable values of all types as quoted strings. The exceptions are canonical numbers and object references. A canonical number is displayed without enclosing quotes. An object reference (OREF) is displayed as follows: `myoref=<OBJECT REFERENCE>[1@%SQL.Statement];` a JSON array or JSON object is displayed as an object reference (OREF). Bit string values and List values are displayed as quoted strings with the data value displayed in encoded form.
The display of numbers and numeric strings is shown in the following example:

```
SET str="fred"
SET num=+123.40
SET canonstr="456.7"
SET noncanon1="789.0"
SET noncanon2="+999"
WRITE

canonstr=456.7
noncanon1="789.0"
noncanon2="+999"
num=123.4
str="fred"
```

Argumentless WRITE displays local variables in case-sensitive string collation order, as shown in the following WRITE output example:

```
A="Apple"
B="Banana"
a= "apple varieties"
a1="macintosh"
a10="winesap"
a19="northern spy"
a2="golden delicious"
aa= "crabapple varieties"
```

Argumentless WRITE displays the subscripts of a local variable in subscript tree order, using numeric collation, as shown in the following WRITE output example:

```
a[1]="United States"
a[1,1]="Northeastern Region"
a[1,1,1]="Maine"
a[1,2]="New Hampshire"
a[1,2,1]="Southeastern Region"
a[1,2,1,1]="Florida"
a[2]="Canada"
a[2,1]="Maritime Provinces"
a[10]="Argentina"
```

Argumentless WRITE executes control characters, such as Formfeed ($CHAR(12)) and Backspace ($CHAR(8)). Therefore, local variables that define control characters would display as shown in the following example:

```
SET name="fred"
SET number=123
SET bell=$CHAR(7)
SET formfeed=$CHAR(10)
SET backspace=$CHAR(8)
WRITE

backspace="
bell="
formfeed="
name="fred"
number=123
```

Multiple backspaces display as follows, given a local variable named back: 1 backspace: back="; 2 backspaces: back""; 3 backspaces: back*="; 4 backspaces: back"k="; 5 backspaces: back"ck="; 6 backspaces: back"ck="; 7 or more backspaces: back"ck=".

An argumentless WRITE must be separated by at least two blank spaces from a command following it on the same line. If the command that follows it is a WRITE with arguments, you must provide the WRITE with arguments with the appropriate line return f format control arguments. This is shown in the following example:

```
SET myvar="fred"
WRITE WRITE ; note two spaces following argumentless WRITE
WRITE WRITE myvar ; formatting needed
WRITE WRITE ,myvar ; formatting provided
```

Argumentless WRITE listing can be interrupted by issuing a CTRL-C, generating an <INTERRUPT> error.
You can use argumentless **WRITE** to display all defined local variables. You can use the **$ORDER** function to return a limited subset of the defined local variables.

**WRITE with Arguments**

**WRITE** can take a single *writeargument* or a comma-separated list of *writearguments*. A **WRITE** command can take any combination of *expression*, *f*, *integer*, and *-integer* arguments.

- **WRITE expression** displays the data value corresponding to the *expression* argument. An *expression* can be the name of a variable, a literal, or any expression that evaluates to a literal value.
- **WRITE *f*** provides any desired *output formatting*. Because the argumented form of **WRITE** provides no automatic formatting to separate argument values or indicate strings, *expression* values will display as a single string unless separated by *f formatting*.
- **WRITE *integer*** displays the character represented by the integer code.
- **WRITE *-integer*** provides device control operations.

**WRITE** arguments are separated by commas. For example:

```plaintext
WRITE "numbers",1,2,3
WRITE "letters","ABC"
```

displays as:

terms123lettersABC

Note that **WRITE** does not append a line return to the end of its output string. In order to separate **WRITE** outputs, you must explicitly specify *f argument formatting characters*, such as the line return (!) character.

```plaintext
WRITE "numbers",1,2,3,!
WRITE "letters","ABC"
```

displays as:

```
terms 123
letters ABC
```

**Arguments**

**pc**

An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). You can specify a postconditional expression for an argumentless **WRITE** or a **WRITE** with arguments. For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

**expression**

The value you wish to display. Most commonly this is either a literal (a quoted string or a numeric) or a variable. However, *expression* can be any valid ObjectScript expression, including literals, variables, arithmetic expressions, object methods, and object properties. For more information on expressions, see Using Caché ObjectScript.

An *expression* can be a variable of any type, including local variables, process-private globals, global variables, and special variables. Variables can be subscripted; **WRITE** only displays the value of the specified subscript node.

Data values, whether specified as a literal or a variable, are displayed as follows:

- **Character strings** display without enclosing quotes. Some non-printing characters do not display: $CHAR 0, 1, 2, 14, 15, 28, 127. Other non-printing characters display as a placeholder character: $CHAR 3, 16–26. Control characters are executed: $CHAR 7–13, 27. For example, $CHAR(8) performs a backspace, $CHAR(11) performs a vertical tab.
• **Numbers** display in canonical form. Arithmetic operations are performed.

• **Extended global references** display as the value of the global, without indicating the namespace in which the global variable is defined. If you specify a nonexistent namespace, Caché issues a `<NAMESPACE>` error. If you specify a namespace for which you do not have privileges, Caché issues a `<PROTECT>` error, followed by the global name and database path, such as the following: `<PROTECT> ^myglobal,c:\intersystems\cache\mgr\`.

• **Caché List** structured data displays in encoded form.

• **Caché bitstrings** display in encoded form.

• **Object References** display as the OREF value. For example, `##class(%SQL.Statement).%New()` displays as the OREF `2@%SQL.Statement`. A JSON dynamic object or a JSON dynamic array displays as an OREF value. For information on OREFs, see “OREF Basics” in Using Caché Objects.

• **%Status** displays as either 1 (success), or a complex encoded failure status, the first character of which is 0.

---

**f**

A format control to position the output on the target device. You can specify any combination of format control characters without intervening commas, but you must use a comma to separate a format control from an expression. For example, when you issue the following **WRITE** to a terminal:

```
WRITE #!!!?6,"Hello",!,"world!"
```

The format controls position to the top of a new screen (#), then issue three line returns (!!!), then indent six columns (?6). The **WRITE** then displays the string **Hello**, performs a format control line return (!), then displays the string **world!**. Note that the line return repositions to column 1; thus in this example, **Hello** is displayed indented, but **world!** is not.

Format control characters cannot be used with an argumentless **WRITE**.

For further details, see Using Format Controls with WRITE.

---

**integer**

The **integer** argument allows you to use a positive integer code to write a character to the current device. It consists of an asterisk followed by any valid ObjectScript expression that evaluates to a positive integer that corresponds to a character. The **integer** argument may correspond to a printable character or a control character. An integer in the range of 0 through 255 evaluates to the corresponding 8-bit ASCII character. An integer in the range of 256 through 65534 evaluates to the corresponding 16-bit Unicode character (on Unicode systems).

As shown in the following example, **integer** can specify an integer code, or specify an expression that resolves to an integer code. The following examples all return the word “Caché”:

```
WRITE !,"Cach",*233
WRITE !,*67,*97,*99,*104,*233
SET accent=233
WRITE !,"Cach",*accent ; variables are evaluated
WRITE !,"Cach",*232+1 ; arithmetic operations are evaluated
WRITE !,"Cach",*00233.999 ; fractional numbers are truncated to integers
```

To write the name of the composer Anton Dvorak with the proper Czech accent marks, use:

```
IF $SYSTEM.Version.IsUnicode() {
  WRITE "Anton Dvo",*345,*225,"k"
} ELSE {WRITE "This example requires a Unicode installation of Caché"}
```

The integer resulting from the expression evaluation may correspond to a control character. Such characters are interpreted according to the target device. A **integer** argument can be used to insert control characters (such as the form feed: *12) which govern the appearance of the display, or special characters such as *7, which rings the bell on a terminal.

For example, if the current device is a terminal, the integers 0 through 30 are interpreted as ASCII control characters. The following commands send ASCII codes 7 and 12 to the terminal.
WRITE *7 ; Sounds the bell
WRITE *12 ; Form feed (blank line)

Here’s an example combining expression arguments with *integer specifying the form feed character:

WRITE "stepping","down","the","stairs"

*integer and $X, $Y

An integer expression does not change the $X and $Y special variables when writing to a terminal. Thus, WRITE "a" and WRITE $CHAR(97) both increment the column number value contained in $X, but WRITE *97 does not increment $X.

You can issue a backspace (ASCII 8), a line feed (ASCII 10), or other control character without changing the $X and $Y values by using *integer. The following Terminal examples demonstrate this use of integer expressions.

Backspace:

WRITE $X,"/","\dollar CHAR(8),",$X   ; displays: 01
WRITE $X,"/","*8,5$,X               ; displays: 02

Linefeed:

WRITE $Y,"\dollar CHAR(10),",$Y   /* displays: 1 2 */
WRITE $Y,"10,",$Y                   /* displays: 4 */

For further details, see the $X and $Y special variables, and “Terminal I/O” in Caché I/O Device Guide.

*-integer

An asterisk followed by a negative integer is a device control code. WRITE supports the following general device control codes:

<table>
<thead>
<tr>
<th>Code</th>
<th>Device Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>*-1</td>
<td>Clears the input buffer upon the next READ.</td>
</tr>
<tr>
<td>*-2</td>
<td>Disconnects a TCP device or a named pipe. See TCP Client/Server Communication and Local Interprocess Communication in the Caché I/O Device Guide.</td>
</tr>
<tr>
<td>*-3</td>
<td>Flushes the output buffer to the device. This forces a write to the file on disk.</td>
</tr>
<tr>
<td>*-9</td>
<td>Truncates the contents of a sequential file at the current file pointer position. In order to truncate a file, the file must be open (using the OPEN command with at least “RW” access) and must be established as the current device (using the USE command). See Sequential File I/O in the Caché I/O Device Guide.</td>
</tr>
<tr>
<td>*-10</td>
<td>Clears the input buffer immediately.</td>
</tr>
<tr>
<td>*-99</td>
<td>Sends compressed stream data. See TCP Client/Server Communication in the Caché I/O Device Guide.</td>
</tr>
</tbody>
</table>

WRITE also supports a number of device control codes for magnetic tape devices. These device control codes are described in the Magnetic Tape I/O chapter of the Caché I/O Device Guide.

Input Buffer Controls

The *-1 and *-10 controls are used for input from a terminal device. These controls clear the input buffer of any characters that have not yet been accepted by a READ command. The *-1 control clears the input buffer upon the next READ. The *-10 control clears the input buffer immediately. If there is a pending CTRL-C interrupt when WRITE *-1 or WRITE *-10 is invoked, WRITE dismisses this interrupt before clearing the input buffer.
An input buffer holds characters as they arrive from the keyboard, even those the user types before the routine executes a **READ** command. In this way, the user can type-ahead the answers to questions even before the prompts appear on the screen. When the **READ** command takes characters from the buffer, Caché echoes them to the terminal so that questions and answers appear together. When a routine detects errors it may use the ***-1** or ***-10** control to delete these type-ahead answers. For further details, see *Terminal I/O* in the *Caché I/O Device Guide*.

For use of ***-1** in **TCP Client/Server Communication** refer to the *Caché I/O Device Guide*.

### Output Buffer Controls

The ***-3** control is used to flush data from an output buffer, forcing a write operation on the physical device. Thus it first flushes data from the device buffer to the operating system I/O buffer, then forces the operating system to flush its I/O buffer to the physical device. This control is commonly used when forcing an immediate write to a sequential file on disk. ***-3** is supported on Windows and UNIX platforms. On other operating system platforms it is a no-op.

For use of ***-3** in **TCP Client/Server Communication** refer to the *Caché I/O Device Guide*.

### Examples

In the following example, the **WRITE** command sends the current value in variable *var1* to the current output device.

```ObjectScript
SET var1="hello world"
WRITE var1
```

In the following example, both **WRITE** commands display the Unicode character for pi. The first uses the **$CHAR** function, the second a **"integer** argument:

```ObjectScript
IF $SYSTEM.Version.IsUnicode() {
  WRITE !,$CHAR(960)
  WRITE !,*960
} ELSE {WRITE "This example requires a Unicode installation of Caché")
```

(Note that the above example requires a Unicode installation of Caché.)

The following example writes first name and last name values along with an identifying text for each. The **WRITE** command combines multiple arguments on the same line. It is equivalent to the two **WRITE** commands in the example that follows it. The **!** character is a format control that produces a line break. (Note that the **!** line break character is still needed when the text is output by two different **WRITE** commands.)

```ObjectScript
SET fname="Bertie"
SET lname="Wooster"
WRITE "First name: ",fname,!,"Last name: ",lname
```

is equivalent to:

```ObjectScript
SET fname="Bertie"
SET lname="Wooster"
WRITE "First name: ",fname,!
WRITE "Last name: ",lname
```

In the following example, assume that the current device is the user’s terminal. The **READ** command prompts the user for first name and last name and stores the input values in variables *fname* and *lname*, respectively. The **WRITE** command displays the values in *fname* and *lname* for the user’s confirmation. The string containing a space character (" ") is included to separate the output names.

```ObjectScript
Test
  READ !,"First name: ",fname
  READ !,"Last name: ",lname
  WRITE !,fname," ",lname
  READ !,"Is this correct? (Y or N) ",check
  IF "Nn"[check {
    GOTO Test
  }
```

The following example writes the current values in the client(1,n) nodes.
The following command writes the current value of an object property:

```objectscript
WRITE person.LastName
```

where `person` is the object reference, and `LastName` is the object property name. Note that dot syntax is used in object expressions; a dot is placed between the object reference and the object property name or object method name.

The following command writes the value returned by the object method `TotalLines()`:

```objectscript
WRITE invoice.TotalLines()
```

A write command for objects can take an expression with cascading dot syntax, as shown in the following example:

```objectscript
WRITE patient.Doctor.Hospital.Name
```

In this example, the `patient.Doctor` object property references the `Hospital` object, which contains the `Name` property. Thus, this command writes the name of the hospital affiliated with the doctor of the specified patient. The same cascading dot syntax can be used with object methods.

A write command for objects can be used with system-level methods, such as the following data type property method:

```objectscript
WRITE patient.AdmitDateIsValid(date)
```

In this example, the `AdmitDateIsValid()` property method returns its result for the current patient object. `AdmitDateIsValid()` is a boolean method for data type validation of the `AdmitDate` property. Thus, this command writes a 1 if the specified date is a valid date, and writes 0 if the specified date is not a valid date.

Note that any object expression can be further specified by declaring the class or superclass to which the object reference refers. Thus, the above examples could also be written:

```objectscript
WRITE ##class(Patient)patient.Doctor.Hospital.Name
WRITE ##class(Patient)patient.AdmitDateIsValid(date)
```

**Notes**

**WRITE with $X and $Y**

A `WRITE` displays the characters resulting from the expression evaluation one at a time in left-to-right order. Caché records the current output position in the `$X` and `$Y` special variables, with `$X` defining the current column position and `$Y` defining the current row position. As each character is displayed, `$X` is incremented by one.

In the following example, the `WRITE` command gives the column position after writing the 11-character string `Hello world`.

```objectscript
WRITE "Hello world"," ",$_X," is the column number"
```

Note that writing a blank space between the displayed string and the `$X` value (" ", $X) would cause that blank space to increment $X before it is evaluated; but concatenating a blank space to $X (" ",$_X) displays the blank space, but does not increment the value of $X before it is evaluated.
Even using a concatenated blank, the display from $X$ or $Y$ does, of course, increment $X$, as shown in the following example:

```
WRITE $Y," 
WRITE $X," 
```

In the first WRITE, the value of $X$ is incremented by the number of digits in the $Y$ value (which is probably not what you wanted). In the second WRITE, the value of $X$ is 0.

With $X$ you can display the current column position during a WRITE command. To control the column position during a WRITE command, you can use the ? format control character. The ? format character is only meaningful when $X$ is at column 0. In the following WRITE commands, the performing indenting:

```
WRITE ?5,"Hello world",!
WRITE "Hello",!,?5,"world"
```

**Using Format Controls with WRITE**

The $f$ argument allows you to include any of the following format control characters. When used with output to the terminal, these controls determine where the output data appears on the screen. You can specify any combination of format control characters.

**! Format Control Character**

Advances one line and positions to column 0 ($Y$ is incremented by 1 and $X$ is set to 0). The actual control code sequence is device-dependent; it generally either ASCII 13 (RETURN), or ASCII 13 and ASCII 10 (LINE FEED).

Caché does not perform an implicit new line sequence for WRITE with arguments. When writing to a terminal it is a good general practice to begin (or end) every WRITE command with a ! format control character.

You can specify multiple ! format controls. For example, to advance five lines, WRITE !!!!!. You can combine ! format controls with other format controls. However, note that the following combinations, though permitted, are not in most cases meaningful: !# or !,# (advance one line, then advance to the top of a new screen, resetting $Y$ to 0) and ?5,! (indent by 5, then advance one line, undoing the increment). The combination ?5! is not legal.

If the current device is a TCP device, ! does not output a RETURN and LINE FEED. Instead, it flushes any characters that remain in the buffer and sends them across the network to the target system.

**# Format Control Character**

Produces the same effect as sending the CR (ASCII 13) and FF (ASCII 12) characters to a pure ASCII device. (The exact behavior depends on the operating system type, device, and record format.) On a terminal, the # format control character clears the current screen and starts at the top of the new screen in column 0. ($Y$ and $X$ are reset to 0.)

You can combine # format controls with other format controls. However, note that the following combinations, though permitted, are not in most cases meaningful: !# or !,# (advance one line, then advance to the top of a new screen, resetting $Y$ to 0) and ?5,# (indent by 5, then advance to the top of a new screen, undoing the increment). The combination ?5# is not legal.

**?n Format Control Character**

This format control consists of a question mark (?) followed by an integer, or an expression that evaluates to an integer. It positions output at the $n$th column location (counting from column 0) and resets $X$. If this integer is less than or equal to the current column location ($n<$X), this format control has no effect. You can reference the $X$ special variable (current column) when setting a new column position. For example, ?$X+3.
/mnemonic Format Control Character

This format control consists of a slash (/) followed by a mnemonic keyword, and (optionally) a list parameters to be passed to the mnemonic.

```
/mnemonic(param1,param2,...)
```

Caché interprets `mnemonic` as an entry point name defined in the active mnemonic space. This format control is used to perform such device functions as rewinding a magnetic tape or positioning the cursor on a screen. If there is no active mnemonic space, an error results. Amnemonic may (or may not) require a parameter list.

You can establish the active mnemonic space in either of the following ways:

- Go to the Management Portal, select [System] > [Configuration] > [Device Settings] > [IO Settings]. View and edit the File, MagTape, Other, or Terminal mnemonic space setting.
- Include the /mnemonic space parameter in the OPEN or USE command for the device.

The following are some examples of mnemonic device functions:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/IC(n)</td>
<td>Inserts spaces for ( n ) characters at the current cursor location, moving the rest of the line to the right</td>
</tr>
<tr>
<td>/DC(n)</td>
<td>Deletes ( n ) characters to the right of the cursor and collapses the line</td>
</tr>
<tr>
<td>/EC(n)</td>
<td>Erases ( n ) characters to the right of the cursor, leaving blanks in their stead</td>
</tr>
</tbody>
</table>

For further details on mnemonics, see the Caché I/O Device Guide.

**Specifying a Sequence of Format Controls**

Caché allows you to specify a sequence of format controls and to intersperse format controls and expressions. When specifying a sequence of format controls it is not necessary to include the comma separator between them (though commas are permitted.) A comma separator is required to separate format controls from expressions.

In the following example, the WRITE command advances the output by two lines and positions the first output character at the column location established by the input for the READ command.

```objectscript
READ !,"Enter the number: ",num
SET col=$X
SET ans=num*num*num
WRITE !!,"Its cube is: ",?col,ans
```

Thus, the output column varies depending on the number of characters input for the READ.

Commonly, format controls are specified as literal operands for each WRITE command. You cannot specify format controls using variables, because they will be parsed as strings rather than executable operands. If you wish to create a sequence of format controls and expressions to be used by multiple WRITE commands, you can use the #Define preprocessor directive to define a macro, as shown in the following example:

```objectscript
#Define WriteMacro "IF YOU ARE SEEING THIS",!,"SOMETHING HAS GONE WRONG",##Continue
$SYSTEM.Status.DisplayError($SYSTEM.Status.Error(x)),!!
SET x=83
Module1
    /* code */
    WRITE $$WriteMacro
Module2
    /* code */
    WRITE $$WriteMacro
```
Escape Sequences with WRITE

The **WRITE** command, like the **READ** command, provides support for escape sequences. Escape sequences are typically used in format and control operations. Their interpretation is specific to the current device type.

To output an escape sequence, use the form:

```
WRITE *27,"char"
```

where *27 is the ASCII code for the escape character, and *char* is a literal string consisting of one or more control characters. The enclosing double quotes are required.

For example, if the current device is a VT-100 compatible terminal, the following command erases all characters from the current cursor position to the end of the line.

```
WRITE *27,"[2J"
```

To provide device independence for a program that can run on multiple platforms, use the **SET** command at the start of the program to assign the necessary escape sequences to variables. In your program code, you can then reference the variables instead of the actual escape sequences. To adapt the program for a different platform, simply make the necessary changes to the escape sequences defined with the **SET** command.

WRITE Compared with Other Write Commands

For a comparison of **WRITE** with the **ZWRITE**, **ZZDUMP**, and **ZZWRITE** commands, refer to the Display (Write) Commands features tables in the “Commands” chapter of *Using Caché ObjectScript*.

See Also

- **USE** command
- **READ** command
- **ZWRITE** command
- **ZZDUMP** command
- **ZZWRITE** command
- **$X** special variable
- **$Y** special variable
- Writing escape sequences for Terminal I/O and Interprocess Communications in the *Caché I/O Device Guide*
- Terminal I/O in *Caché I/O Device Guide*
- Magnetic Tape I/O in *Caché I/O Device Guide*
- Sequential File I/O in *Caché I/O Device Guide*
- The Spool Device in *Caché I/O Device Guide*
**XECUTE**

Executes the specified commands.

```
XECUTE:pc xecutearg,...
X:pc xecutearg,...
```

where `xecutearg` can be either of the following:

```
"cmdline":pc
("(fparams) cmdline",params):pc
```

### Arguments

<table>
<thead>
<tr>
<th>pc</th>
<th>Optional — A postconditional expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmdline</td>
<td>An expression that resolves to a command line consisting of one or more valid ObjectScript commands. Note that the <code>cmdline</code> or <code>(fparams) cmdline</code> must be specified as a quoted string.</td>
</tr>
<tr>
<td>fparams</td>
<td>Optional — A formal parameters list, specified as a comma-separated list enclosed in parentheses. Formal parameters are variables use by <code>cmdline</code>, the values of which are supplied by passing <code>params</code>. Note that the <code>fparams</code> are the first item within the quoted code string.</td>
</tr>
<tr>
<td>params</td>
<td>Optional — A parameters list, specified as a comma-separated list. These are the parameters passed to <code>fparams</code>. If <code>params</code> are specified, an equal or greater number of <code>fparams</code> must be specified.</td>
</tr>
</tbody>
</table>

### Description

**XECUTE** executes one or more ObjectScript command lines, each command line specified by an `xecutearg`. You can specify multiple `xecuteargs`, separated by commas. These `xecutearg` are executed in left-to-right sequence, the execution of each being governed by an optional postconditional expression. There are two syntactical forms of `xecutearg`:

- Without parameter passing. This form uses no parentheses.
- With parameter passing. This form requires enclosing parentheses.

An **XECUTE** can contain any combination of these two forms of `xecutearg`. You can use the `CheckSyntax()` method of the `%Library.Routine` class to perform syntax checking on an `xecutearg` command line string.

In effect, each `xecutearg` is like a one-line subroutine called by a **DO** command and terminated when the end of the argument is reached or a **QUIT** command is encountered. After Caché executes the argument, it returns control to the point immediately after the `xecutearg`.

Each invocation of **XECUTE** places a new context frame on the call stack for your process. The `$STACK` special variable contains the current number of context frames on the call stack.

The **XECUTE** command performs substantially the same operation as the `$XECUTE` function, with the following differences: The command can use postconditionals, the function cannot. The command can specify multiple `xecuteargs`, the function can specify only one `xecutearg`. The command does not require a **QUIT** to complete execution; the function requires an argumented **QUIT** for every execution path.
Arguments

`pc`

An optional postconditional expression. If a postconditional expression is appended to the command keyword, Caché only executes the `XECUTE` command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the `XECUTE` command if the postconditional expression is false (evaluates to zero).

If a postconditional expression is appended to an `xecutearg`, Caché evaluates the argument only if the postconditional expression is true (evaluates to a nonzero numeric value). If the postconditional expression is false, Caché skips that `xecutearg` and evaluates the next `xecutearg` (if one exists). For further details, refer to Command Postconditional Expressions in *Using Caché ObjectScript*.

`cmdline`

Each `cmdline` must evaluate to a string containing one or more ObjectScript commands. Note that in some cases two spaces must be inserted between a command and the command following it. The `cmdline` string must not contain a tab character at the beginning or a <Return> at the end. To specify quotation marks within the `cmdline` string, double the quotation marks.

The following example shows a single `cmdline` containing two commands:

```
XECUTE "WRITE ""hello ",! WRITE ""world",

Because a `cmdline` is a string, it cannot be simply broken across multiple code lines. You can divide a single `cmdline` argument into separate strings joined with the concatenate operator:

```
XECUTE "WRITE ""hello ",!
     WRITE ""world",!
```

You can divide a single `cmdline` argument into multiple separate comma-separated `cmdline` arguments:

```
XECUTE "WRITE ""hello ",!
     WRITE ""world",!
```

The maximum length for `cmdline` depends on the following considerations: Caché stores both the source `cmdline` string and its generated object code as a single string. This resulting string must not exceed the Caché maximum string length. For further details on maximum string length, refer to Long Strings in *Using Caché ObjectScript*.

You can embed /* text */ comments within a `cmdline`, between concatenated `cmdline` strings, or between comma-separated `cmdline` arguments:

```
XECUTE "SET x=""hello "/* 1st val */ SET y=""world"" /* 2nd val */ "
     WRITE x,!* part of 1st cmdline */ ",
     WRITE y,!* 2nd cmdline */ "
```

A `cmdline` can evaluate to a null string (""). In this case, Caché performs no action and continues execution with the next `xecutearg` (if one exists).

If you are passing parameters, the `fparams` formal parameter list must precede the `cmdline` commands, with both elements enclosed in the same quotation marks. While it is recommended that you separate `fparams` from the `cmdline` by one or more spaces, no space is required.

```
SET x=1
XECUTE "((in,out) SET out=in+3", x, .y)
WRITE y
QUIT
```

By default, all local variables used in `cmdline` are public variables. You can designate variables within the command line as private variables by enclosing the command setting them within curly braces. For example:

```
SET x=1
XECUTE "((in,out) { SET out=in+3 }", x, .y)
WRITE y
QUIT
You can override this designation of private variables for specific variables by specifying a public variable list, enclosed in square brackets, immediately after the \texttt{fparams} formal parameter list. The following example specifies a public variable list containing the variable \(x\):

\[
\begin{aligned}
\text{SET } x &= 1 \\
\text{XECUTE } &\{\text{\texttt{(in,out)} } [x] \text{ \{ SET out=\texttt{in}+3 \} }, \text{ x, .y} \} \\
\text{WRITE } y \\
\text{QUIT}
\end{aligned}
\]

\textbf{fparams}

A list of formal parameters, separated by commas and enclosed by parentheses. Formal parameter names must be valid identifiers. Because these formal parameters are executed in another context, they must only be unique within their \texttt{xecutearg}; they have no effect on local variables with the same name in the program that issued the \texttt{XECUTE}, or in another \texttt{xecutearg}. You do not have to use any or all of the \texttt{fparams} in \texttt{cmdline}. However, the number of \texttt{fparams} must equal or exceed the number of \texttt{params} specified, or a \texttt{<PARAMETER>} error is generated.

\textbf{params}

The actual parameters to be passed from the invoking program to \texttt{fparams}, specified as a comma-separated list. The \texttt{params} must be defined variables within the calling program.

You can use a dot prefix to pass a parameter by reference. This is useful for passing a value out from a \texttt{cmdline}. An example is provided below. For further details, refer to Passing by Reference in the “User-defined Code” chapter of Using Caché ObjectScript.

\textbf{Examples}

The following example passes a parameter to a command line that sets a global. Two command lines are provided. Execution of each depends upon their postconditional setting.

\[
\begin{aligned}
\text{SET } &\text{bad}=0, \text{good}=1 \\
\text{SET } &\text{val}="\text{paid in full}"
\end{aligned}
\]

\[
\begin{aligned}
\text{XECUTE } &\{\text{\texttt{(pay)} SET ”acct1=\text{pay}”,val}:\text{bad}, \text{\texttt{(pay)} SET ”acct2=\text{pay}”,val}:\text{good}
\end{aligned}
\]

Here the first \texttt{xecutearg} is skipped because of the value of the \textit{bad} postconditional. The second \texttt{xecutearg} is executed with \textit{val} being passed in as a parameter, supplying a value to the \textit{pay} formal parameter used in the command line.

The following example uses passing by reference (.y) to pass a local variable value from the \textit{cmdline} to the invoking context.

\texttt{CubeIt}

\[
\begin{aligned}
\text{SET } x &= 5 \\
\text{XECUTE } &\{\text{\texttt{(in,out)} SET out=\texttt{in}^3”,x,.y} \}
\end{aligned}
\]

\[
\begin{aligned}
\text{WRITE } !,x,"\text{ cubed is ”},y
\end{aligned}
\]

In the following example, the \texttt{XECUTE} command references the local variables \(x\) and \(y\). \(x\) and \(y\) each contain a string literal consisting of three separate ObjectScript commands that \texttt{XECUTE} invokes.

\[
\begin{aligned}
\text{SET } x &= ”\text{SET } \texttt{id=ans} \texttt{QUIT:ans=**** DO Idcheck}” \\
\text{SET } y &= ”\text{SET } \texttt{acct=num} \texttt{QUIT:acct=**** DO Actcheck}” \\
\text{XECUTE } &x,y
\end{aligned}
\]

The following example uses \texttt{XECUTE} with a \texttt{$SELECT} construction.

\[
\begin{aligned}
\text{XECUTE } &”\text{SET } A=$SELECT(A>100:B,1:D)”
\end{aligned}
\]

The following example executes the subroutine that is the value of \(A\).

\[
\begin{aligned}
\text{SET } A &= ”\text{WRITE } ! \text{ FOR } I=1:1:5 \{ \text{WRITE } ?I*5,I+1 \}” \\
\text{XECUTE } &A
\end{aligned}
\]
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 an XECUTE argument contains a 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 an XECUTE command contains a 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

If an XECUTE argument contains a GOTO command, Caché transfers control to the point specified in the GOTO argument. When it encounters a QUIT, it does not return to the point immediately following the GOTO argument that caused the transfer. Instead, Caché returns control to the point immediately following the XECUTE argument that contained the GOTO.

In the following example, Caché transfers control to the routine ROUT and returns control to the point immediately following the XECUTE argument to write the string “FINISH”. It never writes the string “DONE”.

```
XECUTE "GOTO ^ROUT WRITE !,""DONE"" WRITE !,"FINISH"
```

XECUTE and QUIT

There is an implied QUIT at the end of each XECUTE argument.

XECUTE with $TEXT

If you include a $TEXT function within a cmdLine, it designates lines of code in the routine that contains the XECUTE.

For example, in the following program, the $TEXT function retrieves and executes a line.

```
A
  SET H="WRITE !,"$PIECE($TEXT(HELP+1),","",3)"
  XECUTE H
QUIT
HELP
   ;; ENTER A NUMBER FROM 1 TO 5
```

Running routine A extracts and writes “ENTER A NUMBER FROM 1 TO 5”.

Nested Invocation of XECUTE

ObjectScript supports the use of XECUTE within an 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.

**Implementing Generalized Operations**

A typical use for XECUTE is to implement generalized operations within an application. For example, assume that you want to implement an inline mathematical calculator that would allow the user to perform mathematical operations on any two numbers and/or variables. To make the calculator available from any point in the application, you might use a specific function key (say, F1) to trigger the calculator subroutine.

A simplified version of the code to implement such a calculator might appear as follows.

```objectscript
Start  SET ops=$CHAR(27,21)
      READ !,"Total amount (or F1 for Calculator): ",amt
      IF $ZB=ops { DO Calc
                  ; . . .
Calc  READ !,"Calculator"
      READ !,"Math operation on two numbers and/or variables."
      READ !,"First number or variable name: ",inp1
      READ !,"Mathematical operator (+,-,*,/): ",op
      READ !,"Second number or variable name: ",inp2
      SET doit="SET ans=_inp1_op_inp2"
      XECUTE doit
      WRITE !,"Answer (ans) is: ",ans
      READ !,"Repeat? (Y or N) ",inp
      IF (inp="Y")!(inp="y") { GOTO Calc+2 }
      QUIT
```

When executed, the Calc routine accepts the user inputs for the numbers and/or variables and the desired operation and stores them as a string literal defining the appropriate SET command in variable doit. The XECUTE command references doit and executes the command string that it contains. This code sequence can be called from any number of points in the application, with the user supplying different inputs each time. The XECUTE performs the SET command each time, using the supplied inputs.

**XECUTE and ZINSERT**

You use the XECUTE command to define and insert a single line of executable code from within a routine. You can use the ZINSERT command from the programmer prompt to define and insert by line position a single line of executable code into the current routine. You can use the ZREMOVE command from the programmer prompt to delete by line position one or more lines of executable code from the current routine.

An XECUTE command cannot be used to define a new label. Therefore, XECUTE does not require an initial blank space before the first command in its code line. ZINSERT can be used to define a new label. Therefore, ZINSERT does require an initial blank space (or the name of a new label) before the first command in its command line.

**See Also**

- DO command
- GOTO command
- QUIT command
- ZINSERT command
- $TEXT function
- $EXECUTE function
- $STACK special variable
**ZKILL**

Deletes a node while preserving the node’s descendants.

```
ZKILL:pc array-node,...
ZK:pc array-node,...
```

**Arguments**

<table>
<thead>
<tr>
<th>pc</th>
<th>Optional — A postconditional expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>array-node</td>
<td>A local variable, a process-private global, or a global that is an array node, or a comma-separated list of local, process-private global, or global array nodes.</td>
</tr>
</tbody>
</table>

**Description**

The **ZKILL** command removes the value of a specified `array-node` without killing that node’s descendants. In contrast, the **KILL** command removes the value of a specified array node and all of that node’s descendants. An array node can be a local variable, a process-private variable, or a global variable.

By default, any subsequent reference to this killed `array-node` generates an `<UNDEFINED>` error. You can change Caché behavior to not generate an `<UNDEFINED>` error when referencing an undefined subscripted variable by setting the `%SYSTEM.Process.Undefine()` method.

**Arguments**

- **pc**
  An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in *Using Caché ObjectScript*.

- **array-node**
  A local, process-private global, or global array node. You can specify a single array node, or a comma-separated list of array nodes. For further details on subscripts and nodes, refer to Global Structure in *Using Caché Globals*.

Attempting to use **ZKILL** on a structured system variable (SSVN) (such as `^$GLOBAL`) results in a `<COMMAND>` error.

**Example**

In this example, the **ZKILL** command deletes node `a(1)`, but does not remove node `a(1,1)`.

```
SET a(1)=1,a(1,1)=11
SET x=a(1)
SET y=a(1,1)
ZKILL a(1)
SET z=a(1,1)
WRITE "x=",x," y=",y," z=",z
```

returns `x=1 y=11 z=11`. However, then issuing `a`:

```
WRITE a(1)
```

generates an `<UNDEFINED>` error.
See Also

- KILL command
ZNSPACE

Sets the current namespace.

ZNSPACE:pc nspace
ZN:pc nspace

Arguments

pc  Optional — A postconditional expression.
nspace  A string expression that evaluates to the name of an existing namespace.

Description

ZNSPACE nspace changes the current namespace to the nspace value. nspace can be an explicit namespace name or an implied namespace.

If nspace does not exist, the system generates a <NAMESPACE> error. If you do not have access privileges to a namespace, the system generates a <PROTECT> error, followed by the database path. For example, the %Developer role does not have access privileges to the %SYS namespace. If you have this role and attempt to access this namespace, Caché issues the following error (on a Windows system): <PROTECT> *c:\intersystems\cache\mgr\.

You can change the current namespace by using the ZNSPACE command, the %CD utility (DO ^%CD), or by setting the $NAMESPACE or $ZNSPACE special variables. Use of ZNSPACE or %CD is preferable, because these provide more extensive error checking.

When you wish to temporarily change the current namespace, perform some operation, then revert to the prior namespace, use SET $NAMESPACE. Because $NAMESPACE permits you to NEW $NAMESPACE, it reverts to the original namespace when either the subroutine completes or an unexpected error occurs. See $NAMESPACE special variable for details.

The following methods may assist you when using ZNSPACE:

• To return the name of the current namespace: return the $NAMESPACE or $ZNSPACE special variable value, or invoke the NameSpace() method of the %SYSTEM.SYS class, as follows:

  WRITE $SYSTEM.SYS.NameSpace()

• To list all namespaces (explicit and implicit) available to the current process: invoke the ListAll() method of the %SYS.Namespace class, as follows:

  DO ##class(%SYS.Namespace).ListAll(.result)
  ZWRITE result

When ListAll() lists an implied namespace, it delimits the system name using caret (^) delimiters.

To list all local and (optionally) remotely-mapped namespaces, invoke the List query of the %SYS.Namespace class, as follows:

  SET ListRemote=1
  SET stmt=##class(%SQL.Statement).%New()
  SET status=stmt.%PrepareClassQuery("%SYS.Namespace","List")
  IF status'1 {WRITE "$Prepare failed:" DO $System.Status.DisplayError(status) QUIT}
  SET rset= stmt.%Execute(ListRemote)
  DO rset.%Display()}

This query returns each namespace name, its status (available or not), and whether it is mapped to a remote system.

Note that both of these listings list all namespaces, including those for which the user does not have access privileges.
To test whether a namespace is defined: use the **Exists()** method of %SYS.Namespace class, as follows:

```objectscript
WRITE ##class(%SYS.Namespace).Exists("USER") ; an existing namespace
WRITE ##class(%SYS.Namespace).Exists("LOSER") ; a non-existent namespace
```

These methods are described in the *InterSystems Class Reference*.

For UNIX® systems, the system-wide default namespace is established as a System Configuration option. For Windows systems, it is set using a command line start-up option.

For namespace naming conventions and namespace name translation, see Namespaces in the “Syntax Rules” chapter of *Using Caché ObjectScript*. For information on using namespaces, see Namespaces and Databases in the Caché Programming Orientation Guide. For information on creating and modifying namespaces, see Configuring Namespaces in the Caché System Administration Guide.

### Changing the Current Namespace

You can change the current namespace by using the **ZNSPACE** command, the %CD utility (DO ^%CD), or by setting the $NAMESPACE or $ZNSPACE special variables. From the Terminal prompt use of ZNSPACE or %CD is preferable, because these provide more extensive error checking.

When you wish to temporarily change the current namespace, perform some operation, then revert to the prior namespace, use NEW $NAMESPACE then SET $NAMESPACE. By using NEW $NAMESPACE and SET $NAMESPACE you establish a namespace context that automatically reverts to the prior namespace when the routine concludes or an unexpected error occurs.

#### Implied Namespace

An implied namespace specifies the namespace by system name and directory path. There are three forms:

- "^^." for the current namespace. This can be used to change the namespace prompt from an explicit namespace to the corresponding implied namespace.
- "^^dir" specifying the namespace directory path dir on the current system.
- "^system^dir" specifying the namespace directory path dir on a specified remote system.

For dir, specify a directory path. This is shown in the following examples:

**Windows example:**

```
ZNSPACE "^^c:\InterSystems\Cache\mgr\user"
WRITE $NAMESPACE
```

**Linux example:**

```
ZNSPACE "^RemoteLinuxSystem^/usr/Cache/mgr/user/"
WRITE $NAMESPACE
```

To return the full pathname of the current namespace, you can invoke the **NormalizeDirectory()** method, as shown in the following example:

```
WRITE ##class(%Library.File).NormalizeDirectory(""")
```

### Arguments

**pc**

**Optional** — An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in *Using Caché ObjectScript*.  

---

ZNSPACE


**namespace**

Any valid string expression that evaluates to the name of the new namespace. *namespace* can be an explicit namespace name or an implied namespace.

Namespace names are not case-sensitive. Caché always displays explicit namespace names in all uppercase letters, and implied namespace names in all lowercase letters.

**Examples**

The following example uses **NEW $NAMESPACE** to stack the current namespace. It then uses **ZNSPACE** to change the namespace. The **QUIT** reverts to the stacked namespace:

```objectscript
WRITE "before: ", $NAMESPACE, !
DO Test
WRITE "after: ", $NAMESPACE, !
QUIT
Test
NEW $NAMESPACE
ZNSPACE "%SYS"
WRITE "testing: ", $NAMESPACE, !
QUIT
```

The following example assumes that a namespace called "accounting" already exists. Otherwise, you receive a <NAMESPACE> error.

From the programmer prompt:

```
USER>ZNSPACE "Accounting"
ACCOUNTING>
```

By default, as shown in this example, the Terminal prompt displays the current namespace name. Namespace names are always displayed in uppercase letters.

The following example tests for the existence of a namespace, then uses ZNSPACE to set the current namespace and uses the **TerminalPrompt()** method to set the Terminal prompt either to the specified namespace or to USER:

```objectscript
WRITE !,"Current namespace is ", $NAMESPACE
SET ns="ACCOUNTING"
IF 1==##class(%SYS.Namespace).Exists(ns) {
  WRITE !,"Changing namespace to: ", ns
  ZNSPACE ns
  DO ##class(%SYSTEM.Process).TerminalPrompt(2)
  WRITE !,"and ", $NAMESPACE, " will display at the prompt"
}
ELSE {
  WRITE !,"Namespace ", ns, " does not exist"
  SET ns="USER"
  WRITE !,"Changing namespace to: ", ns
  ZNSPACE ns
  DO ##class(%SYSTEM.Process).TerminalPrompt(2)
  WRITE !,"and ", $NAMESPACE, " will display at the prompt"
}
```

**Notes**

**Namespaces with Default Directories**

If the namespace you select has a default directory on a remote machine, **ZNSPACE** does not change the current directory of your process to that namespace’s directory. Thus, your current namespace becomes the namespace you selected, but your current directory remains the directory that was current before you issued the **ZNSPACE** command.

**Implied Namespace Mapping**

**ZNSPACE** creates additional default mappings from an implied namespace. These mappings are the same as for a normal (explicit) namespace. They allow a process to find and execute the % routines and % globals that are physically located in the CACHESYS and CACHELIB databases (the cache\mgr\ and cache\mgr\cachelib directories).
Setting the $NAMESPACE or $ZNSPACE special variable or running the %CD routine with an implied namespace is the same as issuing a ZNSPACE command.

% Routine Mapping

When a process switches namespaces using the ZNSPACE command, the system routines path mapping is normally reset. This is true for both a normal (explicit) namespace and an implied namespace. The only exception to this is when the process switches from an implied namespace to an implied namespace, in which case the existing mapping is preserved. For further information on implied namespaces, see Global Structure in Using Caché Globals.

You can override this remapping of system routines by using the SysRoutinePath() method of the %SYSTEM.Process class. This can be used to override an existing system routine. Commonly, this is used to create an additional mapping when debugging a % routine. The process must have Write permission for the CACHESYS database. This method should be used with extreme caution.

CAUTION: Changing the mapping of a system routine supplied by InterSystems is strongly discouraged. Doing so could break current or future library routines and methods supplied by InterSystems.

% Global Mapping

The first time a user uses ZNSPACE (or its equivalent) to go to an implied namespace, the system creates a mapping for that implied namespace, as follows: Caché first maps to existing % globals in that implied namespace. Caché then maps all other % globals to CACHESYS.

Once this mapping has been created for an implied namespace, the mapping is stored in shared memory. This means that when any subsequent user goes to that implied namespace, Caché uses this pre-existing global mapping.

To update an implied namespace global mapping you must clear this shared memory storage. A system restart is one way to clear shared memory.

The deprecated $ZUTIL(5) function does not create additional default mappings from an implied namespace. Thus a process cannot find and execute % routines and % globals that are physically located in the CACHELIB database. For this reason, ZNSPACE or $NAMESPACE are strongly preferred; existing uses of $ZUTIL(5) should be phased out wherever possible.

Controlling Namespace Display

Terminal Prompt

By default, the Terminal prompt displays the current namespace name. This default is configurable:

Go to the Management Portal, select [System] > [Configuration] > [Startup Settings]. View and edit the current setting of TerminalPrompt. This also sets the prompt for Telnet windows.

To set this behavior for the current process, use the TerminalPrompt() method of the %SYSTEM.Process class. The system-wide default behavior can be established by setting the TerminalPrompt property of the Config.Startup class.

The Terminal prompt can represent the current namespace as the explicit namespace name or the implied namespace. If the implied namespace path is longer than 27 characters, the prompt is truncated to display an ellipsis, followed by the last 24 characters of the implied namespace path. For example: ...rsystems\cache\mgr\user>

$NAME and $QUERY Functions

The $NAME and $QUERY functions can return the extended global reference form of a global variable, which includes the namespace name. You can control whether these functions return namespace names as part of the global variable name. You can set this extended global reference switch for the current process using the RefInKind() method of the %SYSTEM.Process class. The system-wide default behavior can be established by setting the RefInKind property of the Config.Miscellaneous class. For further information on extended global references, see Global Structure in Using Caché Globals.
Changing Namespaces within Application Code

Object and SQL code assumes that it is running in a single namespace; hence, changing namespaces with open object instances or SQL cursors can lead to code running incorrectly. Typically, there is no need to explicitly change namespaces, as the various Object, SQL, and CSP servers automatically ensure that application code is run in the correct namespace.

Also, changing namespaces demands a relatively high amount of computing power compared to other commands; if possible, application code should avoid it.

See Also

- JOB command
- $NAMESPACE special variable
- $ZNSPACE special variable
- Configuring Namespaces in *Caché System Administration Guide*
ZTRAP

Forces an error with a specified error code.

ZTRAP:pc ztraparg
ZTRAP:pc $ZERROR
ZTRAP:pc $ZE

Arguments

<table>
<thead>
<tr>
<th>pc</th>
<th>Optional — A postconditional expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ztraparg</td>
<td>Optional — An error code string. An error code string is specified as a string literal or an expression that evaluates to a string; only the first four characters of the string are used.</td>
</tr>
<tr>
<td>$ZERROR</td>
<td>The special variable $ZERROR, which can be abbreviated $ZE.</td>
</tr>
</tbody>
</table>

Description

The ZTRAP command accepts both a command postconditional and argument indirection. ZTRAP has three forms:

- Without an argument
- With a string argument
- With $ZERROR

ZTRAP without an argument forces an error with the error code <ZTRAP>.

ZTRAP ztraparg forces an error with the error code <Zxxxx>, where xxxx is the first four characters of the string specified by ztraparg. If you specify an expression, rather than a quoted string literal, the compiler evaluates the expression and uses the first four characters of the resulting string. When evaluating an expression, Caché strips the plus sign and leading and trailing zeros from numbers. All remaining characters of ztraparg are ignored.

ZTRAP $ZERROR does not force a new error. It stops execution at the current program stack level and pops stack levels until another error handler is found. Execution then continues in that error handler with the current error code.

Arguments

pc

An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

ztraparg

A string literal or an expression that evaluates to a string. Any of the following values can be specified for ztraparg:

- A quoted string of any length containing any characters. ZTRAP uses only the first four characters to generate an error code; if there are fewer than four characters, it uses the characters provided. Unlike system error codes, which are always uppercase, case is preserved. Thus:
The **ZTRAP** command accepts argument indirection. For more information, refer to **Indirection** in *Using Caché ObjectScript*.

### Passing Control to an Error Handler with $ZERROR

When the **ZTRAP** argument is the special variable **$ZERROR**, special processing is performed which is useful in **$ZTRAP** error handlers. **ZTRAP $ZERROR** does not force a new error. It stops execution at the current program stack level and pops stack levels until another error handler is found. Execution then continues in that error handler with the current error code. This error handler may be located in a different namespace.

This way of passing control to an error handler is preferable to use of the legacy **ZQUIT** command.

### Examples

This example shows how you use the **ZTRAP** command with an expression to produce an error code:

```ObjectScript
; at this point the routine discovers an error ...
ZTRAP "ER23"
...
```

When the routine is run and it discovers the anticipated error condition, the output appears as follows:

```ObjectScript
<ZER23>label+offset^routine
```

This example shows how the use of a postconditional affects the **ZTRAP** command:

```ObjectScript
; ZTRAP:y<0 "yNEG"
;
```

When the routine is run and `y` is negative, the output is:

```ObjectScript
<ZyNEG>label+offset^routine
```

This example shows how you use argument indirection in the **ZTRAP** command:

```ObjectScript
; SET ERPTR="ERMSG"
SET ERMSG="WXYZ"
;
ZTRAP @ERPTR
```

The output is:

```ObjectScript
<ZWXY>label+offset^routine
```

The following example shows a **ZTRAP** command that invokes a **$ZTRAP** error trap handler defined at a previous context level.
Main
NEW $ESTACK
SET $ZTRAP="OnErr"
WRITE !,"$ZTRAP set to: ",$ZTRAP
WRITE !,"Main $ESTACK= ",$ESTACK 
WRITE !,"Main $ECODE= ",$ECODE," $ZERROR=",$ZERROR
DO SubA
WRITE !,"Returned from SubA" 
WRITE !,"MainReturn $ECODE= ",$ECODE," $ZERROR=",$ZERROR
QUIT
SubA
WRITE !,"SubA $ESTACK= ",$ESTACK 
ZTRAP
WRITE !,"SubA $ECODE= ",$ECODE," $ZERROR=",$ZERROR
QUIT
OnErr
WRITE !,"OnErr $ESTACK= ",$ESTACK 
WRITE !,"OnErr $ECODE= ",$ECODE," $ZERROR=",$ZERROR
QUIT

See Also

- $ZERROR special variable
- $ZTRAP special variable
- Error Handling in Using Caché ObjectScript
- Labels in Using Caché ObjectScript
ZWRITE

Displays variable names and their values and/or expression values.

ZWRITE:pc expression,...
ZW:pc expression,...

Arguments

<table>
<thead>
<tr>
<th>pc</th>
<th>Optional — A postconditional expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>Optional — A variable or expression to display, or a comma-separated list of variables and/or expressions to display. A comma-separated list can contain any combination of variables and expressions.</td>
</tr>
</tbody>
</table>

Description

The ZWRITE command lists names of variables and their values. It lists these variables and their descendents in the format varname=value in canonical order, one variable per line, on the current device. ZWRITE also lists the values of expressions. Expressions are listed as value, one per line, in the order specified. The ZWRITE command has two forms:

- Without an argument
- With arguments

ZWRITE can take an optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

ZWRITE listing can be interrupted by issuing a CTRL-C, generating an <INTERRUPT> error.

ZWRITE without an Argument

ZWRITE without an expression argument is functionally identical to WRITE without an argument. It displays the names and values of all variables in the local variable environment (local variables), including private variables. It does not display process-private globals or special variables. It lists variables by name in ASCII order. It lists subscripted variables in subscript tree order.

ZWRITE without an argument displays an OREF value assigned to a local variable as variable=<OBJECT REFERENCE>[oref]. It does not display any further details about the OREF. For information on OREFs, see “OREF Basics” in Using Caché Objects.

ZWRITE without an argument displays a bitstring assigned to a local variable as a compressed character string, which (because it contains non-printing characters) may appear to be an empty string. It does not display any further details about the bitstring.

For further details, refer to the WRITE command.

ZWRITE with Arguments

ZWRITE with an argument can specify one expression argument or a comma-separated list of expression arguments. These arguments are evaluated in left-to-right order. Each argument can specify a variable or an expression. If expression is a comma-separated list, each variable or expression is displayed on a separate line.

- Variables: displayed as varname=value
- Expressions: evaluated and the results displayed as value
• **Special Variables**: displayed as value

• **Caché List Structures**: displayed as $lb(element1,element2)

• **Subscripted Globals**, including those used to store SQL data values and SQL index values, displayed as $lb() List structures

• **Object References**: displayed as <OBJECT REFERENCE>[oref]. This is the value displayed by ZWRITE without an argument; ZWRITE with an object reference argument displays this value plus additional “general information”, “attribute values”, and (where appropriate) “swizzled references” and “calculated references” information.

• **JSON Arrays and JSON Objects**: displayed as Object References.

• **Bit Strings**: displayed as both the $ZWCHAR compressed binary string, and a user-readable representation of all of the “1” bits in the bitstring.

**ZWRITE** displays a string containing one or more non-numeric characters as a quoted string.

**ZWRITE** displays a numeric value as a **canonical number**. **ZWRITE** displays a numeric string containing a number in canonical form as an unquoted canonical number. **ZWRITE** displays a numeric string not in canonical form as a quoted string. Note however that any arithmetic operation on a non-canonical numeric string converts it to a canonical number. This is shown in the following example:

```
SET numcanon=7.9     // returns number
SET num=+007.90      // returns number
SET strnum="+7.9"    // returns string
SET strcanon="7.9"   // returns number
SET strnumop="+7.90" // returns number
ZWRITE numcanon,num,strnum,strcanon,strnumop
```

**ZWRITE** truncates the display of very long strings and appends . . . to indicate that the string display was truncated.

**ZWRITE** displays values containing control characters (including those created with $LISTBUILD and $BIT) in a readable format. If this formatting causes very long string values to exceed the **maximum string length**, **ZWRITE** truncates the displayed string and appends . . . to indicate that the string was truncated.

For a comparison of **ZWRITE** with the **WRITE**, ZZDUMP, and ZZWRITE commands, refer to the Display (Write) **Commands** features tables in the “Commands” chapter of *Using Caché ObjectScript*.

**Variables**

If **expression** is a variable, **ZWRITE** writes **varname=value** on a separate line. The variable can be a **local variable**, **process-private global**, **global variable**, or **object reference (OREF)**.

**ZWRITE** ignores undefined variables. It does not issue an error. If you specify one or more undefined variables in a comma-separated list of variables, **ZWRITE** ignores the undefined variables and returns the defined variables. This behavior allows you to display multiple variables without checking to determine if all of them are defined. If you specify an undefined variable to **WRITE**, ZZDUMP, or ZZWRITE Caché issues an **UNDEFINED** error.

Variables can be subscripted. If the variable has defined subnodes, **ZWRITE** writes a separate **varname=value** line for each subnode in subscript tree order. When you specify a root node, **ZWRITE** displays all of its subnodes, even when the root node is undefined.

You can use extended global reference to specify a global variable not mapped to the current namespace. **ZWRITE** displays extended global references even when the **RefInKind()** method of the %SYSTEM.Process class or the **RefInKind** property of the Config.Miscellaneous class has been set to strip extended global references. If you specify a nonexistent namespace, Caché issues a <**NAMESPACE**> error. If you specify a namespace for which you do not have privileges, Caché issues a <**PROTECT**> error, followed by the global name and database path, such as the following: <**PROTECT**> ^myglobal,c:\intersystems\cache\mgr\. For further information on subscripted variables and extended global reference, refer to **Global Structure** in *Using Caché Globals*. 
Non-Display Characters

`ZWRITE` displays all printable characters. It displays non-printable characters using the `$CHAR` function, representing each non-printable character as a concatenated `$c(n)` value. It does not execute non-printing control characters. This is shown in the following example:

```objectscript
SET charstr=$CHAR(65,7,66,67,0,68,11,49,50)
ZWRITE charstr
```

Expressions

If `expression` is a literal expression, `ZWRITE` evaluates the expression and writes the resulting `value` on a separate line. If the expression contains an undefined variable, Caché issues an `<UNDEFINED>` error.

If the expression is a multidimensional property, `ZWRITE` will not display the property descendants. To display an entire multidimensional property with `ZWRITE`, either `MERGE` it into a local array and display the array, or display the entire object.

Caché List Structures

You can specify a Caché list structure (%List) to `ZWRITE` as a variable or an expression. `ZWRITE` displays a list structure as `$lb(element1,element2)`. This is shown in the following example:

```objectscript
SET FullList = $LISTBUILD("Red","Blue","Green","Yellow")
SET SubList = $LIST(FullList,2,4)
SET StrList = $LISTFROMSTRING("Crimson^Azure^Lime","^")
ZWRITE FullList,SubList,StrList
```

A Subscripted Global and its Descendants

The following example shows `ZWRITE` displaying the contents of a subscripted global variable and all of its descendent nodes. This example displays the data defined for the Sample.Person persistent class that projects to an SQL table. The global variable takes its name from the persistent class name (not the SQL table name) and is case-sensitive; a “D” is appended to indicate a data global. Note that the descendent nodes contain list structures, which are displayed as `$LISTBUILD` ($lb) constructions:

```objectscript
ZWRITE ^Sample.PersonD
```

To display a single data record, you can specify the `RowID` value as the global subscript, as shown in the following:

```objectscript
ZWRITE ^Sample.PersonD(22)
```

To display the contents of an index, you can specify the persistent class name with an “I” appended, supplying the index name as the subscript. The index name is case-sensitive:

```objectscript
ZWRITE ^Sample.PersonI("NameIDX")
```

Additional non-printing characters used in lists are also displayed.

The following example shows `ZWRITE` using extended global reference to display the contents of a subscripted global variable located in a specified namespace:

```objectscript
ZWRITE ^["USER"]Sample.PersonD
```

The namespace name can be a different namespace or the current namespace. Namespace names are not case-sensitive. `ZWRITE` always displays the extended global reference, regardless of the setting of the `RefInKind` method or property, which can be set to strip extended global references from globals returned by `$QUERY` or `$NAME`. 
**Object References**

You can specify an object reference (OREF) to `ZWRITE` as either a variable or an expression. If you have specified an object reference, `ZWRITE` displays `variable=<OBJECT REFERENCE>[oref]` or `<OBJECT REFERENCE>[oref]` and also displays General Information, Attribute Values, and (when appropriate) Swizzled References and Calculated References for the properties of the object, one attribute per line.

**Note:** The `<OBJECT REFERENCE>` identifier prefix may not be displayed when executing `ZWRITE` through a browser interface, because browsers interpret angle brackets as tags.

If the `ZWRITE` argument is an embedded object property, `ZWRITE` displays General Information and Attribute Values for the array elements of the container property, one attribute per line. The display format is the same as the `%SYSTEM.OBJ.Dump()` method.

The following example displays the OREF, followed by “general information”, “attribute values” and “swizzled references”:

```ObjectScript
SET oref = ##class(%SQL.Statement).%New()
ZWRITE oref
```

The following example displays the OREF, followed by “general information”, “attribute values”, and “calculated references”:

```ObjectScript
SET doref=##class(%iKnow.Domain).%New("mytempdomain")
DO doref.%Save()
SET domId=doref.Id
ZWRITE doref
SET stat=##class(%iKnow.Domain).%DeleteId(domId)
```

The following examples displays the OREF, followed by “general information”, “attribute values”, “swizzled references”, and “calculated references”:

```ObjectScript
NEW $NAMESPACE
SET $NAMESPACE="Samples"
SET poref=##class(Sample.Person).%OpenId(1)
ZWRITE poref

NEW $NAMESPACE
SET $NAMESPACE="Samples"
SET myquery = "SELECT TOP 2 Name,DOB FROM Sample.Person"
SET oref = ##class(%SQL.Statement).%New()
SET qStatus = oref.%Prepare(myquery)
    IF qStatus'1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = oref.%Execute()
ZWRITE rset
```

For information on OREFs, see “OREF Basics” in *Using Caché Objects*.

**JSON Arrays and JSON Objects**

`ZWRITE` displays JSON dynamic arrays and JSON dynamic objects as object references. The referenced object classes are `%Library.DynamicArray` and `%Library.DynamicObject`. This is shown in the following example:

A JSON array:

```ObjectScript
SET jarray = ["apples","oranges"]
ZWRITE jarray
```

A JSON object:

```ObjectScript
SET job = {"fruit":"apples","count":24}
ZWRITE job
```

For information on object references (OREFs), see “OREF Basics” in *Using Caché Objects*. 
**Bitstrings**

You can specify a bitstring to **ZWRITE** as either a variable or an expression. If the **ZWRITE** argument is a Caché compressed bitstring (created using the $BIT function), **ZWRITE** displays the decimal representation of the compressed binary string as $ZWCHAR ($zwc) two-byte (wide) characters.

**ZWRITE** also displays a comment that lists the uncompressed “1” bits in left-to-right order as a comma-separated list. If there are three or more consecutive “1” bits, it lists them as a range (inclusive) with two dot syntax (n..m). For example, the bitstring [1,0,1,1,1,0,1] is shown as /*$bit(1,3..6,8)*/. The bitstring [1,1,1,1,1,1,1] is shown as /*$bit(1..8)*/. The bitstring [0,0,0,0,0,0,0] is shown as /*$bit()*/. The following example shows **ZWRITE** bitstring output:

```objectscript
SET $BIT(a,1) = 0
SET $BIT(a,2) = 0
SET $BIT(a,3) = 1
SET $BIT(a,4) = 0
SET $BIT(a,5) = 1
SET $BIT(a,6) = 1
SET $BIT(a,7) = 1
SET $BIT(a,8) = 0
ZWRITE a
```

**Examples**

**ZWRITE Without an Argument**

In following example, **ZWRITE** without an argument lists all defined local variables in ASCII name order.

```objectscript
SET A="A",a="a",AA="AA",aA="aA",aa="aa",B="B",b="b"
ZWRITE
```

returns:

```
A="A"
AA="AA"
B="B"
a="a"
aA="aA"
aa="aa"
b="b"
```

In the following example **ZWRITE** without an argument lists canonical and non-canonical numeric values:

```objectscript
SET w=10
SET x=+0012.00
SET y="6.5"
SET z="007"
SET a=w+x+y+z
ZWRITE
```

returns:

```
a=35.5
w=10
x=12
y=6.5
z="007"
```

**ZWRITE with Arguments**

In the following example, **ZWRITE** displays three variables as varname=value, each on its own line:

```objectscript
SET alpha="abc"
SET x=100
SET y=80
SET sum=x+y
ZWRITE x,sum,alpha
```
In the following example, **ZWRITE** evaluates an expression in the first argument. It returns the expression as *value*, and the variable as *varname=value*:

```
SET x=100
SET y=80
ZWRITE x+y,y
```

The following example compares **ZWRITE** and **WRITE** when displaying different variable values. **ZWRITE** returns quotation marks delimiting strings, **WRITE** does not:

```
SET a=+007.00
SET b=9E3
SET c="+007.00"
SET d=""
SET e="Rhode Island"
SET f="Rhode.."Island"
ZWRITE a,b,c,d,e,f
WRITE !,a,!,b,!,c,!,d,!,e,!,f
```

**ZWRITE Displaying Subscript Subnodes**

The following example shows **ZWRITE** displaying the contents of subscripted process-private global variables. **ZWRITE** displays the subscripts of the variable in hierarchical order:

```
SET ^||fruit(1)="apple",^||fruit(4)="banana",^||fruit(8)="cherry"
SET ^||fruit(1,1)="Macintosh",^||fruit(1,2)="Delicious",^||fruit(1,3)="Granny Smith"
SET ^||fruit(1,2,1)="Red Delicious",^||fruit(1,2,2)="Golden Delicious"
SET ^||fruit="Fruits"
WRITE "global arg ZWRITE: ",!
ZWRITE ^||fruit
```

Note that specifying a root node displays all subnodes, even when the root node itself is undefined:

```
SET fruit(1)="apple",fruit(4)="bananas",fruit(8)="cherry"
SET fruit(1,1)="Macintosh",fruit(1,2)="Delicious",fruit(1,3)="Granny Smith"
SET fruit(1,2,1)="Red Delicious",fruit(1,2,2)="Golden Delicious"
WRITE "global arg ZWRITE: ",!
ZWRITE fruit
```

**See Also**

- **WRITE** command
- **ZZDUMP** command
- **ZZWRITE** command
- **Display (Write) Commands** in the “Commands” chapter of *Using Caché ObjectScript*
ZZDUMP

Displays an expression in hexadecimal dump format.

| Arguments |
| --- | --- |
| pc | Optional — A postconditional expression. |
| expression | The data to be displayed in hexadecimal dump format. You can specify a number, a string (enclosed in quotation marks), or a variable that resolves to one of these. You can specify a single expression, or a comma-separated list of expressions. |

Description

ZZDUMP displays an expression in hexadecimal dump format. ZZDUMP is primarily of interest to system programmers, but it can be useful in viewing strings that contain control characters.

ZZDUMP returns a number or string value in the following format:

| position: hexdata printdata |

Arguments

pc

Caché executes the ZZDUMP command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

expression

You can specify expression as a numeric, a string literal, or a variable that resolves to one of these. You can specify a single expression, or a comma-separated list of expressions. Specifying a comma-separated list of expressions is parsed as issuing a separate ZZDUMP command for each expression. Execution of a comma-separated list stops when the first error occurs.

An expression can be a variable of any type, including local variables, process-private globals, global variables, and special variables. You can use extended reference to specify a global variable in another namespace. If you specify a nonexistent namespace, Caché issues a <NAMESPACE> error. If you specify a namespace for which you do not have privileges, Caché issues a <PROTECT> error, followed by the global name and database path, such as the following: <PROTECT> ^myglobal,c:\intersystems\cache\mgr\.

Non-printing characters are represented in hexdata by their hexadecimal value, and in printdata by a placeholder dot (.). Control characters are not executed.

Examples

The following example shows ZZDUMP returning hex dumps for two single-character string variables. Note that each comma-separated expression is treated as a separate invocation of ZZDUMP:

```
SET x="A"
SET y="B"
ZZDUMP x,y
```

| 0000: | 41 | A |
| 0000: | 42 | B |
The following example shows `ZZDUMP` returning a hex dump for a string variable too long for a single dump line. Note that the `position` for the second dump line (0010:) is in hexadecimal:

```objectscript
SET z="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
ZZDUMP z
```

```
0000: 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50     ABCDEFGHIJKLMNOP
0010: 51 52 53 54 55 56 57 58 59 5A                       QRSTUVWXYZ
```

The following example shows `ZZDUMP` returning hex dumps for three variables. Note that no hex dump (not even a blank line) is returned for a null string variable. Also note that a number is converted to canonical form (leading and trailing zeros and plus sign removed); a string containing a number is not converted to canonical form:

```objectscript
SET x=+007
SET y=""
SET z="+007"
ZZDUMP x,y,z
```

```
0000: 37                                         7
0000: 2B 30 30 37                               +007
```

### Notes

#### Unicode

If one or more characters in a `ZZDUMP` expression is a wide (Unicode) character, all characters in that expression are represented as wide characters. The following examples show variables containing a Unicode characters. In all cases, all characters are displayed as wide characters.

```objectscript
SET x=$CHAR(987)
SET y=$CHAR(987)_"ABC"
ZZDUMP x,y
```

```
0000: 03DB                                                    ?
0000: 03DB 0041 0042 0043                                     ?ABC
```

#### ZZDUMP Compared with Write Commands

For tables comparing `ZZDUMP` with the `WRITE`, `ZWRITE`, and `ZZWRITE` commands, refer to the Display (Write) Commands in the “Commands” chapter of *Using Caché ObjectScript*.

### See Also

- **WRITE** command
- **ZWRITE** command
- **ZZWRITE** command
- **$CHAR** function
- **$DOUBLE** function
- **$LISTBUILD** function
- **$ZHEX** function
- Display (Write) Commands in the “Commands” chapter of *Using Caché ObjectScript*
 ZZWRITE

Displays the values of variables or expressions.

ZZWRITE:pc expression,...

Arguments

<table>
<thead>
<tr>
<th>pc</th>
<th>Optional — A postconditional expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A variable or expression to display, or a comma-separated list of variables and/or expressions to display. A comma-separated list can contain any combination of variables and expressions.</td>
</tr>
</tbody>
</table>

Description

The ZZWRITE command evaluates an expression and displays a value on the current device. This expression can be a literal, local variable, process-private global, global variable, or special variable. ZZWRITE can evaluate a comma-separated list of expressions; it displays the results in the order specified, one expression per line. ZZWRITE displays the result of each expression as %val=value.

ZZWRITE without an argument is a no-op. It performs no operation and issues no error.

ZZWRITE and ZWRITE

ZZWRITE, like ZWRITE, displays non-printing characters and encoded data such as Caché lists, bitstrings, and %Status strings in a human-readable format. It does not execute control characters. Both commands provide an extensive display of object reference (oref) values, consisting of the oref value followed by the same “general information”, “attribute values”, and (where appropriate) “swizzled references” and “calculated references” returned by the %SYSTEM.OBJ.Dump() method.

ZZWRITE displays the same data values as ZWRITE with an argument, with the following differences:

• Variable Names: ZZWRITE displays the value of every expression or variable as %val=value. ZWRITE displays local, process-private, and global variables as varname=value, and literals, expressions, and special variables as value.

• Undefined Variables: ZZWRITE issues an <UNDEFINED> error for an undefined variable. ZWRITE ignores undefined variables.

• Subscripts: ZZWRITE displays the value of the specified subscript node. ZWRITE displays the subscript node and all defined subnodes in subscript tree order.

• Extended Global Reference: ZZWRITE displays the value of an extended global reference as %val=value (like any other expression), giving no indication that the value is defined in another namespace. ZWRITE displays an extended global reference variable name showing the namespace that contains the global.

For further details on how various data values are displayed, refer to ZWRITE.

For tables comparing ZZWRITE with the WRITE, ZWRITE, and ZZDUMP commands, refer to the Display (Write) Commands features tables in the “Commands” chapter of Using Caché ObjectScript.
Arguments

pc

An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

expression

An expression to evaluate, or a comma-separated list of expressions. An expression can consist of, or contain local variables, process-private globals, global variables, or special variables. It cannot be a private variable. Variables can be subscripted. Expressions are evaluated in strict left-to-right order.

You can use extended global reference to specify a global variable not mapped to the current namespace. If you specify a nonexistent namespace, Caché issues a <NAMESPACE> error. If you specify a namespace for which you do not have privileges, Caché issues a <PROTECT> error, followed by the global name and database path, such as the following: <PROTECT> ^myglobal,c:\intersystems\cache\mgr\. For further information on subscripted variables and extended global reference, refer to Global Structure in Using Caché Globals.

See Also

• WRITE command
• ZWRITE command
• ZZDUMP command
• Display (Write) Commands in the “Commands” chapter of Using Caché ObjectScript
The following are reference pages for routine and debugging commands supported by ObjectScript. These commands supplement the ObjectScript general commands described earlier in this document. Commands are listed in alphabetical order.

Further introductory information on ObjectScript command syntax and conventions is provided at the beginning of the general commands reference pages. Additional information on ObjectScript commands can be found in the Commands chapter of *Using ObjectScript*. 
PRINT

Displays lines of code from the current routine on the current device.

<table>
<thead>
<tr>
<th>PRINT:pc lineref1:lineref2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P:pc lineref1:lineref2</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>Optional — A postconditional expression.</td>
</tr>
<tr>
<td>lineref1</td>
<td>Optional — The line to be displayed, or the first line in a range of lines to be displayed. Can be a label name, a numeric offset (+n) or a label name and a numeric offset. If omitted, the entire current routine is displayed.</td>
</tr>
<tr>
<td>:lineref2</td>
<td>Optional — The last line in a range of lines to be displayed. To define a range, lineref1 must be specified.</td>
</tr>
</tbody>
</table>

**Description**

The `PRINT` command displays lines of code from the currently loaded routine. The output is sent to the current device. You establish the current device with the `USE` command. For the name of the current routine, access the `$ZNAME` special variable. For the device ID of the current device, access the `$IO` special variable.

**Note:** The `PRINT` and `ZPRINT` commands are functionally identical.

The displayed lines include all labels, comments, and whitespace, with the exception that entirely blanks lines in a MAC routine, which are removed by the compiler, are neither displayed nor counted. For this reason, `PRINT` displays and counts the following multi-line comment as two lines, not three:

```plaintext
/* This comment includes
   a blank line */
```

`PRINT` has two forms:

- Without arguments
- With arguments

`PRINT` without arguments displays all the lines of code in the currently loaded routine.

`PRINT` with arguments displays the specified lines of code. `PRINT lineref1` displays the line specified by `lineref1`. `PRINT lineref1:lineref2` displays the range of lines starting with `lineref1` and ending with `lineref2` (inclusive).

You can use the `$TEXT` function to return a single line of code.

**Arguments**

**pc**

An optional postconditional expression. Caché executes the `PRINT` command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to **Command Postconditional Expressions** in *Using Caché ObjectScript*. 
lineref1
The line to be printed or the first in a range of lines to be displayed or printed. Can be specified in either of the following syntactical forms:

- \(+\text{offset}\) where \text{offset} is a positive integer specifying the line number within the current routine. +1 is the first line in the routine, which may be a label line. +0 always returns the empty string.
- \text{label[+\text{offset}]\text{}} where \text{label} is a label within the routine and \text{offset} is the line number counting from the label (with the label itself counting as offset 0). If you omit the \text{offset} option, or specify \text{label+0}, Caché prints the label line. \text{label+1} prints the line after the label.

A label may be longer than 31 characters, but must be unique within the first 31 characters. \textbf{PRINT} matches only the first 31 characters of a specified \textit{label}. Label names are case-sensitive, and may contain Unicode characters.

lineref2
The last line in a range of lines to be displayed. Specify in the same way as \textit{lineref1}. \textit{lineref1} must be specified to specify \textit{lineref2}. \textit{lineref1} and \textit{lineref2} are separated by a colon (:) character. No whitespace may appear between the colon and \textit{lineref2}.

If \textit{lineref2} is earlier in the line sequence than \textit{lineref1}, \textbf{PRINT} ignores \textit{lineref2} and displays the single line of code specified by \textit{lineref1}.

Examples
Given the following lines of code:

```
AviationLetters
Abc
  WRITE "A is Abel","!
  WRITE "B is Baker","!
  WRITE "C is Charlie","!
  Def WRITE "D is Delta","!
  /* Not sure about E */
  WRITE "F is Foxtrot","!
```

\textbf{PRINT} with no \textit{lineref} arguments displays all nine lines, including the comment line.

\textbf{PRINT} +0 displays the empty string.

\textbf{PRINT} +1 displays the \texttt{AviationLetters} label.

\textbf{PRINT} +8 displays the /* Not sure about E */ comment line.

\textbf{PRINT} +10 displays the empty string.

\textbf{PRINT} \texttt{Def} or \textbf{PRINT} \texttt{Def+0} display the \texttt{Def WRITE "D is Delta","!} line. This is a label line that also includes executable code.

\textbf{PRINT} \texttt{Def+1} displays the \texttt{WRITE "E is Epsilon","!} line.

Range Examples
\textbf{PRINT} +0:+3 displays the empty string.

\textbf{PRINT} +1:+3 displays the first three lines.

\textbf{PRINT} +3:+3 displays the third line.

\textbf{PRINT} +3:+1 displays the third line; \textit{lineref2} is ignored.

\textbf{PRINT} +3:Abc+1 displays the third line. Both \textit{lineref1} and \textit{lineref2} are specifying the same line.
PRINT +3:abc+1 displays from the third line to the end of the routine. Line labels are case-sensitive, so the range endpoint was not found.

PRINT Abc+1:+4 displays lines 3 and 4.

PRINT Abc+1:Abc+2 displays lines 3 and 4.

PRINT Abc:Def displays lines 2, 3, 4, 5, and 6.

PRINT Abc+1:Def displays lines 3, 4, 5, and 6.

PRINT Def:Abc displays the Def WRITE "D is Delta",! line. Because lineref2 is earlier in the code, it is ignored.

**See Also**

- ZINSERT command
- ZLOAD command
- ZPRINT command
- ZREMOVE command
- ZSAVE command
- ZZPRINT command
- $TEXT function
- $IO special variable
- Comments in *Using Caché ObjectScript*
- Labels in *Using Caché ObjectScript*
- The Spool Device in *Caché I/O Device Guide*
ZBREAK

Sets a breakpoint or watchpoint.

ZBREAK:pc
ZB:pc
ZBREAK:pc location:action:condition:execute_code
ZB:pc location:action:condition:execute_code
ZBREAK:pc /command:option
ZB:pc /command:option

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>Optional — A postconditional expression. Specifies a code line location (that sets a breakpoint), a local variable var (which sets a watchpoint), or $ (the single step breakpoint). If the location specified already has a breakpoint/watchpoint defined, the new specification completely replaces the old one. You can optionally preface location with a sign: +, −, or −−. A location without a sign prefix sets the specified breakpoint/watchpoint. A − (minus) prefix disables the breakpoint/watchpoint. A + (plus) prefix re-enables a disabled breakpoint/watchpoint. A −− (minus-minus) prefix removes the breakpoint/watchpoint. The following signs can be specified without a location: − (minus) = disable all breakpoints and watchpoints. + (plus) = re-enable all disabled breakpoints and watchpoints.</td>
</tr>
<tr>
<td>location</td>
<td>Specifies an expression that will be evaluated to a boolean value when the breakpoint/watchpoint is triggered. The expression must be surrounded by quotation marks. If true, action is carried out. If not specified, the default is true. If omitted, and either :condition or :execute_code is specified, the colon must appear as a placeholder.</td>
</tr>
<tr>
<td>:action</td>
<td>Optional — Specifies the action to take when the breakpoint/watchpoint is triggered, specified as an alphabetic code. Action code letters may be uppercase or lowercase, but must be enclosed in quotation marks. If omitted, default is “B”. If omitted, and either :condition or :execute_code is specified, the colon must appear as a placeholder.</td>
</tr>
<tr>
<td>:condition</td>
<td>Optional — Specifies ObjectScript code to be executed if :condition is true. The code must be surrounded by quotation marks if it is a literal.</td>
</tr>
<tr>
<td>:execute_code</td>
<td>Specifies ObjectScript code to be executed if :condition is true. The code must be surrounded by quotation marks if it is a literal.</td>
</tr>
<tr>
<td>/command:option</td>
<td>A command governing all breakpoints and watchpoints. The slash (/) prefix is mandatory. Available commands are: /CLEAR, /DEBUG, /TRACE, /ERRORTRAP, /INTERRUPT, /STEP, and /NOSTEP. All except /CLEAR take an option, as described below. Most /command names can be specified as a single-letter abbreviation: /C, /T, and so forth.</td>
</tr>
</tbody>
</table>

Description

ZBREAK sets breakpoints at specific lines of code and watchpoints on specific local variables to allow you to interrupt program execution for debugging. Once established, a breakpoint or watchpoint persists for the duration of the current
process, or until explicitly removed or cleared. Breakpoints and watchpoints are persistent across namespaces. You can establish a maximum of 20 breakpoints and 20 watchpoints per process. These maximums include both enabled and disabled breakpoints and watchpoints. Attempting to set more than 20 breakpoints or watchpoints results in a <COMMAND> error.

Various ways to use the ZBREAK command are described in the Debugging chapter in Using Caché ObjectScript. The use of watchpoints in a FOR loop is described in “FOR and Watchpoints” section of the FOR command reference page.

**Required Permission**

To use ZBREAK statements when running code, the user must be assigned to a role (such as %Developer or %Manager) that provides the %Development resource with U (use) permission. A user is assigned to a role either through the SQL GRANT statement, or by using the Management Portal System Administration, Security, Users option. Select a user name to edit its definition, then select the Roles tab to assign that user to a role.

To use ZBREAK the user must have WRITE access for the database in which the code resides. Otherwise, stepping, breakpoints, and watchpoints will be disabled. For example, a user who does not have WRITE access to the %SYS (CACHESYS) or the CACHELIB database will not be able to debug routines in that database. Caché disables debugging when a routine in one of these databases is entered, and only restores debugging once the routine quits. This has the effect that any other code called by that routine will also have debugging disabled, regardless of whether the user has WRITE access for that code's database.

**Listing Breakpoints**

The argumentless ZBREAK command lists the current breakpoints and watchpoints. It lists both enabled and disabled breakpoints and watchpoints.

**Listing Local Variable Values**

You can follow any ZBREAK command with an argumentless WRITE command on the same line. (Note that an argumentless ZBREAK must be followed by two spaces.) The argumentless WRITE lists the values of all local variables at the time when the ZBREAK line is encountered; not when it is put into effect.

**Disable/Enable All Existing Breakpoints/Watchpoints**

A ZBREAK command can be followed by a sign with no location specification. A minus sign (−) argument disables all current breakpoints and watchpoints. A plus sign (+) argument re-enables all previously disabled breakpoints and watchpoints. The following ZBREAK without location commands are supported:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ZBREAK -</code></td>
<td>Disables all existing breakpoints and watchpoints.</td>
</tr>
<tr>
<td><code>ZBREAK +</code></td>
<td>Re-enables all previously disabled breakpoints and watchpoints. You cannot re-enable removed breakpoints/watchpoints.</td>
</tr>
</tbody>
</table>

**ZBREAK Help Text**

To view online help text about ZBREAK at the terminal prompt, specify a question mark, as follows:

```
USER> ZBREAK ?
```

**Arguments**

`pc`

An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.
**location**

The *location* argument is required if you wish to set or unset breakpoints or watchpoints. It can consist of a sign (plus, minus, or minus-minus) followed by a breakpoint or watchpoint specification. Various combinations of signs and breakpoint/watchpoint specifications are supported:

- *varname*: a local variable. Establishes a watchpoint each time the variable value is set. The asterisk prefix is mandatory. Setting a watchpoint for an undefined local variable does not generate an error. You can optionally specify an *action*, *condition*, and/or *execute_code* for each watchpoint. Any of these optional arguments may be omitted, but if specified or skipped, the mandatory colon separators (:) must be specified.

- $: a single step breakpoint.

- A line reference, specified as *label+offset^routine* to specify a breakpoint. You can specify any combination of *label*, *offset*, and *^routine* (in that order). If you omit *label*, the breakpoint location is counted by offset from the top of the routine (*ZBREAK* statements are counted as offset lines). If you omit *+offset*, the breakpoint location is the specified label line. If you omit *^routine*, the breakpoint location is assumed to be in the currently loaded routine; you can use *^routine* to specify a routine location other than the currently loaded routine. Setting a breakpoint for a nonexistent label, offset, or routine does not generate an error.

  To set a breakpoint, specify the *label+offset^routine* location and, optionally, the *action*, *condition*, and/or *execute_code*, as follows: *label+offset^routine*:action:condition:execute_code. Any of these optional arguments may be omitted, but if an argument is skipped, the mandatory colon separator (:) must be specified.

  ZBREAK does not move the edit pointer.

  Optionally, a location argument may be prefixed by a sign which indicates what to do with an existing breakpoint or watchpoint at the specified location. You can specify no sign (set), a minus sign (disable), a plus sign (re-enable), or two minus signs (remove). You can also specify a sign before a $ single-step breakpoint. Attempting to disable, re-enable, or remove a non-existent breakpoint or watchpoint generates a <COMMAND> error.

<table>
<thead>
<tr>
<th>Sign Prefix</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>Set breakpoint/watchpoint at <em>location</em>.</td>
</tr>
<tr>
<td>--location#delay</td>
<td>Disable breakpoint/watchpoint at <em>location</em>. The optional #delay integer specifies the number of iterations to disable this breakpoint or watchpoint before breaking. The default is to disable all encounters with the breakpoint or watchpoint. No spaces are permitted before the # symbol.</td>
</tr>
<tr>
<td>+location</td>
<td>Re-enable breakpoint/watchpoint at <em>location</em>.</td>
</tr>
<tr>
<td>--location</td>
<td>Remove breakpoint/watchpoint at <em>location</em>. To remove all breakpoints and watchpoints use ZBREAK /CLEAR.</td>
</tr>
</tbody>
</table>

A line reference *location* must occur on a command boundary. A variable that is set within a command expression cannot be used as a *location*. This type of variable setting occurs in the target parameters of the $DATA, $ORDER, and $QUERY functions.

**action**

A code that specifies the action to take when the breakpoint/watchpoint is triggered. For breakpoints, the action occurs before the line of code is executed. For watchpoints, the action occurs after the command that modifies the local variable. An *action* can only be specified when setting a breakpoint/watchpoint; not when disabling or re-enabling.
### Action Code | Description
---|---
“B” | Suspends execution and displays the line at which the break occurred, with a caret (^) indicating the location within the line. A GOTO resumes execution. “B” is the default.
“L” | Suspends execution for single-step execution of lines using GOTO. Single-step mode suspended during DO, XECUTE, or user-defined functions.
“L+” | Suspends execution for single-step execution of lines using GOTO. Single-step mode also applies to DO, XECUTE, and user-defined functions.
“S” | Suspends execution for single-step execution of commands using GOTO. Single-step mode suspended during DO, FOR, XECUTE, or user-defined functions.
“S+” | Suspends execution for single-step execution of commands using GOTO. Single-step mode also applies to DO, FOR, XECUTE, and user-defined functions.
“T” | Outputs a trace message to the trace device. Can be combined with any other action code. For example “TB” means suspend execution (“B”) and output trace message (“T”). “T” by itself does not suspend execution. This action only works if a previous ZBREAK command or the current ZBREAK specifies ZBREAK /TRACE:ON. See /TRACE below.
“N” | Take no action at this breakpoint or watchpoint.

#### condition
A boolean expression. When true (1), the action should be taken and the execute_code (if present) executed. When false (0), the action and execute_code are ignored. Default is true (1).

#### execute_code
The ObjectScript code to be executed. This code is executed before the action being carried out. Before the code is executed, the value of $TEST is saved. After the code has executed, the value of $TEST as it existed in the program being debugged is restored.

The execute_code is performed internally by an XECUTE command. An XECUTE can access only public variables. However, you can specify parameter passing in the execute_code to pass private variables to the XECUTE.

Because an XECUTE argument contains a quoted string, quotes must be doubled when the code is passed via the ZBREAK execute_code argument. The easiest way to do this correctly is to first write the code as an actual XECUTE command, then double all the quotes to create the corresponding the ZBREAK execute_code.

For example, to display the new value of the variable var when it is changed, first write an XECUTE command to display it:

```
XECUTE {"(arg) WRITE ""now var="",arg,"",$GET(var,"<UNDEFINED>"})
```

Then the equivalent ZBREAK command will be:

```
ZBREAK *var::*:"("(arg) WRITE """"now var="",arg,"",$GET(var,"<UNDEFINED>"))"
```

#### /command:option
A command keyword used to set the ZBREAK environment for subsequent ZBREAK commands. The /CLEAR command takes no option. The other command keywords are followed by an option, separated by a colon. No spaces are permitted.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/CLEAR</td>
<td>Remove all breakpoints.</td>
</tr>
<tr>
<td>Keyword</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>/DEBUG:device</td>
<td>Clear or set debug device.</td>
</tr>
<tr>
<td>/TRACE</td>
<td>Enable or disable sending trace messages to the trace device (the “T” action in subsequent ZBREAK commands.) Options are :ON=enable trace. :OFF=disable trace. :ALL=trace all lines. You can redirect output with the :ON or :ALL options by specifying a device. For example ZBREAK /TRACE:ON:device.</td>
</tr>
<tr>
<td>/ERRORTRAP</td>
<td>Enable or disable $ZTRAP, $ETRAP, and TRY / CATCH error trapping. Options are :ON and :OFF.</td>
</tr>
<tr>
<td>/INTERRUPT</td>
<td>Specify Ctrl-C action. Options are :NORMAL and :BREAK.</td>
</tr>
<tr>
<td>/STEP</td>
<td>Enable stepping through code modules. Options are :EXT (user language extensions); :METHOD (object methods); :DESTRUCT (the %Destruct object method).</td>
</tr>
<tr>
<td>/NOSTEP</td>
<td>Disable stepping through code modules. Options are :EXT (user language extensions); :METHOD (object methods); :DESTRUCT (the %Destruct object method).</td>
</tr>
</tbody>
</table>

The /TRACE command keyword specifies a trace output device that is used to receive trace messages. It must be specified before issuing a ZBREAK with action="T". It can be specified as a separate ZBREAK command (ZBREAK /TRACE:ON:device), or these two commands can be combined ZBREAK S:"T",/TRACE:ON:device.

Only one trace output device can be active at a time. Only trace messages are written to the trace output device; normal WRITE operations continue to write to the user terminal.

- /TRACE:ON activates the InterSystems Terminal as the recipient for trace messages.
- /TRACE:ON:device activates an existing output device (commonly a .txt file) as the recipient for trace messages. You must use the OPEN command to open the device before invoking ZBREAK /TRACE:ON:device. If you specify a device that is not open, Caché issues a <NOTOPEN> error. To open a sequential file, the directory must exist, and either the file must exist, or the OPEN command must specify the "N" option to create the file. This sequence of operations is shown in the following Windows example of a Terminal session:

  USER>SET btrace="C:\Logs\mydebugtrace.txt"
  USER>OPEN btrace:"WN"
  USER>ZBREAK /TRACE:ON:btrace
  USER>ZBREAK
  BREAK: Trace ON
  Trace device=C:\Logs\mydebugtrace.txt
  No breakpoints
  No watchpoints

If a prior ZBREAK /TRACE:ON:device1 has already activated a trace output device, ZBREAK /TRACE:ON:device2 replaces the device1 trace device with the device2 trace device.

- /TRACE:ALL enables line stepping, writing a message to the trace device for each line. This line stepping does not stop, but proceeds through the code being debugged. You can specify either ZBREAK /TRACE:ALL or ZBREAK /TRACE:ALL:device. /TRACE:ALL enables trace line stepping; you can invoke this option before or after activating a trace device using /TRACE:ON:device. /TRACE:ALL:device both activates a trace device and enable line stepping to that trace device. You must use the OPEN command to open the device before invoking /TRACE:ALL:device. If you specify a device that is not open, Caché issues a <NOTOPEN> error. To open a sequential file, the directory must exist, and either the file must exist, or the OPEN command must specify the "N" option to create the file. This sequence of operations is shown in the following Windows example of a Terminal session:
Routine and Debugging Commands

USER>SET steptrace="C:\Logs\mysteptrace.txt"
USER>OPEN steptrace:"WN"
USER>ZBREAK /TRACE:ALL:steptrace

ZBREAK
BREAK:L+ Trace ON
Trace device=C:\Logs\mysteptrace.txt
$ (single step) F:ET $:0 C: E:
No watchpoints

USER>

• /TRACE:OFF de-activates the current trace output device; it does not close the device. You can specify either ZBREAK /TRACE:OFF or ZBREAK /TRACE:OFF:device. /TRACE:OFF deactivates the current trace output device; if no trace device is active, Caché performs no operation and issues no error. /TRACE:OFF:device deactivates the specified trace output device. If the specified device is not the current trace output device, /TRACE:OFF:device issues a <NOTOPEN> error.

/STEP and /NOSTEP

The /STEP and /NOSTEP command keywords control whether the debugger steps through certain types of code modules:

• The :EXT option governs user-written language extensions created in %ZLANG. To an application, these language extensions appear as a single command or function call. By default, the debugger does not step through these %ZLANG routines, regardless of the action argument setting.

• The :METHOD option governs stepping through object methods called by the application. By default, the debugger steps through object method code.

• The :DESTRUCT option governs stepping through the %Destruct object method. The %Destruct method is implicitly called whenever an object is destroyed. By default, the debugger does not step through %Destruct object method code.

You can specify more than one /STEP or /NOSTEP option, as shown in the following example:

ZBREAK /STEP:METHOD:DESTRUCT

Examples

The following example shows how ZBREAK with no arguments lists breakpoints and watchpoints. The first ZBREAK lists no breakpoints or watchpoints. The program then sets two watchpoints on the x and y local variables, and ZBREAK displays this. Next the program sets a $ single-step breakpoint, and ZBREAK displays the breakpoint and the two watchpoints. Next the program disables the x local variable watchpoint; this has no effect on the ZBREAK display. Finally, the program removes the x local variable watchpoint; this watchpoint disappears from the ZBREAK display:

ZBREAK
ZBREAK *x:"B"
ZBREAK *y:"B"
ZBREAK
ZBREAK $
ZBREAK
ZBREAK -*x
ZBREAK
ZBREAK --*x
ZBREAK
ZBREAK /CLEAR

For further examples of ZBREAK usage, refer to the Debugging chapter in Using Caché ObjectScript.

See Also

• BREAK command

• FOR command
• OPEN command
• USE command
• Debugging chapter in Using Caché ObjectScript
ZINSERT

Inserts a line of code in the current routine.

ZINSERT:pc "code":location
ZI:pc "code":location

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>Optional — A postconditional expression.</td>
</tr>
<tr>
<td>code</td>
<td>A line of ObjectScript code, specified as a string literal (enclosed in quotation marks) or a variable that contains a string literal. The first character must be either a space or a label name.</td>
</tr>
<tr>
<td>location</td>
<td>Optional — The line after which ZINSERT inserts the code. Can be a label name, a numeric offset (+n) or a label name and a numeric offset. If you omit location, the code is inserted at the current line location (edit pointer).</td>
</tr>
</tbody>
</table>

Description

This command inserts a line of ObjectScript source code into the current routine and advances the edit pointer to immediately after the inserted line.

ZINSERT "code" inserts a specified line of ObjectScript code in the current routine at the current edit pointer position.

ZINSERT "code":location inserts the specified line of code in the current routine after the specified line location. You can specify a line location as the number of lines offset from the beginning of the routine, as a label, or as the number of lines offset from a specified label.

After ZINSERT inserts a line of code, it resets the edit pointer to the end of this new line of code. This means the next ZINSERT places its line of code directly after the last inserted line, unless the next ZINSERT explicitly specifies a location.

ZINSERT effects only the local copy of the routine. It does not change the routine as stored on disk. To store inserted lines, you must use the ZSAVE command to save the routine. ZINSERT incrementally compiles each line.

The ZINSERT command is usually executed from the programmer prompt. It should not be coded into the body of a routine, because its operation would effect the execution of that routine. To call ZINSERT from a routine, use the XECUTE command.

You can access the $ZNAME special variable to determine the name of the current routine.

The Edit Pointer

CAUTION: ZINSERT moves the edit pointer.

The edit pointer is set as follows:

- ZLOAD sets the edit pointer to the beginning of the routine.
- ZINSERT sets the edit pointer to immediately after the line it inserts. For example, specifying ZINSERT " SET x=1":+4 then ZINSERT " SET y=2" inserts lines 5 and 6.
- ZREMOVE sets the edit pointer to the line it removes. For example, specifying ZREMOVE +4 then ZINSERT " SET y=2" removes line 4 and replaces line 4 with the inserted line.
- ZPRINT (or PRINT) sets the edit pointer to the end of lines it printed. For example, specifying ZPRINT then ZINSERT " SET y=2" inserts the line at the end of the routine; specifying ZPRINT +1:+4 then ZINSERT " SET y=2" inserts the line as line 5.
• ZSAVE sets the edit pointer to the end of the routine. For example, specifying ZSAVE myroutine then ZINSERT " SET y=2" inserts the line at the end of the routine.

Arguments

pc

An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

code

A line of ObjectScript code, specified as a string literal (in quotes), or a variable that contains a string literal. This line of code can contain one or more ObjectScript commands, a new label name, or both a label and one or more commands. Because the code is inserted into a routine, it must follow ObjectScript formatting. Therefore, the first character of the code string literal must either be a blank space (standard ObjectScript indentation) or a label. The enclosing quotation marks are required. Since quotation marks enclose the line of code you are inserting, quotes within the code itself must be doubled.

location

The line after which ZINSERT will insert the code. It can take either of the following forms:

<table>
<thead>
<tr>
<th>+offset</th>
<th>An expression that resolves to a positive integer that identifies a line location as an offset number of lines from the beginning of the routine. ZINSERT inserts its code line immediately after this specified line. To insert a line at the beginning of the routine, specify +0. The plus sign is mandatory. If you omit +offset, the line identified by label is located.</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>A existing line label in the current routine. Must be a literal value; a variable cannot be used to specify label. Line labels are case-sensitive. If omitted, +offset is counted from the beginning of the routine.</td>
</tr>
<tr>
<td>label+offset</td>
<td>Specifies a label and a line count offset within the labelled section. If you omit the +offset value, or specify label+0, Caché locates the label line and inserts immediately after it.</td>
</tr>
</tbody>
</table>

Note: ZINSERT is only for use with the current routine. Attempting to specify label^routine for the location generates a <SYNTAX> error.

Lines of code are numbered beginning with 1. Thus a location of +1 inserts a line of code after the first line of the routine. To insert a line at the start of the routine or the start of a labelled section (before the existing first line), use an offset of +0.

For example:

```
ZINSERT "Altstart SET c=12,d=8":+0
```

inserts the code line at the start of the routine. By using an offset of +0 (or omitting the location), you can insert a line into an otherwise empty current routine.

A label may be longer than 31 characters, but must be unique within the first 31 characters. ZINSERT matches only the first 31 characters of a specified label. Label names are case-sensitive, and may contain Unicode characters.

Examples

The following example inserts the code line SET x=24 after the fourth line within the current routine. Because this inserted code line does not begin with a label, an initial space must be included as the required line start character.
ZINSERT * SET x=24*:+4

In the following example, assume that the currently loaded routine contains a label called "Checktest". The ZINSERT command inserts a new line after the sixth line within Checktest (Checktest+6). This new line contains the label "Altcheck" and the command SET y=12.

ZINSERT "Altcheck SET y=12":Checktest+6

Note that because the inserted code line begins with the label “Altcheck”, no initial space is required after the quotation mark.

The following example inserts the code line SET x=24 WRITE !,"x is set to ",x after the fourth line within the current routine. Because an inserted code line is enclosed in quotation marks, the quotation marks in the WRITE command must be doubled.

ZINSERT " SET x=24 WRITE !,""x is set to ",x":+4

Notes

ZINSERT and ZREMOVE

You can use the ZREMOVE command to remove one or more lines of code from the currently executing routine. Thus by using ZREMOVE and ZINSERT, you can substitute a new code line for an existing code line. These operations only affect the copy of the routine currently being run by your process.

Note: ZINSERT inserts a line after the specified location. ZREMOVE removes a line at the specified location. For example, if you insert a line with ZINSERT " SET x=1":+4, to remove this line you must specify ZREMOVE +5.

ZINSERT, XECUTE, and $TEXT

You use the XECUTE command to define and insert a single line of executable code from within a routine. You use the ZINSERT command to define and insert by line position a single line of executable code from outside a routine.

An XECUTE command cannot be used to define a new label. Therefore, XECUTE does not require an initial blank space before the first command in its code line. ZINSERT can be used to define a new label. Therefore ZINSERT does require an initial blank space (or the name of a new label) before the first command in its command line.

The $TEXT function permits you to extract a line of code by line position from within a routine. $TEXT simply copies the specified line of code as a text string; it does not affect the execution of that line or change the current line location (edit pointer) when extracting from the current routine. (Using $TEXT to extract code from a routine other than the current routine does change the current line location.) $TEXT can supply a line of code to the XECUTE command. $TEXT can also supply a line of code to a WRITE command, and thus supply a code line to the programmer prompt.

See Also

- XECUTE command
- ZLOAD command
- ZPRINT command
- ZREMOVE command
- ZSAVE command
- $TEXT function
- $ZNAME special variable
ZLOAD

Loads a routine into the current routine buffer.

ZLOAD:pc routine
ZL:pc routine

Arguments

| pc       | Optional — A postconditional expression. |
| routine  | Optional — The routine to be loaded. If omitted, Caché loads a routine from the current device. |

Description

The ZLOAD command loads an ObjectScript routine into the routine buffer. ZLOAD has two forms:

- ZLOAD without an argument
- ZLOAD with an argument

ZLOAD without an Argument

The ZLOAD command without an argument loads an ObjectScript routine from the current device into the routine buffer. To load a routine from a device, execute the following:

1. An OPEN command to open the device.
2. A USE command to make the device the current device.
3. A ZLOAD command without arguments.

Line loading will continue until Caché reads a null string line ("""). This loaded routine has no name until you file it with the ZSAVE command.

ZLOAD with an Argument

ZLOAD routine loads the specified (existing) routine into the routine buffer from the current namespace.

ZLOAD does an implicit argumentless ZREMOVE when it loads the routine. That is, ZLOAD deletes any routine previously loaded as it writes the loaded routine. You can use the SZNAME special variable to determine the currently loaded routine. When ZLOAD loads a routine, it positions the line pointer at the beginning of the routine.

Once loaded, a routine remains the current one for the process until you load another routine explicitly, with a ZLOAD command, or implicitly, with a DO or a GOTO command.

As long as the routine is current, you can edit the routine (with ZINSERT and ZREMOVE commands), display one or more lines with the ZPRINT command, or return a single line with the $TEXT function.

You can only use the ZLOAD command when you enter it from the programmer prompt or when you call it using an XECUTE command or a $XECUTE function. It should not be coded into the body of a routine because its operation would affect the execution of that routine. Specifying ZLOAD in a routine results in a compile error. Any attempt to execute ZLOAD from within a routine also generates an error.
Arguments

pc

An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

routine

The name of the routine to be loaded. Routine names are case-sensitive. If the specified routine does not exist, the system generates a <NOROUTINE> error.

You must have execute permission for routine to be able to ZLOAD it. If you do not have this permission, Cache generates a <PROTECT> error.

If ZLOAD successfully loads routine, all subsequent errors append the name of the currently loaded routine. This occurs whether or not the error has any connection to the routine, and occurs across namespaces. For further details, refer to the $ZERROR special variable.

Examples

The following Terminal example loads the routine ROUT, saved on disk:

USER>ZLOAD ROUT

The following Terminal example loads the first routine from the device dev:

USER>OPEN dev
USER>USE dev
USER>ZLOAD

Notes

Routine Behavior with ZLOAD

If you specify routine, Caché looks for the routine in the pool of routine buffers in memory. If the routine is not there, Caché loads the ObjectScript object code version of the routine into one of the buffers. The ObjectScript source (intermediate) code remains in the ^ROUTINE global of the current namespace, but is updated if you make edits and save the changes.

For example, ZLOAD Test loads the object code version of the routine Test (if it is not already loaded). You can view and edit the contents of the ^ROUTINE global using the Management Portal. Select System Explorer, Globals, then select the current namespace from the drop-down list of namespaces in the left-hand column.

If you omit routine, Caché loads new lines of code that you enter from the current device, usually the keyboard until you terminate the code by entering a null line (that is, just press <Return>). This routine has no name until you save it with a subsequent ZSAVE command.

ZLOAD and Language Modes

When a routine is loaded, the current language mode changes to the loaded routine’s language mode. At the conclusion of called routines, the language mode is restored to the language mode of the calling routine. However, at the conclusion of a routine loaded with ZLOAD the language mode is not restored to the previous language mode. For further details on checking and setting language modes, refer to the LanguageMode() method of the %SYSTEM.Process class.

See Also

• OPEN command
• USE command
• XECUTE command
• ZSAVE command
• $XECUTE function
• $ZNAME special variable
ZPRINT

Displays lines of code from the current routine on the current device.

\[
\text{ZPRINT:pc lineref1:lineref2} \\
\text{ZF:pc lineref1:lineref2}
\]

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>Optional — A postconditional expression.</td>
</tr>
<tr>
<td>lineref1</td>
<td>Optional — The line to be displayed, or the first line in a range of lines to be displayed. Can be a label name, a numeric offset (+n) or a label name and a numeric offset. If omitted, the entire current routine is displayed.</td>
</tr>
<tr>
<td>:lineref2</td>
<td>Optional — The last line in a range of lines to be displayed. To define a range, lineref1 must be specified.</td>
</tr>
</tbody>
</table>

**Description**

The ZPRINT command displays lines of code from the currently loaded routine. The output is sent to the current device. You establish the current device with the USE command. For the name of the current routine, access the $ZNAME special variable. For the device ID of the current device, access the $IO special variable.

**Note:** The ZPRINT and PRINT commands are functionally identical.

The displayed lines include all labels, comments, and whitespace, with the exception that entirely blanks lines in a MAC routine, which are removed by the compiler, are neither displayed nor counted. For this reason, ZPRINT displays and counts the following multi-line comment as two lines, not three:

```
/* This comment includes
   a blank line */
```

ZPRINT has two forms:
- Without arguments
- With arguments

ZPRINT without arguments displays all the lines of code in the currently loaded routine.

ZPRINT with arguments displays the specified lines of code. ZPRINT lineref1 displays the line specified by lineref1. ZPRINT lineref1:lineref2 displays the range of lines starting with lineref1 and ending with lineref2 (inclusive).

The lineref arguments count lines and line offsets using the object code version of the routine. After modifying a routine, you must re-compile the routine for ZPRINT to correctly count lines and line offsets that correspond to the source version.

You can use the $TEXT function to return a single line of code.

**Arguments**

**pc**

An optional postconditional expression. Caché executes the ZPRINT command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.
lineref1

The line to be printed or the first in a range of lines to be displayed or printed. Can be specified in either of the following syntactical forms:

• `+offset` where `offset` is a positive integer specifying the line number within the current routine. +1 is the first line in the routine, which may be a label line. +0 always returns the empty string.

• `label+offset` where `label` is a label within the routine and `offset` is the line number counting from the label (with the label itself counting as offset 0). If you omit the `offset` option, or specify `label+0`, Caché prints the label line. `label+1` prints the line after the label.

A label may be longer than 31 characters, but must be unique within the first 31 characters. ZPRINT matches only the first 31 characters of a specified `label`. Label names are case-sensitive, and may contain Unicode characters.

lineref2

The last line in a range of lines to be displayed. Specify in the same way as lineref1. lineref1 must be specified to specify lineref2. lineref1 and lineref2 are separated by a colon (:) character. No whitespace may appear between the colon and lineref2.

If lineref2 is earlier in the line sequence than lineref1, ZPRINT ignores lineref2 and displays the single line of code specified by lineref1.

Examples

Given the following lines of code:

```objectscript
AviationLetters
Abc
    WRITE "A is Abel",!
    WRITE "B is Baker",!
    WRITE "C is Charlie",!
Def WRITE "D is Delta",!
    /* Not sure about E */
    WRITE "F is Foxtrot",!
```

ZPRINT with no lineref arguments displays all nine lines, including the comment line.

ZPRINT +0 displays the empty string.

ZPRINT +1 displays the AviationLetters label.

ZPRINT +8 displays the /* Not sure about E */ comment line.

ZPRINT +10 displays the empty string.

ZPRINT Def or ZPRINT Def+0 display the Def WRITE "D is Delta",! line. This is a label line that also includes executable code.

ZPRINT Def+1 displays the WRITE "E is Epsilon",! line.

Range Examples

ZPRINT +0:+3 displays the empty string.

ZPRINT +1:+3 displays the first three lines.

ZPRINT +3:+3 displays the third line.

ZPRINT +3:+1 displays the third line; lineref2 is ignored.

ZPRINT +3:Abc+1 displays the third line. Both lineref1 and lineref2 are specifying the same line.
ZPRINT +3:abc+1 displays from the third line to the end of the routine. Line labels are case-sensitive, so the range endpoint was not found.

ZPRINT Abc+1:+4 displays lines 3 and 4.

ZPRINT Abc+1:Abc+2 displays lines 3 and 4.

ZPRINT Abc:Def displays lines 2, 3, 4, 5, and 6.

ZPRINT Abc+1:Def displays lines 3, 4, 5, and 6.

ZPRINT Def:Abc displays the Def WRITE "D is Delta",! line. Because lineref2 is earlier in the code, it is ignored.

See Also

- PRINT command
- ZINSERT command
- ZLOAD command
- ZREMOVE command
- ZSAVE command
- ZZPRINT command
- $TEXT function
- $IO special variable
- Comments in Using Caché ObjectScript
- Labels in Using Caché ObjectScript
- The Spool Device in Caché I/O Device Guide
ZREMOVE

Erasers a line or range of lines from the current routine.

ZREMOVE:pc lineref1:lineref2
ZR:pc lineref1:lineref2

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pc</code></td>
<td>Optional — A postconditional expression.</td>
</tr>
<tr>
<td><code>lineref1</code></td>
<td>Optional — The single line, or first line in a range of lines to be removed. If you omit <code>lineref1</code>, ZREMOVE deletes the entire current routine.</td>
</tr>
<tr>
<td><code>:lineref2</code></td>
<td>Optional — The last in a range of lines to be removed.</td>
</tr>
</tbody>
</table>

Description

The ZREMOVE command erases routine lines from the current routine. ZREMOVE has two forms:

- Without an argument
- With arguments

You can only use the ZREMOVE command when you enter it at the programmer prompt or when you call it using an XECUTE command. It should not be coded into the body of a routine because its operation would effect the execution of that routine.

Only the local copy of the routine is affected, not the routine as stored on disk. To store the modified code, you must use the ZSAVE command to save the routine.

**Without an Argument**

ZREMOVE without an argument erases the entire current routine. The following Terminal session shows this operation:

```
USER>DO ^myroutine
USER>ZLOAD myroutine
USER>WRITE $ZNAME myroutine
USER>ZREMOVE
USER>WRITE $ZNAME
USER>
```

**With Arguments**

ZREMOVE with arguments erases code lines in the current routine. ZREMOVE `lineref1` erases the specified line. ZREMOVE `lineref1:lineref2` erases the range for lines starting with the first line reference and ending with the second line reference, inclusive.

The following Terminal session shows this operation. This example uses a dummy routine (^myroutine) in which each line sets a variable to a string naming that line:
Arguments

**pc**

An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

**lineref1**

The line to be removed, or the first in a range of lines to be removed. It can take any of the following formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+offset</td>
<td>Specifies a line number within the routine. A positive integer, counting from 1.</td>
</tr>
<tr>
<td>label</td>
<td>Specifies a label within the routine. ZREMOVE erases only the label line itself. This includes any code that follows the label on that line.</td>
</tr>
<tr>
<td>label+offset</td>
<td>Specifies a label and the number of line to offset within the labeled section, counting the label line as line 1.</td>
</tr>
</tbody>
</table>

A label may be longer than 31 characters, but must be unique within the first 31 characters. ZREMOVE matches only the first 31 characters of a specified label. Label names are case-sensitive, and may contain Unicode characters.

You can use lineref1 to specify a single line of code to remove. You specify the code line either as an offset from the beginning of the routine (+lineref1) or as an offset from a specified label (module2+lineref1).

- **ZREMOVE +7**: removes the 7th line counting from the beginning of the routine.
- **ZREMOVE +0**: performs no operation, generates no error.
- **ZREMOVE +999**: if 999 is greater than the number of lines in the routine, performs no operation, generates no error.
- **ZREMOVE Test1**: removes the label line Test1.
- **ZREMOVE Test1+0**: removes the label line Test1.
- **ZREMOVE Test1+1**: removes the first line following label line Test1.
- **ZREMOVE Test1+999**: removes the 999th line following label line Test1. This line may be in another labeled module. If 999 is greater than the number of lines from label Test1 to the end of the routine, performs no operation, generates no error.

Labels are counted as code lines in a line offset. Whitespace lines are not counted as code lines in line offset. Comments are not counted as code lines in a line offset, with one exception. A single-line comment delimited by ;; is counted as a code line for the purposes of line offset.

**lineref2**

The last line in a range of lines to be removed. Specify lineref2 in any of the formats used for lineref1. The colon prefix (:) is mandatory.
You specify a range of lines as +lineref1:+lineref2. **ZREMOVE** removes the range of lines, inclusive of lineref1 and lineref2. If lineref1 and lineref2 refer to the same line, **ZREMOVE** removes that single line.

If lineref2 appears earlier in the routine code than lineref1, no operation is performed and no error is generated. For example: ZREMOVE +7:+2, ZREMOVE Test1+1:Test1, ZREMOVE Test2:Test1 would perform no operation.

**Examples**

This command erases the fourth line within the current routine.

```
ZREMOVE +4
```

This command erases the sixth line after the label Test1.

```
ZREMOVE Test1+6
```

This command erases lines three through ten, inclusive, within the current routine.

```
ZREMOVE +3:+10
```

This command erases the label line Test1 through the line that immediately follows it, within the current routine.

```
ZREMOVE Test1:Test1+1
```

This command erases all of the line from label Test1 through label Test2, inclusive of both labels, within the current routine.

```
ZREMOVE Test1:Test2
```

**See Also**

- **PRINT** command
- **XECUTE** command
- **ZINSERT** command
- **ZLOAD** command
- **ZSAVE** command
- **$ZNAME** special variable
ZSAVE

Saves the current routine.

ZSAVE:pc routine
ZS:pc routine

Arguments

<table>
<thead>
<tr>
<th>pc</th>
<th>Optional — A postconditional expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>routine</td>
<td>Optional — A new name for the routine. Must be a valid identifier.</td>
</tr>
</tbody>
</table>

Description

The ZSAVE command saves the current routine. You use ZLOAD to load the routine, then use ZSAVE to save any changes you have made to the routine with ZINSERT and ZREMOVE commands. ZSAVE is also used to name a routine loaded with an argumentless ZLOAD, or a nameless routine created by ZINSERT commands.

CAUTION: ZSAVE moves the edit pointer to the end of the routine. Issuing ZSAVE moves the edit pointer even if no actual save was needed.

If you ZLOAD a routine and then ZSAVE it to the same routine name without making any changes to the routine, Caché does not update the routine’s timestamps or generated flag.

You can use ZSAVE in two contexts: from the Terminal prompt, or invoked by an XECUTE command within a program.

ZSAVE has two forms:
- Without an argument
- With an argument

ZSAVE without an argument saves the current routine under its current name (that is, the name under which you previously saved it) on disk in the current namespace. If the current routine does not yet have a name, an argumentless ZSAVE generates a <COMMAND> error.

ZSAVE routine saves the current routine to disk as the specified routine name in the current namespace.

If use the XECUTE command to invoke ZSAVE routine, the system creates a Load frame to preserve the current routine. When the XECUTE command concludes, Caché uses this Load frame to restore the routine name prior to the XECUTE as the current routine. This is shown in the following example:

```
WRITE "Current routine name",!
WRITE "initial name: ",$ZNAME,!
SET x = "WRITE "$ZNAME"
SET y = "ZSAVE mytest"
SET z = "WRITE "' changed to "',$ZNAME,!
XECUTE x,y,z
WRITE "restored name: ",$ZNAME,!
```

Arguments

pc

An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.
**routine**

A name under which to save the routine.routine must be a valid routine name. You can use the $ZNAME("string",1)function to determine if string is a valid routine name. You can use the $ZNAME special variable to determine the name of the currently loaded routine.

Commonly, routine is a new name for the routine, but it can be the current routine name. If a routine by that name already exists, Caché overwrites it. Note that you are not asked to confirm the overwrite. A routine name must be unique within the first 255 characters; routine names longer than 220 characters should be avoided.

If you specify ZSAVE routine in an X

If you omit routine, the system saves the routine under its current name. If no current name exists, ZSAVE generates a <COMMAND> error.

**Example**

The following Terminal session example executes a ZSAVE command to save the currently loaded routine:

```plaintext
USER>DO ^myroutine
this is line 8
this is line 9
USER>ZLOAD myroutine

USER>PRINT +8
WRITE "this is line 8",!
USER>ZREMOVE +8

USER>PRINT +8
WRITE "this is line 9",!
USER>ZSAVE myroutine

USER>DO ^myroutine
this is line 9
USER>
```

**Notes**

**Where to Use ZSAVE**

The ZSAVE command works only when entered at the programmer prompt or when coded within an XECUTE statement in a routine. It should not be coded into the body of a routine because its operation would effect the execution of that routine. If you code ZSAVE in a routine, but outside of an XECUTE statement, that ZSAVE saves the current program.

**ZSAVE and Routine Recompilation**

If you have issued a command that modifies source code, ZSAVE recompiles and saves the routine. If the source code for the routine is unavailable, ZSAVE fails with a <NO SOURCE> error and does not replace the existing object code. For example, the following commands load the %SS object code routine, attempt to remove lines from the (nonexistent) source code, and then attempt to save to the ^text global. This operation fails with a <NO SOURCE> error:

```
ZLOAD %SS ZREMOVE +3 ZSAVE ^test
```

If you have not issued a command that modifies source code, ZSAVE saves the object code in the specified routine. (Obviously, no recompile occurs.) For example, the following commands load the %SS object code routine and then save it to the ^text global. This operation succeeds:

```
ZLOAD %SS ZSAVE ^test
```

**ZSAVE with % Routines**

You receive a <PROTECT> error if you try to ZSAVE a %routine to a remote dataset, even if that dataset is the current dataset for the process. The percent sign prefix is used for the names of non-modifiable routines, such as system utilities.
Concurrent ZSAVE Operations

When using ZSAVE in a networked environment, a situation may occur in which two different jobs might concurrently save a routine and assign it the same name. This operation has the potential for one routine overwriting part of the other, producing unpredictable results. When this possibility exists, acquire an advisory lock on the routine before the ZSAVE operation. For example, LOCK ^ROUTINE("name"). For further details, refer to the LOCK command. When running a job across ECP, the saved source is more vulnerable to such concurrent saves because local buffer protection is not visible to other clients.

See Also

- XECUTE command
- ZLOAD command
- ZINSERT command
- ZREMOVE command
- $ZNAME special variable
ZZPRINT

Displays one or more source code lines from a routine.

ZZPRINT:pc "entry+offset^routine":before:after

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>Optional — A postconditional expression.</td>
</tr>
<tr>
<td>entry</td>
<td>Optional — The name of an entry (label) within routine. If omitted, ZZPRINT begins at the start of routine (line 0).</td>
</tr>
<tr>
<td>offset</td>
<td>Optional — An integer specifying the number of lines to offset from entry (if specified), or from the beginning of routine (if entry is omitted). An offset requires the plus sign (+) prefix. Commonly offset is a positive integer. An offset can be a negative number; for example, &quot;subsect+-3^mytest&quot;, which displays the line 3 lines prior to subsect. If offset is omitted, display begins at start of entry.</td>
</tr>
<tr>
<td>routine</td>
<td>The routine from which to display source code lines. A routine name is always preceded by a caret (^) prefix.</td>
</tr>
<tr>
<td>before</td>
<td>Optional — A positive integer count specifying the number of lines before the line specified in &quot;entry+offset^routine&quot; to display. Does not include the &quot;entry+offset^routine&quot; line.</td>
</tr>
<tr>
<td>after</td>
<td>Optional — A positive integer count specifying the number of lines after the line specified in &quot;entry+offset^routine&quot; to display. Does not include the &quot;entry+offset^routine&quot; line.</td>
</tr>
</tbody>
</table>

Description

The ZZPRINT command displays one or more source code lines from a specified ObjectScript routine. It can display a designated line of code and a specified number of lines that appear before and/or after that designated line of source code, enabling you to see it in context. The entry value specifies a starting point for display. However, ZZPRINT does not limit its display to the specified entry; offset, before, and after may include source code lines in prior or subsequent entries.

The displayed lines include all labels, comments, and whitespace, with the exception of entirely blanks lines. A blank line, whether in code or within a multiline comment, is neither displayed nor counted. For this reason, ZZPRINT displays and counts the following multi-line comment as two lines, not three:

```caché
/* This comment includes
   a blank line */
```

ZZPRINT does not count or display preprocessor statements.

Command Delimiters and Blank Space Requirements

The delimiting quotation marks (" ") are required. The opening quotation mark must appear exactly one space after the command name or postconditional expression. Any number of spaces can appear following the opening quotation mark or proceeding or following the closing quotation mark.

The caret (^) prefix to the routine name is required. If you specify an offset, any number of spaces can precede the caret prefix. If you do not specify an offset, there must be no spaces between the entry name and caret prefix to the routine name.

The plus sign (+) is required to specify an offset. The plus sign must immediately follow the entry name (if specified). A plus sign not immediately followed by an integer represents an offset of zero. For example, a plus sign immediately followed
by the caret (^) prefix, by a space, or by a non-number character all represent an offset of zero. A plus sign immediately followed by a integer represents a positive offset. A plus sign immediately followed by a negative integer represents a negative offset.

A colon (:) delimiter is required to specify a before line count. To specify an after line count you must specify both colon delimiters, whether or not you specify a before value. Any number of spaces are permitted before or after these colon delimiters.

**Arguments**

**pc**

An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

**entry**

The name of an entry within routine. Entry names are case-sensitive. If routine does not contain the specified entry, the system generates a <NOLABEL> error.

**offset**

An integer specifying the number of lines to offset from the beginning of entry (if specified), or the beginning of routine. Omitting offset is the same as an offset of zero. An offset of zero from entry is the entry label line itself (line 1). An offset of zero from routine is the (nonexistent) line prior to the beginning of the routine (line 0). Therefore, to display a comment line at the beginning of a routine, you must specify an offset of 1 ("+1^mytest").

Offset counts include all source code lines except preprocessor statements and completely blank lines. These are neither displayed nor counted. An offset that exceeds the available lines in routine when counting from the specified (or implied) entry point returns a null string.

**routine**

The name of the routine from which to display source code lines. Routine names are case-sensitive. If the specified routine does not exist, the system generates a <NOROUTINE> error.

You must have read permission for routine to be able to ZZPRINT it. If you do not have this permission, Cache generates a <PROTECT> error.

**before**

A positive integer specifying the number of lines to display before the specified line. This enables you to view a source code line in the context of the lines immediately prior to it. This is a count of lines backwards from a designated program line. The before count does not include the program line designated by "entry+offset^routine". If the before count is larger than the number of available code lines, ZZPRINT displays the available code lines; it does not issue an error.

**after**

A positive integer specifying the number of lines to display after the specified line. This enables you to view a source code line in the context of the lines immediately following it. This is a count of lines forwards from a designated program line. The after count does not include the program line designated by "entry+offset^routine". If the after count is larger than the number of available code lines, ZZPRINT displays the available code lines; it does not issue an error.

When you specify an after value, you can specify or omit a before value:

- To display a designated program line and a specified number of lines after that line, specify a positive integer for after and a value of 0 for before.
To display all lines in the routine prior to the designated program line and a specified number of lines after that line, specify an after value without specifying a before value. (Just specify the placeholder colon.) If you specify an after without a before, ZZPRINT displays from the start of routine to the location after the designated program line specified by the after line count.

To display a specified number of lines before and after a designated program line, specify a positive integer for before and a positive integer for after.

See Also

- PRINT command
- ZPRINT command
- $TEXT function
- Comments in Using Caché ObjectScript
- Labels in Using Caché ObjectScript
ObjectScript Functions

A function performs an operation and returns a value. This value may be the result of the operation, or an indicator that the operation completed successfully or failed. By convention, Caché functions that set a variable to a value set the variable, then return the value of that variable prior to the operation.

This document describes system functions (also known as intrinsic functions). System functions are identified by a $ character prefix to the name and parentheses following the name. The parentheses are not specified when referring to a function in documentation. You can supplement these system functions by creating user-supplied functions (also known as extrinsic functions). User-supplied functions are identified by a $$ prefix.

The names of ObjectScript special variables also begin with a $ character, but special variables have no parentheses.

The ObjectScript $ZUTIL (SZU) functions are deprecated and are no longer documented. Use the $ZUTIL Replacement List to find appropriate substitutes.

ObjectScript functions that are deprecated are listed separately as Legacy Functions. These should not be used for new coding.

For more information on ObjectScript functions generally, see the Functions description in the “Introducing ObjectScript” chapter of Using ObjectScript; for more information on defining your own functions, see User-Defined Code in Using ObjectScript.

To invoke a system function, use the form:

$\text{name} (\text{parameters})$

<table>
<thead>
<tr>
<th>name</th>
<th>The name of the function. The preceding dollar sign ($) is required. Function names are shown here in all uppercase letters, but they are, in fact, not case-sensitive. You can abbreviate most function names. In the Synopsis for each function, the full name syntax is first presented, and below it is shown the abbreviated name (if one exists).</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameters</td>
<td>One or more values to be passed to the function. Function arguments, or parameters, are always enclosed in parentheses and follow the function name. The parentheses are mandatory, even if the function has no parameters. Multiple parameters are separated from each other by commas. The parameters are positional and must match the order of the parameters expected by the function. Missing parameters in this sequence can be indicated by supplying the appropriate number of commas; no trailing commas are required for parameters missing from the end of the parameter list. Spaces are permitted anywhere in the parameter list. No spaces are permitted between \text{name} and the open parenthesis character.</td>
</tr>
</tbody>
</table>

The Synopsis for each function contains only literal syntactical punctuation. The Synopsis does not include punctuation for format conventions, such as what parameters of the syntax are optional. This information is provided in the table of parameters immediately following the Synopsis.

The one exception is the ellipsis (...). An ellipsis following a comma indicates that the parameter (or parameter group) preceding the comma can be repeated multiple times as a comma-separated list.

Any platform-specific function is marked with the name of the platform that supports it. Any function that is not marked with a platform abbreviation is supported by all Caché platforms.
$ASCII

Converts a character to a numeric code.

$ASCII(expression,position)  
$A(expression,position)

**Parameters**

<table>
<thead>
<tr>
<th>expression</th>
<th>The character to be converted.</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>Optional — The position of a character within a character string, counting from 1. The default is 1.</td>
</tr>
</tbody>
</table>

**Description**

$ASCII returns the character code value for a single character specified in *expression*. This character can be an 8-bit (extended ASCII) character or a 16-bit (Unicode) character. The returned value is a positive integer.

The *expression* parameter may evaluate to a single character or to a string of characters. If *expression* evaluates to a string of characters, you can include the optional *position* parameter to indicate which character you want to convert.

**Parameters**

*expression*

An expression that evaluates to a quoted string of one or more characters. The expression can be specified as the name of a variable, a numeric value, a string literal, or any valid ObjectScript expression. If *expression* yields a string of more than one character, use *position* to select the desired character. If you omit *position* for a character string, $ASCII returns the numeric code for the first character. $ASCII returns -1 if the *expression* evaluates to a null string.

*position*

The position must be specified as a nonzero positive integer. It may be signed or unsigned. You can use a noninteger numeric value in *position*; however, Caché ignores the fractional portion and only considers the integer portion of the numeric value. If you do not include *position*, $ASCII returns the numeric value of the first character in *expression*. $ASCII returns -1 if the integer value of *position* is larger than the number of characters in *expression* or less than 1.

**Examples**

The following example returns 87, the ASCII numeric value of the character W.

```objectscript
WRITE $ASCII("W")
```

The following example returns 960, the numeric equivalent for the Unicode character "pi".

```objectscript
WRITE $ASCII($CHAR(959+1))
```

The following example returns 84, the ASCII numeric equivalent for the first character in the variable Z.

```objectscript
SET Z="TEST"
WRITE $ASCII(Z)
```

The following example returns 83, the ASCII numeric equivalent for the third character in the variable Z.

```objectscript
SET Z="TEST"
WRITE $ASCII(Z,3)
```
The following example returns a -1 because the second argument specifies a position greater than the number of characters in the string.

```
SET Z="TEST"
WRITE ASCIIZ,5
```

The following example uses $ASCII in a FOR loop to convert all of the characters in variable x to their ASCII numeric equivalents. The $ASCII reference includes the position parameter, which is updated for each execution of the loop. When position reaches a number that is greater than the number of characters in x, $ASCII returns a value of -1, which terminates the loop.

```
SET x="abcdefghijklmnopqrstuvwxyz"
FOR i=1:1 {
    QUIT:$ASCII(x,i)=−1
    WRITE !,$ASCII(x,i)
}
QUIT
```

The following example generates a simple checksum for the string X. When $CHAR(CS) is concatenated with the string, the checksum of the new string is always zero. Therefore, validation is simplified.

```
CXSUM
SET x="Now is the time for all good men to come to the aid of their party"
SET CS=0
FOR i=1:1:$LENGTH(x) {
    SET CS=CS+$ASCII(x,i)
    WRITE !,"Checksum is":,CS
}
SET CS=128−CS#128
WRITE !,"Final checksum is":,CS
```

The following example converts a lowercase or mixed-case alphabetic string to all uppercase.

```
ST
SET String="ThIs Is a MiXeDCAse stRiNg"
WRITE !,"Input: ",String
SET Len=$LENGTH(String),Nstring=" 
FOR i=1:1:Len { DO CNVT }
QUIT
CNVT
SET Char=$EXTRACT(String,i),Asc=$ASCII(Char)
IF Asc>96,Asc<123 {
    SET Char=$CHAR(Asc−32)
    SET Nstring=Nstring_Char
}
ELSE {
    SET Nstring=Nstring_Char
}
WRITE !,"Output: ",Nstring
QUIT
```

**Notes**

**Unicode Support**

The $ASCII function supports both 8-bit and 16-bit characters. For 8-bit characters, it returns the numeric values 0 through 255. For 16-bit (Unicode) characters it returns numeric codes up to 65535.

The Unicode value for a character is usually expressed as a 4-digit number in hexadecimal notation, using the digits 0-9 and the letters A-F (for 10 through 15, respectively). However, standard functions in the ObjectScript language generally identify characters according to ASCII numeric codes, which are base-10 decimal values, not hexadecimal.

Hence, the $ASCII function supports Unicode encoding by returning the decimal Unicode value of the inputted character, instead of the hexadecimal value that the Unicode standard recommends. This way, the function remains backward compatible, while also supporting Unicode. To convert a decimal number to hexadecimal, use the $ZHEX function.

For further details on Caché Unicode support, refer to Unicode in *Using Caché ObjectScript*. 

Caché ObjectScript Reference
Surrogate Pairs

$ASCII does not recognize surrogate pairs. Surrogate pairs are used to represent some Chinese characters and to support the Japanese JIS2004 standard. You can use the $WISWIDTH function to determine if a string contains a surrogate pair. The $WASCII function recognizes and correctly parses surrogate pairs. $ASCII and $WASCII are otherwise identical. However, because $ASCII is generally faster than $WASCII, $ASCII is preferable for all cases where a surrogate pair is not likely to be encountered.

Note: $WASCII should not be confused with $ZWASCII, which always parses characters in pairs.

Related Functions

The $CHAR function is the inverse of $ASCII. You can use it to convert an integer code to a character. $ASCII converts a single character to an integer. To convert a 16-bit (wide) character string to an integer use $ZWASCII. To convert a 32-bit (long) character string to an integer use $ZLASCII. To convert a 64-bit (quad) character string to an integer use $ZQASCII. To convert a 64-bit character string to an IEEE floating-point number ($DOUBLE data type) use $ZDASCII.

See Also

- READ command
- WRITE command
- $CHAR function
- $WASCII function
- $WISWIDTH function
- $ZLASCII function
- $ZWASCII function
- $ZQASCII function
- $ZDASCII function

ObjectScript Functions
Returns and or sets the bit value of a specified position in a bitstring.

$BIT(bitstring,position)

SET $BIT(bitstring,position) = value

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitstring</td>
<td>An expression that evaluates to a bitstring. For $BIT, bitstring can be any expression that resolves to a bitstring, including a variable of any type, $FACTOR, a user-defined function, or an oref.prop, ..prop, or i%prop property reference. For SET $BIT, bitstring can be a variable of any type, including an i%Prop() property instance variable.</td>
</tr>
<tr>
<td>position</td>
<td>The bit position within bitstring. A literal or an expression that evaluates to a positive integer. Bit positions are counted from 1.</td>
</tr>
<tr>
<td>value</td>
<td>The bit value to set at position. A literal or an expression that evaluates to the integer 0 or 1.</td>
</tr>
</tbody>
</table>

**Description**

$BIT is used to return a bit value from a compressed bit string. $BIT(bitstring,position) returns the bit value (0 or 1) at the specified position, position, in the given bitstring expression bitstring. If no value has been defined for a position, $BIT returns 0 for that position. The position is counted from 1; if position is less than 1 (0 or a negative number) or is greater than the length of the bitstring, the function returns 0.

If bitstring is an undefined variable or the null string (""") the $BIT function returns 0. If bitstring is not a valid bitstring value an <INVALID BIT STRING> error occurs.

For general information on $BIT and other bitstring functions, see below.

**SET $BIT**

SET $BIT is used to set a specified bit value in a bitstring compressed bit string. If the bitstring is not defined, SET $BIT defines the bitstring variable as a compressed bit string and sets the specified bit value.

SET $BIT(bitstring,position) = value performs an atomic bit set on the bitstring specified by bitstring. If value is 1, then the bit at position position is set to 1. If value is 0, the bit is cleared (set to 0). Only an integer value of 0 or 1 should be used; Caché converts any non-numeric value, such as “true” or “false” to 0.

The bit position is counted from 1. If bitstring is shorter than the specified position, Caché pads the bitstring with 0 bits to the specified position. If you specify a position of 0, the system generates a <VALUE OUT OF RANGE> error.

The bitstring variable must be either an undefined variable, a variable already set to a bitstring value, or a variable set to the empty string ("""). Attempting to use SET $BIT on a variable already set to a non-bitstring value results in an <INVALID BIT STRING> error.

The SET $BIT bitstring parameter does not support oref.property or .. property syntax.

The SET $BIT bitstring parameter supports i%property syntax for both local (non-inherited) properties and properties inherited from a super class. If attempting to set inherited property in existing code is generating a <FUNCTION> error, recompiling the routine should resolve this error and allow setting of the inherited property.
**Displaying a Bitstring**

As shown in the examples, you can use `WRITE` to display the contents of an individual bit in a bitstring as the return value of `$BIT`.

Caché has several commands to display the contents of a variable. However, because a `$BIT` bitstring is a compressed binary string, `WRITE` does not display a useful value. `ZZDUMP` displays the hexadecimal representation of the compressed binary string, which is also not a useful value for most purposes.

`ZWRITE` and `ZZWRITE` display the decimal representation of the compressed binary string as `$ZWCHAR ($zwc)` two-byte (wide) characters. However, they also display a comment that lists the uncompressed “1” bits in left-to-right order as a comma-separated list. If there are three or more consecutive “1” bits, it lists them as a range (inclusive) with two dot syntax (n..m). For example, the bitstring `[1,0,1,1,1,0,1]` is shown as /*$bit(1,3..6,8)*/. The bitstring `[1,1,1,1,1,1,1]` is shown as /*$bit(1..8)*/. The bitstring `[0,0,0,0,0,0,0]` is shown as /*$bit()*/.

`$DATA` returns 1 for a compressed binary string variable, including an all-zeros bitstring, such as `[0,0,0,0,0,0,0]`. `$GET` returns the empty string for a compressed binary string, regardless of its value; `$GET` also returns the empty string for an undefined variable.

**Examples**

Note in the following examples that a bit that has not been set always has a value of 0.

The following example uses `SET $BIT` to create a compressed bitstring. It then invokes `$BIT` repeatedly to display the bits in the bitstring:

```objectscript
SET $BIT(a,1) = 0
SET $BIT(a,2) = 0
SET $BIT(a,3) = 1
SET $BIT(a,4) = 0
SET $BIT(a,5) = 0
SET $BIT(a,6) = 1
SET $BIT(a,7) = 1
SET $BIT(a,8) = 0
// Test single bits within the bitstring
WRITE "bit #2 value: ", $BIT(a,2),!
WRITE "bit #7 value: ", $BIT(a,7),!
WRITE "bit #8 value: ", $BIT(a,8),!
WRITE "bit #13 value: ", $BIT(a,13),!
// Write the bitstring
WRITE "bitstring value: "
FOR x=1:1:8 {WRITE $BIT(a,x) }
WRITE !!,"compressed bitstring: ", ZZDUMP a
```

Because Caché pads the bitstring with 0 bits to the specified position, the following example returns the exact same bitstring data value. However, note that because the #8 bit is not defined, the compressed bitstring `a` is not identical to the compressed bitstring `b`:

```objectscript
SET $BIT(b,3) = 1
SET $BIT(b,6) = 1
SET $BIT(b,7) = 1
// Test single bits within the bitstring
WRITE "bit #2 value: ", $BIT(b,2),!
WRITE "bit #7 value: ", $BIT(b,7),!
WRITE "bit #8 value: ", $BIT(b,8),!
WRITE "bit #13 value: ", $BIT(b,13),!
// Write the bitstring
WRITE "bitstring value: "
FOR x=1:1:8 {WRITE $BIT(b,x) }
WRITE !!,"compressed bitstring: ", ZZDUMP b
```

For this reason, it is a recommended programming practice to always explicitly set the highest defined bit in the bitstring, even when its assigned value is 0.

You can use a `FOR expr comma-separated list` to set multiple bits, as shown in the following example:
FOR i=3,6,7 { SET $BIT(b,i) = 1 }  
// Test single bits within the bitstring  
WRITE "bit #2 value: ",$BIT(b,2),!  
WRITE "bit #7 value: ",$BIT(b,7),!  
WRITE "bit #8 value: ",$BIT(b,8),!  
WRITE "bit #13 value: ",$BIT(b,13),!  
// Write the bitstring  
WRITE "bitstring value: "  
FOR x=1:1:8 {WRITE $BIT(b,x) }  
WRITE !!,"compressed bitstring: "  
ZZDUMP b

In the following example, successive invocations of $BIT return the bits of the bitstring generated by $FACTOR:

FOR i=1:1:32 {WRITE $BIT($FACTOR(2*31-1),i) }

The following example returns a random 16-bit bitstring:

SET x=$RANDOM(65536)  
FOR i=1:1:16 {WRITE $BIT($FACTOR(x),i) }

General Information on Bitstring Functions

Bitstring functions manipulate encoded bit-based data. Although a bitstring can be used with any ObjectScript command or function, it is generally meaningful only within the context of the bit functions.

The $BIT bitstring functions perform atomic operations. Therefore, no locking is required when performing bitstring operations.

The $BIT bitstring functions perform internal compression of bitstrings. Therefore, the actual data length of a bitstring and its physical space allocation may differ. $BIT bitstring functions use the data length of bitstrings. In most circumstances, the physical space allocation should be invisible to the user. bitstring compression is invisible to users of the $BIT functions.

However, because this compressed binary representation is optimized for each bitstring, one cannot assume that two “identical” bitstrings (which were created differently) have identical internal representations. Caché selects from four separate bitstring internal representations to optimize for both sparse bitstrings and non-sparse bitstrings. Therefore, while matching operations on individual bits yield predictable results, comparisons of entire bitstrings may not.

$BIT bitstring functions support a maximum bitstring length of 262,104 bits (32763 x 8) for Caché. (Unlike with certain InterSystems legacy products, it is not an error in Caché to perform an operation on a bit that is beyond the bitstring length.) However, it is strongly recommended for performance reasons that you divide long bitstrings into chunks of less than 65,280 bits. This is the maximum number of bits that can fit in a single 8KB database block.

Bits in a bitstring are numbered with the first (leftmost) bit as position 1. All bitstring comparisons are performed left-to-right.

In the examples, bitstrings are shown within matching square brackets ([...]), with the bits delimited by commas. For example, a bitstring of four 1 bits is shown as [1,1,1,1], with the least significant bits to the right.

In all the bitstring functions, variables named bitstring are bitstrings that can be specified as values, variables, or expressions.

$BIT Functions and $ZBIT Functions

The $BIT functions replace the earlier $ZBIT functions. New code should only use the $BIT functions; the $ZBIT functions will continue to be supported for legacy applications. The $BIT functions and the $ZBIT functions are incompatible; $BIT functions compress bitstrings, $ZBIT functions do not. Therefore, the two types of bitstring functions should not be used on the same bitstring. Attempting to do so results in an <INVALID BIT STRING> error.

You can use the $SYSTEM.Bit.ZBitToBit() method to convert a $ZBIT-format bit string to a $BIT-format bit string. This is shown in the following example:
ZBitSet
SET zb=$ZBITSTR(4,1) /* the $ZBIT string 1111 */
SET zbnew=$ZBITSET(zb,2,0) /* the $ZBIT string 1011 */
ZBitToBit
SET bb=$SYSTEM.Bit.ZBitToBit(zbnew)
WRITE !,$BIT(bb,1),$BIT(bb,2),$BIT(bb,3),$BIT(bb,4)

See Also

• $BITCOUNT function
• $BITFIND function
• $BITLOGIC function
• $FACTOR function
• $ZBOOLEAN function
$BITCOUNT

Returns the number of bits in a bitstring.

$BITCOUNT(bitstring,bitvalue)

**Parameters**

<table>
<thead>
<tr>
<th>bitstring</th>
<th>An expression that evaluates to a bitstring. Can be a variable of any type, $FACTOR, a user-defined function, or an oref.prop, ..prop, or i%prop property reference.</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitvalue</td>
<td>Optional — The value (0 or 1) to count within the bitstring.</td>
</tr>
</tbody>
</table>

**Description**

The $BITCOUNT function counts the number of bits within a bitstring. A bitstring is an encoded string which is interpreted by the system as a series of bits. You can create a bitstring using $BIT or $BITLOGIC.

$BITCOUNT(bitstring) returns the number of bits in *bitstring*.

$BITCOUNT(bitstring, bitvalue) returns the number of bits of type *bitvalue* (0 or 1) in *bitstring*.

The maximum bitstring length is 262,104 bits (32763 x 8).

Specifying a *bitstring* value that is not a Caché encoded bitstring generates an <INVALID BIT STRING> error. For further information, refer to General Information on Bitstring Functions.

**Examples**

If *bitstrings* = [0,0,1,1,0], then the result of $BITCOUNT(bitstring) is 5:

```objectscript
    SET $BIT(a,1) = 0
    SET $BIT(a,2) = 0
    SET $BIT(a,3) = 1
    SET $BIT(a,4) = 1
    SET $BIT(a,5) = 0
    WRITE !,$BITCOUNT(a)
```

If *bitstring* = [0,0,1,1,0], then the result of $BITCOUNT(bitstring,0) would be 3.

```objectscript
    SET $BIT(a,1) = 0
    SET $BIT(a,2) = 0
    SET $BIT(a,3) = 1
    SET $BIT(a,4) = 1
    SET $BIT(a,5) = 0
    WRITE !,"number of zero bits: ",$BITCOUNT(a,0)
    WRITE !,"number of one bits: ",$BITCOUNT(a,1)
```

The following example returns the number of 1 bits in a random 16-bit bitstring generated by $FACTOR:

```objectscript
    SET x=$RANDOM(65536)
    FOR i=1:1:16 {WRITE $BIT($FACTOR(x),i) }
    WRITE !,"Number of 1 bits= ",$BITCOUNT($FACTOR(x),1)
```

**See Also**

- $BIT function
- $BITFIND function
- $BITLOGIC function
- $FACTOR function
ObjectScript Functions

- **SZBOOLEAN** function
$BITFIND

Returns the position of the specified bit value within a bitstring.

$BITFIND(bitstring, bitvalue, position, direction)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitstring</td>
<td>An expression that evaluates to a bitstring. Can be a variable of any type, $FACTOR, a user-defined function, or an oref.prop, ..prop, or i%prop property reference.</td>
</tr>
<tr>
<td>bitvalue</td>
<td>The value (0 or 1) to search for within the bitstring.</td>
</tr>
<tr>
<td>position</td>
<td>Optional — The bit position from which the search begins, specified as a positive integer. Bit positions are counted from 1 from the beginning of the bit string. Search is inclusive of this position. A position value of 0 is treated as specifying position 1.</td>
</tr>
<tr>
<td>direction</td>
<td>Optional — A direction flag. Available values are 1 and -1. 1 = Search forward (left to right) from the beginning of the bitstring (or from position) towards the end (this is the default). -1 = Search backward from the end of the bitstring (or from position) towards the beginning.</td>
</tr>
</tbody>
</table>

**Description**

$BITFIND(bitstring, bitvalue) returns the position of the first occurrence of the specified bitvalue (0 or 1) within the bitstring bitstring. Bit positions are counted from 1.

$BITFIND(bitstring, bitvalue, position) returns the position of the first occurrence at or after position of the specified bitvalue in bitstring.

If the desired bit value is not found, or if position (searching forward) is greater than the number of bits within the bitstring, the return value is 0. If the specified bitstring is an undefined variable, the return value is 0. If the specified bitstring is not a valid bitstring, an <INVALID BIT STRING> error is issued.

There is also general information on bitstring functions available.

**Examples**

If bitstring = [0,0,1,1,0], then the result of $BITFIND(bitstring,1) would be 3:

```ObjectScript
// Set a to [0,0,1,1,0]
SET $BIT(a,1) = 0
SET $BIT(a,2) = 0
SET $BIT(a,3) = 1
SET $BIT(a,4) = 1
SET $BIT(a,5) = 0
// Find first 1 bit within a
WRITE !,$BITFIND(a,1)
```

If bitstring = [0,0,1,1,0], when searching from position 3, the first bit of value 1 is bit position 3 (because search is inclusive of the position bit) and the first bit of value 0 is bit position 5:

```ObjectScript
// Set a to [0,0,1,1,0]
SET $BIT(a,1) = 0
SET $BIT(a,2) = 0
SET $BIT(a,3) = 1
SET $BIT(a,4) = 1
SET $BIT(a,5) = 0
// Find first 1 bit from position 3
WRITE !,"found a 1 at bit position: ",$BITFIND(a,1,3)
// Find first 0 bit from position 3
WRITE !,"found a 0 at bit position: ",$BITFIND(a,0,3)
```
If $bitstring = [0,0,1,1,0]$, when searching backwards from $position$ 99, the first bit of value 1 is bit position 4 and the first bit of value 0 is bit position 5:

```objectscript
// Set a to [0,0,1,1,0]
SET $BIT(a,1) = 0
SET $BIT(a,2) = 0
SET $BIT(a,3) = 1
SET $BIT(a,4) = 1
SET $BIT(a,5) = 0
WRITE !,"found a 1 at bit position: ",$BITFIND(a,1,99,-1)
WRITE !,"found a 0 at bit position: ",$BITFIND(a,0,99,-1)
```

The following example returns a list of all of the 1 bit positions and a list of all of the 0 bit positions:

```objectscript
// Set a to [0,0,1,1,0]
SET $BIT(a,1) = 0
SET $BIT(a,2) = 0
SET $BIT(a,3) = 1
SET $BIT(a,4) = 1
SET $BIT(a,5) = 0
SET pos=0
WRITE !,"Bit positions with value 1: 
FOR  { SET pos=$BITFIND(a,1,pos+1) QUIT:'pos  WRITE pos", " }
WRITE !,"Bit positions with value 0: 
FOR  { SET pos=$BITFIND(a,0,pos+1) QUIT:'pos  WRITE pos", " }
```

The following example returns the position of the first 1 bit in a random 16-bit bitstring generated by $FACTOR:

```objectscript
SET x=$RANDOM(65536)
FOR i=1:1:16 {WRITE $BIT($FACTOR(x),i) }
WRITE !,"The first 1 bit is at position ",$BITFIND($FACTOR(x),1)
```

See Also

- $BIT function
- $BITCOUNT function
- $BITLOGIC function
- $FACTOR function
- $ZBOOLEAN function
$BITLOGIC

Performs bit-wise operations on bitstrings.

$BITLOGIC(bitstring_expression, length)

**Parameters**

| bitstring_expression | A logical expression consisting of one or more bitstring variables and the logical operators &, |, ^, and ~. A bitstring can be specified as a local variable, a process-private global, a global, an object property, or the constant "". The null string ("") has a bitstring length of 0. A bitstring cannot be specified using a function (such as $FACTOR) that returns a bitstring. |
|---------------------|------------------------------------------------------------------------------------------------|
| length              | Optional — The length, in bits, of the resulting bitstring. If *length* is not specified it defaults to the length of the longest bitstring in *bitstring_expression*. |

**Description**

$BITLOGIC evaluates a bit-wise operation on one or more bitstring values, as specified by *bitstring_expression*, and returns the resulting bitstring.

A bitstring is an encoded (compressed) string which is interpreted as a series of bits. Only bitstrings created using $BIT, $FACTOR, or $BITLOGIC, or the null string (""), should be supplied to the $BITLOGIC function. Typically, bitstrings are used for index operations. Refer to general information on bitstring functions in $BIT for further details.

$BITLOGIC is incompatible with any of the legacy $ZBIT bitstring functions, and should not be used in combination with them. $BITLOGIC and $ZBOOLEAN use different data formats. The results of one cannot be used as input to the other.

**Bitstring Optimization**

The most basic $BITLOGIC operation is $BITLOGIC(a). Seemingly, this operation does not do anything: bitstring a is input and the same bitstring a is output. However, $BITLOGIC performs bitstring compression which it optimizes by selecting from several compression algorithms. Therefore, if bitstring a has undergone substantial changes since its creation, passing it through $BITLOGIC can result in re-optimization of the bitstring. Refer to $BIT for further details.

For example, following a large number of delete operations an index bitstring may have become a sparse bitstring, consisting wholly or mainly of zeros. Passing this index bitstring through $BITLOGIC may result in substantial performance improvements.

**Bitstring Logical Operators**

$BITLOGIC can evaluate only the bitstring operators listed in the following table:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>XOR (exclusive OR)</td>
</tr>
<tr>
<td>~</td>
<td>NOT (one’s complement)</td>
</tr>
</tbody>
</table>
The `bitstring_expression` can contain a single bitstring (¬A), two bitstrings (A&B), or more than two bitstrings (A&B|C), up to the current maximum of 31 bitstrings. Evaluation is performed left-to-right. Logical operations may be grouped by parentheses within the `bitstring_expression`, following standard ObjectScript order of operations. If a variable used within `$BITLOGIC` is undefined, it is treated as a null string (“”).

$BITLOGIC treats a null string as a bitstring of indefinite length, in which all bits are set to 0’s.

Note: When $BITLOGIC is supplied more than two bitstring operands, it must create bitstring temporaries to hold the intermediate results. Under some extreme circumstances (many bitstrings and/or extremely large bitstrings), it can exhaust the space allocated to hold such temporaries. Bitstring pair operations do not have this limitation, and are thus preferable for large bitstring operations.

The NOT (¬) operator can be used as a unary operator (for example, ¬A), or can be used in combination with other operators (for example, A&¬B). It performs the one’s complement operation on a string, turning all 1’s to 0’s and all 0’s to 1’s. Multiple NOT operators can be used (for example, ~~~A).

**The length Argument**

If `length` is not specified, it defaults to the length of the longest bitstring in `bitstring_expression`.

If `length` is specified, it specifies the logical length of the resulting bitstring.

- If `length` is larger than one or more of the bitstrings in `bitstring_expression`, those bitstrings are zero-filled to that length before bitstring logic operations are performed.
- If `length` is smaller than one or more of the bitstrings in `bitstring_expression`, those bitstrings are truncated to that length before bitstring logic operations are performed.
- If `length` is 0, a bitstring of length 0 (a null string) is returned.

**Examples**

The following example creates some simple bitstrings and demonstrates the use of `$BITLOGIC` on them:

```objectscript
// Set a to [1,1]
SET $BIT(a,1) = 1
SET $BIT(a,2) = 1
// Set b to [0,1]
SET $BIT(b,1) = 0
SET $BIT(b,2) = 1
WRITE !,"bitstring a=",$BIT(a,1),$BIT(a,2)
WRITE !,"bitstring b=",$BIT(b,1),$BIT(b,2)
SET c = $BITLOGIC(~b)
WRITE !,"The one's complement of b=",$BIT(c,1),$BIT(c,2)
// Find the intersection (AND) of a and b
WRITE !,"The AND of a and b=",$BIT(c,1),$BIT(c,2)
SET c = $BITLOGIC(a&~b)   // c should be [1,0]
// Find the union (OR) of a and b
WRITE !,"The OR of a and b=",$BIT(c,1),$BIT(c,2)
SET c = $BITLOGIC(a|b)   // c should be [1,1]";
SET c = $BITLOGIC(a&b)   // c should be [0,0]
WRITE !,"The XOR of a and b=",$BIT(c,1),$BIT(c,2)
QUIT
```

The following example shows the results of specifying a `length` greater than the input bitstring. The string is zero-filled before the logic operation is performed.

```objectscript
// Set a to [1,1]
SET $BIT(a,1) = 1
SET $BIT(a,2) = 1
WRITE !,"bitstring a=",$BIT(a,1),$BIT(a,2)
SET c = $BITLOGIC(~a,7)   // c should be [0,0,0,0,0,0,0,0]
WRITE !,"~a (length 7)=" $BIT(c,1),$BIT(c,2),$BIT(c,3),$BIT(c,4),$BIT(c,5),$BIT(c,6),$BIT(c,7),$BIT(c,8)
QUIT
```
Here the one’s complement (−) of $11$ is $0011111$. Bits 3 through 7 were set to zero before the − operation was performed. This example also displays an eighth bit, which is beyond the specified string length and thus unaffected by the $\texttt{$BITLOGIC$}$ operation. It is, of course, displayed as 0.

The following example shows the results of specifying a length less than the input bitstring. The bitstring is truncated to the specified length before logical operations are performed. All bits beyond the specified length default to 0.

```objectscript
// Set a to [1,1,1]
SET $BIT(a,1) = 1
SET $BIT(a,2) = 1
SET $BIT(a,3) = 1
WRITE 1,*"bitstring a=",$BIT(a,1),$BIT(a,2),$BIT(a,3)
SET c = $BITLOGIC(a,2)
WRITE 1," a (length 2)="
WRITE $BIT(c,1),$BIT(c,2),$BIT(c,3),$BIT(c,4)
SET c = $BITLOGIC(~a,2)
WRITE 1,"~a (length 2)="
WRITE $BIT(c,1),$BIT(c,2),$BIT(c,3),$BIT(c,4)
```

The following example shows that when length is not specified, it defaults to the length of the longest bitstring. Shorter bitstrings are zero-filled before the logical operation is performed.

```objectscript
// Set a to [1,1,1]
SET $BIT(a,1) = 1
SET $BIT(a,2) = 1
SET $BIT(a,3) = 1
// Set b to [1,1]
SET $BIT(b,1) = 1
SET $BIT(b,2) = 1
SET c = $BITLOGIC(a&~b)
WRITE 1," a&~b="
WRITE $BIT(c,1),$BIT(c,2),$BIT(c,3)
SET c = $BITLOGIC(a&~b,3)
WRITE 1,"a&~b,3="
WRITE $BIT(c,1),$BIT(c,2),$BIT(c,3)
```

Here the two $\texttt{$BITLOGIC$}$ operations (with and without a length argument) both return the same value: 001.

**See Also**

- $\texttt{$BIT$}$ function
- $\texttt{$BITCOUNT$}$ function
- $\texttt{$BITFIND$}$ function
- $\texttt{$ZBOOLEAN$}$ function
- Operators in *Using Caché ObjectScript*
$CASE

Compares expressions and returns the value of the first matching case.

$CASE(target,case:value,case:value,...,:default)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>A literal or expression the value of which is to be matched against cases.</td>
</tr>
<tr>
<td>case</td>
<td>A literal or expression the value of which is to be matched with the results of the evaluation of target.</td>
</tr>
<tr>
<td>value</td>
<td>The value to be returned upon a successful match of the corresponding case.</td>
</tr>
<tr>
<td>default</td>
<td>Optional — The value to be returned if no case matches target.</td>
</tr>
</tbody>
</table>

**Description**

The $CASE function compares target to a list of cases (literals or expressions), and returns the value of the first matching case. An unlimited number of case:value pairs can be specified. Cases are matched in the order specified (left-to-right); matching stops when the first exact match is encountered.

If there is no matching case, default is returned. If there is no matching case and no default is specified, Caché issues an <ILLEGAL VALUE> error.

Caché permits specifying $CASE with no case:value pairs. It always returns the default value, regardless of the target value.

**Parameters**

**target**

$CASE evaluates this expression once, then matches the result to each case in left-to-right order.

**case**

A case can be a literal or an expression; matching of literals is substantially more efficient than matching expressions, because literals can be evaluated at compile time. Each case must be paired with a value. An unlimited number of case and value pairs may be specified.

**value**

A value can be a literal or an expression. Using $CASE as an argument of a GOTO command or a DO command restricts value as follows:

- When using a $CASE statement with a GOTO command, each value must be a valid line label. It cannot be an expression.
- When using a $CASE statement with a DO command, each value must be a valid DO argument. These DO arguments can include parameters.

**default**

The default is specified like a case:value pair, except that there is no case specified between the comma (used to separate pairs) and the colon (used to pair items). The default is optional. If specified, it is always the final parameter in a $CASE function. The default value follows the same GOTO and DO restrictions as the value parameter.
If there is no matching `case` and no `default` is specified, Caché issues an `<ILLEGAL VALUE>` error.

### Examples

The following example takes a day-of-week number and returns the corresponding day name. Note that a default value “entry error” is provided:

```objectscript
SET daynum=$ZDATE($HOROLOG,10)
WRITE $CASE(daynum,
  1:"Monday",2:"Tuesday",3:"Wednesday",
  4:"Thursday",5:"Friday",
  6:"Saturday",0:"Sunday",:"entry error")
```

The following example takes as input the number of bases achieved by a baseball batter and writes out the appropriate baseball term:

```objectscript
SET hit=$RANDOM(5)
SET atbat=$CASE(hit,1:"single",2:"double",3:"triple",4:"home run",:"strike out")
WRITE hit," = ",atbat
```

The following example uses `$CASE` as the `DO` command argument. It calls the routine appropriate for the `exp` exponent value:

```objectscript
Start ; Raise an integer to a randomly-selected power.
SET exp=$RANDOM(6)
SET num=4
DO $CASE(exp,0:NoMul(),2:Square(num),3:Cube(num),:Exponent(num,exp))
WRITE !,num," = ",result,!
RETURN
Square(n)
SET result=n*n
SET result="Squared = ":result
RETURN
Cube(n)
SET result=n*n*n
SET result="Cubed = ":result
RETURN
Exponent(n,x)
SET result=n
FOR i=1:1:x-1 { SET result=result*n }
SET result="exponent ":x=" ":result
RETURN
NoMul()
SET result="multiply by zero"
RETURN
```

The following example tests whether the character input is a letter or some other character:

```objectscript
READ "Input a letter: ",x
SET chartype=$CASE(x?1A,1:"letter",:"other")
WRITE chartype
```

The following example uses `$CASE` to determine which subscripted variable to return:

```objectscript
SET dabbrv="W"
SET wday(1)="Sunday",wday(2)="Monday",wday(3)="Tuesday",
  wday(4)="Wednesday",wday(5)="Thursday",wday(6)="Friday",wday(7)="Saturday"
```

The following example specifies no `case:value` pairs. It return the `default` string “not defined”:

```objectscript
SET dummy=3
WRITE $CASE(dummy,:"not defined")
```

### See Also

- `DO` command
- `GOTO` command
- `IF` command
• $SELECT function
$CHAR

Converts the integer value of an expression to the corresponding ASCII or Unicode character.

$CHAR(expression,...)
$C(expression,...)

**Parameter**

| expression | The integer value to be converted. |

**Description**

$CHAR returns the character that corresponds to the decimal (base-10) integer value specified by `expression`. This character can be an 8-bit (ASCII) character, or a 16-bit (Unicode) character. For 8-bit characters, the value in `expression` must evaluate to a positive integer in the range 0 to 255. For 16-bit characters, specify integers in the range 256 through 65535 (hex FFFF). Values larger than 65535 return an empty string. Values from 65536 (hex 10000) through 1114111 (hex 10FFFF) are used to represent Unicode surrogate pairs; these characters can be returned using $WCHAR. On an 8-bit Caché installation, $CHAR returns the empty string for `expression` values greater than 255.

You can specify `expression` as a comma-separated list, in which case $CHAR returns the corresponding character for each expression in the list.

The $ASCII function is the inverse of $CHAR.

**Parameter**

**expression**

The expression can be an integer value, the name of a variable that contains an integer value, or any valid ObjectScript expression that evaluates to an integer value. To return characters for multiple integer values, specify a comma-separated list of expressions.

You can use the $ZHEX function to specify a character using a hexadecimal character code, rather than a decimal (base-10) character code. In the following example, both $CHAR statements return the Greek letter pi:

```
WRITE $CHAR(960),!
WRITE $CHAR($ZHEX("3C0"))
```

**Examples**

The following example uses $CHAR in a **FOR** loop to output the characters for all ASCII codes in the range 65 to 90. These are the uppercase alphabetic characters.

```
FOR i=65:1:90 {
    WRITE !,$CHAR(i) }
```

The following example uses $CHAR in a **FOR** loop to output the Japanese Hiragana characters:

```
IF $SYSTEM.Version.IsUnicode() {
    FOR i=12353:1:12435 {
        WRITE !,$CHAR(i) }
} ELSE {WRITE "This example requires a Unicode version of Caché"}
```

The following two examples show the use of multiple expression values. The first returns “AB” and the second returns “AaBbCcDdEeFfGgHhIjJkKk”:

```
WRITE $CHAR(65,66),!
FOR i=65:1:75 {
    WRITE $CHAR(i,i+32) }
```
The following example shows the use of a multibyte character. The character corresponding to the integer 960 is the symbol for pi:

```
IF $SYSTEM.Version.IsUnicode() {
WRITE $CHAR(960) 
} ELSE {WRITE "This example requires a Unicode version of Caché"
}
```

### Notes

#### $CHAR with the WRITE Command

When you use $CHAR to write characters with the WRITE command, the output characters reset the positions of the special variables $X and $Y. This is true even for the NULL character (ASCII 0), which is not the same as a null string ("""). As a rule, you should use $CHAR with caution when writing nonprinting characters, because such characters may produce unpredictable cursor positioning and screen behavior.

#### $CHAR and %List Structures

Because a %List structure (%Library.List) is an encoded string using non-printing characters, certain $CHAR values result in a %List structure containing a single element. The $CHAR combinations that return a %List structure are as follows:

- $CHAR(1) returns an empty list: $lb().
- $CHAR(1,1) returns a two-element empty list: $lb(,).
- $CHAR(2,1), $CHAR(2,2), or $CHAR(2,12) returns a list containing the empty string: $lb(" ").
- $CHAR(2,4) returns $lb(0).
- $CHAR(2,5) returns $lb(-1).
- $CHAR(2,8) or $CHAR(2,9) returns $lb($double(0)).

$CHAR combinations that involve more than two characters and result in a single-element list have the following syntax:

```
$CHAR(count,flag,string)
```

- **count** is the total number of characters. For example, $CHAR(5,1,65,66,67) or $CHAR(5,1) _"ABC"_.
- **flag** is an integer specifying how string should be represented. Valid flag values include 1, 2, 4, 5, 6, 7, 8, 9, 12, and 13. These flag interpretations have nothing to do with the usual ASCII interpretation of this non-print character.
  - flag=1, flag=12, and flag=13 return the literal string value as the list element.
  - flag=2 is only valid if count is an even number. It returns a list element containing one or more wide Unicode characters derived from string, often one or more Chinese characters.
  - flag=4 returns the positive ASCII numeric code for the character(s) as the list element. flag=4 cannot be used when count>10.
  - flag=5 returns a negative integer ASCII numeric code for the character(s) as the list element. flag=5 cannot be used when count>10.
  - flag=6 returns a positive integer derived from string as the list element:
    - $CHAR(3,6,n) always returns $lb(0).
    - $CHAR(count,6,string) when count > 3 returns a (usually) large positive integer derived from the ASCII numeric value. The number of trailing zeros corresponds to the ASCII value of the first character in string, the leading numeric value to the ASCII value of the second character in string. For example, $CHAR(4,6,0,7) returns $lb(7); $CHAR(4,6,3,7) returns $lb(7000).
- flag=6 cannot be used when count>11.
• flag=7 returns a negative integer derived from string as the list element:
  – $CHAR(3,7,n)$ returns a negative number with the number of zeros corresponding to the value of n: $0 = -1$, $1 = -10$, $2 = -100$, $3 = -1000$, etc.
  – $CHAR(count,7,string)$ when $count > 3$ returns a (usually) large negative integer. The number of trailing zeros corresponds to the ASCII value of the first character in string.

  flag=7 cannot be used when $count > 11$.
• flag=8 returns $DOUBLE(x)$ where x is a small number. flag=8 cannot be used when $count > 6$.
• flag=9 returns $DOUBLE(x)$ where x is a large number. flag=9 cannot be used when $count > 10$.

string is a numeric or string of $count – 2$ characters. For example, a string of three characters can be represented as either $CHAR(5,flag,65,66,67)$ or $CHAR(5,flag)_"ABC"$. The string value becomes the list element, its value represented as specified by flag.

For further details, refer to $LISTBUILD$ and $LISTVALID$.

**Numeric Values in $CHAR$ Arguments**

You can use signed numeric values for expression. Caché ignores negative numbers and only evaluates positive or unsigned numbers. In the following example, the $CHAR$ with signed integers returns only the first and third expression, ignoring the second expression, which is a negative integer.

```
WRITE !,$CHAR(65,66,67)
WRITE !,$CHAR(+65,-66,67)
```

ABC
AC

You can use floating point numeric values for expression. Caché ignores the fractional portion of the argument and only considers the integer portion. In the following example, $CHAR$ ignores the fractional portion of the number and produces the character represented by character code 65, an uppercase A.

```
WRITE $CHAR(65.5)
```

**Unicode Support**

On a Unicode installation of Caché, $CHAR$ supports Unicode characters when represented by decimal (base-10) integers. On an 8-bit installation of Caché, $CHAR$ returns the empty string for integers greater than 255.

The Unicode value for a character is usually expressed as a 4-digit number in hexadecimal notation, using the digits 0-9 and the letters A-F. However, standard functions in the ObjectScript language generally identify characters according to ASCII codes, which are decimal values, not hexadecimal.

Hence, the $CHAR$ function supports Unicode encoding by returning a character based on the decimal Unicode value that was input, not the more standard hexadecimal value. You can specify a hexadecimal Unicode value using the $ZHSEX$ function using quotes, as follows $CHAR($ZHSEX("hexnum"))$. You can also use $ZHSEX$ without quotes to convert a decimal number to hexadecimal, as follows: $hexnum = ZHSEX(decnum)$.

For further details on Caché Unicode support, refer to Unicode in *Using Caché ObjectScript*.

**Surrogate Pairs**

$CHAR$ does not recognize surrogate pairs. Surrogate pairs are used to represent some Chinese characters and to support the Japanese JIS2004 standard. You can use the $SWISWIDE$ function to determine if a string contains a surrogate pair. The $SWCHAR$ function recognizes and correctly parses surrogate pairs. $CHAR$ and $SWCHAR$ are otherwise identical. However, because $CHAR$ is generally faster than $SWCHAR$, $CHAR$ is preferable for all cases where a surrogate pair is not likely to be encountered.
Note:  $WCHAR should not be confused with $ZWCHAR, which always parses characters in pairs.

Functions Related to $CHAR

The $ASCII function is the inverse of $CHAR. You can use it to convert a character to its equivalent numeric value. $ASCII converts all characters, including Unicode characters. In addition, all Caché platforms support the related functions, $ZLCHAR and $ZWCHAR. They are similar to $CHAR, but operate on a word (two bytes) or a long word (four bytes). You can use $ZISWIDE to determine if there are any multibyte (“wide”) characters in the expression of $CHAR.

See Also

• READ command
• WRITE command
• $ASCII function
• $WCHAR function
• $WISWIDE function
• $ZHEX function
• $ZLCHAR function
• $ZWCHAR function
• $X special variable
• $Y special variable
$CLASSMETHOD

Executes a named class method in the designated class.

$CLASSMETHOD(classname, methodname, arg1, arg2, arg3, ... )

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>classname</strong></td>
<td><em>Optional</em> — An expression that evaluates to a string. The content of the string must match exactly the name of an existing, accessible, previously compiled class. In the case of references to Caché classes, the name may be either in its canonical form (%Library.String), or its abbreviated form (%String). If <code>classname</code> is omitted, the current class context is used. (You can use <code>$THIS</code> to determine the current class context.) Note that when <code>classname</code> is omitted the placeholder comma must be specified.</td>
</tr>
<tr>
<td><strong>methodname</strong></td>
<td>An expression which evaluates to a string. The value of the string must match the name of an existing class method in the class identified by <code>classname</code>.</td>
</tr>
<tr>
<td><strong>arg1, arg2, arg3, ...</strong></td>
<td><em>Optional</em> — A series of expressions to be substituted sequentially for the arguments to the designated method. The values of the expressions can be of any type. It is the responsibility of the implementor to make sure that the type of the supplied expressions match what the method expects, and have values within the bounds declared. (If the specified method expects no arguments then no arguments beyond the <code>methodname</code> need be given in the function invocation. If the method requires arguments, the rules that govern what must be supplied are those of the target method.)</td>
</tr>
</tbody>
</table>

Description

$CLASSMETHOD permits an ObjectScript program to invoke an arbitrary class method in an arbitrary class. Both the class name and the method name may be computed at runtime or supplied as string constants. To invoke an instance method rather than a class method, use the `$METHOD` function.

If the method takes arguments, they are supplied by the list of arguments that follow the method name. A maximum of 255 argument values may be passed to the method.

The invocation of $CLASSMETHOD as a function or a procedure determines the invocation of the target method. You can invoke $CLASSMETHOD using the JOB command or the DO command, discarding the return value.

An attempt to invoke a nonexistent class results in a `<CLASS DOES NOT EXIST>` error, followed by the current namespace name and the specified class name. For example, attempting to invoke the nonexistent `classname` “Fred” results in the error `<CLASS DOES NOT EXIST> *User.Fred. Specifying the empty string for `classname` results in `<CLASS DOES NOT EXIST> *(No name).` An attempt to invoke a nonexistent class method results in a `<METHOD DOES NOT EXIST>` error.

Examples

The following example shows $CLASSMETHOD used as a function:

```objectscript
SET classname = "%Dictionary.ClassDefinition"
SET classmethodname = "NormalizeClassname"
SET singleargument = "%String"
WRITE $CLASSMETHOD(classname, classmethodname, singleargument),!
```
It returns `%Library.String.

The following example shows `$CLASSMETHOD` with two parameters:

```ObjectScript
WRITE $CLASSMETHOD("%Library.Persistent","%PackageName"),!
WRITE $CLASSMETHOD("%Library.Persistent","%ClassName")
```

These calls return `%Library` and `%Persistent.

The following example uses `$CLASSMETHOD` to execute a Dynamic SQL query:

```ObjectScript
SET q1="SELECT Age,Name FROM Sample.Person 
SET q2="WHERE Age > ? AND Age < ? 
SET q3="ORDER by Age"
SET myquery=q1_q2_q3
SET rset=$CLASSMETHOD("%SQL.Statement","%ExecDirect",,myquery,12,20)
DO rset.%Display()
WRITE !,"Teenagers in Sample.Person"
```

See Also

- `$CLASSNAME` function
- `$METHOD` function
- `$PARAMETER` function
- `$PROPERTY` function
- `$THIS` special variable
$CLASSNAME

Returns the name of a class.

$CLASSNAME(n)

Parameter

| n  | Optional — An object reference (OREF) to an class instance. If omitted, the class name of the current class is returned. |

Description

$CLASSNAME returns the name of a class. Commonly, it takes an object reference (OREF) and returns the corresponding class name. $CLASSNAME with no argument returns the name of the current class. $CLASSNAME always returns the full class name (for example, %SQL.Statement), not the short version of the class name omitting the package name (for example, Statement).

$CLASSNAME is functionally equivalent to the %ClassName(1) method of the %Library.Base superclass. The $CLASSNAME function gives better performance than the %ClassName(1) method for returning the full class name. To return the short version of the class name, you can use either %ClassName() or %ClassName(0).

For information on OREFs, see “OREF Basics” in Using Caché Objects.

Examples

The following example creates an instance of a class. $CLASSNAME takes the instance OREF and returns the corresponding class name:

```objectscript
SET dynoref = ##class(%SQL.Statement).%New()
WRITE "instance class name: ", $CLASSNAME(dynoref)
```

In the following example, $CLASSNAME with no parameter returns the class name of the current class context. In this case, it is the DocBook.Utils class. This is the same class name contained in the $THIS special variable:

```objectscript
WRITE "class context: ", $CLASSNAME(),!
WRITE "class context: ", $THIS
```

The following example shows that the $CLASSNAME function and the %ClassName(1) method return the same values. It also shows use of the %ClassName() method (with no argument or with a 0 argument) to return the short version of the class name:

```objectscript
CurrentClass
WRITE "current full class name: ", $CLASSNAME(),!
WRITE "current full class name: ", ..%ClassName(1),!
WRITE "current short class name: ", ..%ClassName(0),!
WRITE "current short class name: ", ..%ClassName(),!!
```

ClassInstance

```objectscript
SET x = ##class(%SQL.Statement).%New()
WRITE "oref full class name: ", $CLASSNAME(x),!
WRITE "oref full class name: ", x.%ClassName(1),!
WRITE "oref short class name: ", x.%ClassName(0),!
WRITE "oref short class name: ", x.%ClassName(),!
```

See Also

- $CLASSMETHOD function
- $METHOD function
- $PARAMETER function
• $PROPERTY function
• $THIS special variable
$COMPILE

Compiles source code, producing executable object code.

$$\textit{COMPILE(source,language,errors,object)}$$
$$\textit{COMPILE(source,language,errors,,rname)}$$

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>An array variable containing the source code to be compiled.</td>
</tr>
<tr>
<td>language</td>
<td>An integer flag specifying the programming language of the source code.</td>
</tr>
<tr>
<td>errors</td>
<td>A local variable that receives any errors that occur during compilation. This variable is a List structure, with one element for each error reported. Each error is itself a List structure, specifying error location and type (see below).</td>
</tr>
<tr>
<td>object</td>
<td>Optional — An array used to hold the compiled object code.</td>
</tr>
<tr>
<td>rname</td>
<td>Optional — (Second syntactic form only) a string specifying a routine name used to store the compiled object code in the ^rOBJ global.</td>
</tr>
</tbody>
</table>

**Description**

$COMPILE compiles source code and produces object code (the executable form of the routine). $COMPILE reports compilation errors, and can be used to check source code for compilation errors without actually producing object code. Before compiling, any macros in the source code must be processed by a preprocessor such as the ObjectScript macro preprocessor.

**Note:** Caché provides several powerful tools for compiling source code, Studio being one of them. Commonly, source code compilation is performed using these tools, rather than this $COMPILE function.

$COMPILE has two syntactic forms:

- The first $COMPILE syntax form returns the object code in the object array. It first kills the object variable. After the compilation the object array is set to the size of the compiled object code.
  
  The object array contains the object code in the same format as it would be in the ^rOBJ global. The object code in ^rOBJ can be replaced with the new object code by the command MERGE ^rOBJ(rname)=object(1). However, the MERGE command is not atomic when setting multiple nodes, so this operation could cause unpredictable results if another process is concurrently loading the same routine.

  If you omit the object parameter, the source code is compiled and checked for errors, but no object code is created.

- The second $COMPILE syntax form returns the object code directly into ^rOBJ(rname). The $COMPILE operation internally locks ^rOBJ(rname), preventing any other process from loading the routine object code until the new object code is completely stored.

  Commonly, the object argument is omitted with this syntactic form. If you specify the object parameter, the object variable is killed, but is not set. The other omitted arguments (represented by placeholder commas) are for internal use and should not be specified.

$COMPILE returns a status code as follows: 0 = no errors were detected and object code was created. 1 = errors were detected and object code was created. -1 = errors were detected and no object code was created.

When the ObjectScript compiler detects an error, it creates object code at that point which throws an error when that line is executed. When the Basic compilers detect an error they do not produce any object code.
To compile a class, use the \texttt{Compile()} method of the %SYSTEM.OBJ class.

**Parameters**

\texttt{source}

An array containing the source code to be compiled. The source can be in the format of an INT routine for ObjectScript, a BAS routine for Caché Basic, or an MVI routine for Caché MVBasic. The array element \texttt{source(0)} must contain the number of lines of source code, and each \texttt{source(n)} contains line number \texttt{n} of the source code. The source lines must be numbered consecutively from 1 through \texttt{n} with no omitted lines. The \texttt{source} parameter can be an unsubscripted local variable name, or a possibly subscripted global name.

If \texttt{source(0)} is undefined, the system generates an \texttt{<UNDEFINED>} error, regardless of the \texttt{%SYSTEM.Process.Undef() method setting.}

If a \texttt{source(0)} value is larger than the number of lines of source code, or a consecutive source code line is missing, the system generates an \texttt{<UNDEFINED>} error, followed by the name of the missing source code line. This behavior can be changed by setting the \texttt{%SYSTEM.Process.Undef()} method. These types of errors are shown in the following examples:

\begin{verbatim}
SET src(0)=4,src(1)="TestA ",src(2)=" WRITE 123",src(3)=" WRITE 456,!
SET stat=$COMPILE(src,0,errs,TestA)    /* generates <UNDEFINED> *src(4) */
SET src(0)=4,src(1)="TestA ",src(3)=" WRITE 123",src(4)=" WRITE 456,!
SET stat=$COMPILE(src,0,errs,TestA)    /* generates <UNDEFINED> *src(2) */
SET src(0)=3,src(1)="TestA ",src(3)=" WRITE 123",src(4)=" WRITE 456,!
SET stat=$COMPILE(src,0,errs,TestA)    /* generates <UNDEFINED> *src(2) */
\end{verbatim}

**language**

The language mode specifying the type of source to be compiled. Common values are:

- 0 = ObjectScript
- 9 = Caché Basic
- 11 = Caché MVBasic

Other values specify legacy ObjectScript modes and should be used only after consultation with InterSystems support.

**errors**

An unsubscripted local variable that is set to any errors detected by the compiler. Any existing value is killed. If no errors are detected, the variable is set to the empty string ("""). If errors are detected, the \texttt{errors} variable is set to a \texttt{$LIST} structure with one element for each error. Each error is itself a \texttt{$LIST} structure with the format $LISTBUILD(line,offset,errnum,text) where:

- \texttt{line} = the line number where the error was detected
- \texttt{offset} = the offset in the source line of the error
- \texttt{errnum} = an error number for the type of error
- \texttt{text} = text describing the error

The Basic compilers add two more elements to the list, with text describing the location of the error, and the source line itself.
An array that receives the object code output of the compiler. The object parameter can be an unsubscripted local variable name, or a possibly subscripted global name. The contents of the object array are described above.

The routine name that specifies where the object code should saved in the ^rOBJ subscripted global. $COMPILE kills any existing contents of ^rOBJ(rname) before saving the new object code.

Interrupting a Compile

You can issue a Ctrl-C or invoke the ^RESJOB utility to interrupt a compile in progress. These compile interrupts are supported for all language modes.

Examples

The following example compiles a four-line ObjectScript program using the first $COMPILE format:

```
SourceCode
SET src(0)=4
SET src(1)="TestA 
SET src(2)=WRITE "Hello 
SET src(3)=WRITE "World",!
SET src(4)=" QUIT"
CompileSource
SET stat=$COMPILE(src,0,errs,TestA)
IF stat=0 {WRITE "Compile successful"
ELSE {WRITE "status=",stat,
    WRITE "number of compile errors=",LISTLENGTH(errs) }
```

The following example compiles the same four-line ObjectScript program using the second $COMPILE format:

```
SourceCode
SET src(0)=4
SET src(1)="TestB 
SET src(2)=WRITE "Hello 
SET src(3)=WRITE "World",!
SET src(4)=" QUIT"
CompileSource
SET stat=$COMPILE(src,0,errs,,,,"TestB")
IF stat=0 {WRITE "Compile successful",!
    DO ^TestB }
ELSE {WRITE "status=",stat,
    WRITE "number of compile errors=",LISTLENGTH(errs) }
```

The following example performs compilation error checking on a seven-line ObjectScript program. Note that this $COMPILE only tests for errors; it does not provide a variable to receive the object code from a successful compile. In this example every line of source code contains an error; $COMPILE only returns the compile-time errors in lines 1, 3, 5, 6, and 7, not runtime errors such as a divide-by-zero error (line 2) or an undefined variable error (line 4):

```
SourceCode
SET src(0)=7
SET src(1)="?TestC 
SET src(2)=" SET a=2/0"
SET src(3)=" SET b=3+2"
SET src(4)=" SET c=xxx"
SET src(5)=" SET d=5"
SET src(6)=" SET 123="abc"
SET src(7)=" SET f=7"
CompileSource
SET stat=$COMPILE(src,0,errs)
IF stat {WRITE $LISTLENGTH(errs)," Compile Errors ",!
    FOR i=1:1:$LISTLENGTH(errs) {
        WRITE ",i,": "
        SET errn=$LIST(errs,i)
        FOR j=1:1:$LISTLENGTH(errn) {
            WRITE $LIST(errn,j)," 
        }
    }
} ELSE {WRITE "Compile successful",!
    WRITE "but no object code generated" }
See Also

- “ObjectScript Macros and the Macro Preprocessor” in Using Caché ObjectScript
- Using Studio
$DATA

Checks if a variable contains data.

$DATA(variable,target)  $D(variable,target)

Parameters

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>The variable whose status is to be checked. A local or global variable, subscripted or unsubscripted. The variable may be undefined. You cannot specify a simple object property reference as variable; you can specify a multidimensional property reference as variable with the syntax obj.property.</td>
</tr>
<tr>
<td>target</td>
<td>Optional — A variable into which $DATA returns the current value of variable.</td>
</tr>
</tbody>
</table>

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 subscripted array element. It can be a multidimensional object property; it cannot be a non-multidimensional object property.

The possible status values that may be returned are as follows:

<table>
<thead>
<tr>
<th>Status Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The variable is undefined. Any reference would cause an &lt;UNDEFINED&gt; error.</td>
</tr>
<tr>
<td>1</td>
<td>The variable exists and contains data, but has no descendants. Note that the null string (&quot;&quot;) qualifies as data.</td>
</tr>
<tr>
<td>10</td>
<td>The variable identifies an array element that has descendants (contains a downward pointer to another array element) but does not contain data. Any direct reference to such a variable will result in an &lt;UNDEFINED&gt; error. For example, if y(1) is defined, but y is not, $DATA(y) returns 10, set x=y will produce an &lt;UNDEFINED&gt; error.</td>
</tr>
<tr>
<td>11</td>
<td>The variable identifies an array element that has descendants (contains a downward pointer to another array element) and contains data. Variables of this type can be referenced in expressions.</td>
</tr>
</tbody>
</table>

You can use modulo 2 (#2) arithmetic to return a boolean value from $DATA: $DATA(var) #2 returns 0 for the undefined status codes (0 and 10), and returns 1 for the defined status codes (1 and 11).

Status values 1 and 11 indicate only the presence of data, not the type of data.

You can use the Undefined() method of the %SYSTEM.Process class to set behavior when encountering an undefined variable. For more information on <UNDEFINED> errors, refer to the $ZERROR special variable.

Parameters

variable

The variable that is being tested for the presence of data:

- variable can be a local variable, a global variable, or a process-private global (PPG) variable. It can be subscripted or unsubscripted.
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.

- `variable` can be a multidimensional object property; for example, `$.DATA(.Count)`. It cannot be a non-multidimensional object property. Attempting to use `$.DATA` on a non-multidimensional object property results in an `<OBJECT DISPATCH>` error.

`$.DATA` cannot return a data status value for a Proxy Object property. Caché instead issues a message that the specified property does not exist. This property access limitation is unique to the class `%ZEN.proxyObject`, which is defined in the `InterSystems Class Reference`.

- If `variable` is the `^$ROUTINE` structured system variable, the possible returned status values are 1 or 0.

**Target**

An optional parameter.Specify the name of a local variable, a process-private global, or a global. This `target` variable does not need to be defined. If `target` is specified, `$.DATA` writes the current data value of `variable` into `target`. If `variable` is undefined, the `target` value remains unchanged.

The `ZBREAK` command cannot specify the `target` parameter as a watchpoint.

**Examples**

This example writes a selected range of records from the `^client` array, a sparse array consisting of three levels. The first level contains the client’s name, the second the client’s address, and the third the client’s accounts, account numbers, and balances. A client can have up to four separate accounts. Because `^client` is a sparse array there may be undefined elements at any of the three levels. The contents for a typical record might appear as follows:

```
^client(5) John Jones
^client(5,1) 23 Bay Rd./Boston/MA 02049
^client(5,1,1) Checking/45673/1248.00
^client(5,1,2) Savings/27564/3270.00
^client(5,1,3) Reserve Credit/32456/125.00
^client(5,1,4) Loan/81263/460.00
```

The code below provides a separate subroutine to handle the output for each of the three array levels. It uses the `$.DATA` function at the start of each subroutine to test the current array element.

The `$.DATA=0` test in Level1, Level2, and Level3 tests whether the current array element is undefined. If TRUE, it causes the code to `QUIT` and revert to the previous level.

The `$.DATA=10` test in Level1 and Level2 tests whether the current array element contains a pointer to a subordinate element, but no data. If TRUE, it causes the code to write out a “No Data” message. The code then skips to the `FOR` loop processing for the next lower level. There is no `$.DATA=10` test in Level3 because there are no elements subordinate to this level.

The `WRITE` commands in Level2 and Level3 use the `$.PIECE` function to extract the appropriate information from the current array element.

```objectscript
Start Read !,"Output how many records: ",n
   Read !,"Start with record number: ",s
   For i=s:1:s+(n-1) {  
      If $.DATA("^client(i)) {  
         If $.DATA("^client(i,1))=10 {  
            Write !,"Name: No Data"
         } Else {  
            Write !,"Name: ",^client(i)
         }
         If $.DATA("^client(i,1,1)) {  
            If $.DATA("^client(i,1,1))=10 {  
               Write !,"Address: No Data"
            } Else {  
               Write !,"Address: ",$.PIECE("^client(i,1,1,"/",1)
               Write ", ",$.PIECE("^client(i,1,1,"/",2)
               Write ", ",$.PIECE("^client(i,1,1,"/",3)
```
When executed, this code might produce output similar to the following:

Output how many records: 3
Start with record number: 10
Name: Jane Smith
Address: 74 Hilltop Dr., Beverly, MA 01965
Account: Checking #: 34218 Balance: 876.72
Account: Reserve Credit #: 47821 Balance: 1200.00
Name: Thomas Brown
Address: 46 Huron Ave., Medford, MA 02019
Account: Checking #: 59363 Balance: 205.45
Account: Savings #: 41792 Balance: 1560.80
Account: Reserve Credit #: 64218 Balance: 125.52
Name: Sarah Copley
Address: No Data
Account: Checking #: 30021 Balance: 762.28

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:

```objectscript
SET obj = ##class(%ResultSet).%New("%SYS.Namespace:List")
DO obj.Execute()
WRITE $DATA(obj.Data,targ),!                // returns 0
SET targ="blank"
WHILE targ"="
DO obj.Next()
WRITE $DATA(obj.Data(targ)),!  // returns 10
WRITE ",",$DATA(obj.Data("Nsp"),targ),! // returns 1
IF targ"="
WRITE ",Namespace: ",targ,!
WRITE !,"Done!"
```

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 is not defined (Status Value = 0).

Subsequent references to the same global variable can use a naked global reference, as shown in the following example:

```objectscript
IF $DATA(^A(1,2,3))#2 {  
SET x=^(3)  }
```

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.

**Functions Related to $DATA**

For related information, see $GET and $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**

- KILL command
- SET command
- $GET function
- $ORDER function
- Using Multidimensional Storage (Globals) in Using Caché Globals
$DECIMAL

Returns a number converted to a Caché floating point value.

$DECIMAL(num,digits)

Parameters

<table>
<thead>
<tr>
<th>num</th>
<th>The numeric value to be converted. Commonly this is an IEEE floating point number.</th>
</tr>
</thead>
<tbody>
<tr>
<td>digits</td>
<td>Optional — An integer that specifies the number of significant digits to return. $DECIMAL rounds the return value to that number of digits, using the IEEE floating point rounding algorithm. Valid values are 1 through 38, and 0. If digits is greater than the number of digits the value is returned unchanged. If digits is 0, no rounding is performed on num unless it has more than 20 significant digits (see below for details on 0 value).</td>
</tr>
</tbody>
</table>

Description

$DECIMAL returns a floating-point number converted to the Caché decimal floating-point data type. This function is used to convert a fractional number in IEEE double-precision (64-bit) binary floating-point format to the corresponding fractional number in Caché decimal floating-point format. This is the inverse of the operation performed by the $DOUBLE function.

The Caché SQL data types DOUBLE and DOUBLE PRECISION represent IEEE floating point numbers; the FLOAT data type represents standard Caché fractional numbers.

IEEE floating point numbers are represented internally using 53 binary bits. Because most fractional decimal numbers have no exact binary representation, a fractional number in $DOUBLE format will usually differ slightly from its $DECIMAL conversion. Standard Caché fractional numbers have an approximate precision of 18.96 decimal digits on all supported Caché system platforms. When an IEEE floating point number is displayed as a fractional number the binary bits are often converted to a fractional number with far more than 18 decimal digits. This does not mean that IEEE floating point numbers are more precise than standard Caché fractional numbers.

The num value can be specified as a number or a numeric string. It is resolved to canonical form (leading and trailing zeros removed, multiple plus and minus signs resolved, etc.) before $DECIMAL conversion. If the num value is outside of the range of values that can be converted to FLOAT data type, $DECIMAL generates a <MAXNUMBER> error. Specifying a nonnumeric string to num returns 0. Specifying a mixed-numeric string (for example "7dwarves" or "7.5.4") to num truncates the input value at the first nonnumeric character then converts the numeric portion.

Default rounding is done as follows:

- **Fractional Numbers:** If digits is not specified and num has more than 19 significant digits, $DECIMAL rounds the fractional portion so that the resulting number is 19 digits (or fewer, if the rounding results in trailing zeros). $DECIMAL always rounds to the nearest fractional value with the greater absolute value.

- **Very Large Integer Numbers:** If digits is not specified and num has more than 19 significant integer digits to the left of the decimal point, $DECIMAL rounds so that the resulting integer has 19 significant digits, with the remaining integer digits represented by zeros.

- **Very Small Fractional Numbers less than 1:** If digits is not specified and num is a fractional number with more than 19 zeroes to the right of the decimal point before the significant value, $DECIMAL preserves the zeros and then rounds so that the significant portion of the fraction is rounded to 19 significant digits.

- **Integers with Very Small Fractional Numbers:** If digits is not specified and num is a number contains a non-zero integer portion with more than 19 zeroes to the right of the decimal point before a significant value, $DECIMAL rounds to the integer.
The digits argument can be used to round the return value to a specified number of digits. Trailing zeros are always deleted. If digits is a positive integer, rounding is done using the IEEE rounding standard. If num has more than 38 significant digits (and digits=38) $DECIMAL rounds the fractional portion of the number at the 38th digit and represents all of the following num digits with zeros. If digits is greater than 38, an <ILLEGAL VALUE> error is generated.

If digits is 0, the number is cast to string collation; this is equivalent to (+num)_"". If num is 20 digits or less, digits=0 is the same as digits=20.

However, if digits is 0 and num is more than 20 digits, special rounding (not IEEE rounding) is performed, as follows. Rounding is performed to return 20 digits. Special rounding is then performed on the 20th digit if it rounds to a 0 or a 5. In the case when the 20th digit would round up to a 0 or a 5, Caché rounds it down to a 9 or a 4, respectively. In the case when the 20th digit would round down to a 0 or a 5, Caché rounds it up to a 1 or a 6, respectively. Other 20th digit values are returned unchanged. This rounding algorithm is used to provide correct numeric collation and avoid rounding inconsistencies.

**Integer Divide**

With certain values, Caché decimal floating-point and IEEE double numbers yield a different integer divide product. For example:

```objectscript
WRITE !,"Integer divide operations:";
WRITE !,"Cache \: ",$DECIMAL(4.1)\.01  // 410
WRITE !,"Double \: ",$DOUBLE(4.1)\.01   // 409
```

For further details on arithmetic operations involving IEEE double numbers, see the appendix “Numeric Computing in InterSystems Applications” in the Caché Programming Orientation Guide.

**INF and NAN**

If num is INF, a <MAXNUMBER> error is generated. If num is NAN, an <ILLEGAL VALUE> error is generated. These invalid values are shown in the following example:

```objectscript
SET i=$DOUBLE("INF")
SET n=$DOUBLE("NAN")
WRITE $DECIMAL(i),!
WRITE $DECIMAL(n)
```

**Examples**

The following example demonstrates that $DECIMAL has no effect when applied to a fractional number that is already in Caché format:

```objectscript
SET x=$DECIMAL($ZPI)
SET y=$ZPI
IF x=y { WRITE !,"Identical:" 
  WRITE !,"Cache $DECIMAL: ",x 
  WRITE !,"Native Cache: ",y }
ELSE { WRITE !,"Different:" 
  WRITE !,"Cache $DECIMAL: ",x 
  WRITE !,"Native Cache: ",y }
```

The following example returns the value of pi as a $DOUBLE value and as a standard Caché numeric value. This example shows that equality operations should not be attempted between $DOUBLE and standard Caché numbers, and that equivalence cannot be restored by using $DECIMAL to convert IEEE back to Caché:

```objectscript
SET x=$DECIMAL($ZPI)
SET y=$DOUBLE($ZPI)
SET z=$DECIMAL(y)
IF x=y { WRITE !,"Cache & IEEE Same" 
  ELSEIF x=z { WRITE !,"Cache & IEEE-to-Cache same" 
  ELSE { WRITE !,"All three different" 
    WRITE !,"Cache decimal: ",x 
    WRITE !,"IEEE float: ",y 
    WRITE !,"IEEE to Cache: ",z 
```
The following example returns the $DECIMAL conversion of pi as a $DOUBLE value. These conversions are rounded by different $digits argument values:

```objectscript
SET x=$DOUBLE($ZPI)
WRITE !,$DECIMAL(x)
   /* returns 3.141592653589793116 (19 digits) */
WRITE !,$DECIMAL(x,1)
   /* returns 3 */
WRITE !,$DECIMAL(x,8)
   /* returns 3.1415927 (note rounding) */
WRITE !,$DECIMAL(x,12)
   /* returns 3.14159265359 (note rounding) */
WRITE !,$DECIMAL(x,18)
   /* returns 3.14159265358979312 */
WRITE !,$DECIMAL(x,19)
   /* returns 3.141592653589793116 (19 digits) */
WRITE !,$DECIMAL(x,20)
   /* returns 3.141592653589793116 (19 digits) */
WRITE !,$DECIMAL(x,21)
   /* returns 3.141592653589793116 (19 digits) */
WRITE !,$DECIMAL(x,0)
   /* returns 3.1415926535897931159 (20 digits) */
```

**See Also**

- ZZDUMP command
- $DOUBLE function
- $FNUMBER function
- $NUMBER function
- Data Types in Caché SQL Reference
- Operators in Using Caché ObjectScript
- Numeric Computing in InterSystems Applications in Caché Programming Orientation Guide
$DOUBLE

Returns a number converted to a 64-bit floating-point value.

$DOUBLE (num)

Parameter

| num     | The numeric value to be converted. You can also specify the strings “NAN” and “INF” (and their variants). |

Description

$DOUBLE returns a number converted to the IEEE double-precision (64-bit) binary floating-point data type. This type of floating-point number can contain up to 20 digits. If num has more than 20 digits, $DOUBLE rounds the fractional portion to the appropriate number of digits. If the integer portion of num is more than 20 digits, $DOUBLE rounds the integer to 20 significant digits and represents the additional digits with zeros.

$DOUBLE converts a Caché floating-point number to an IEEE double-precision floating-point number. $DECIMAL performs the inverse operation, converting an IEEE double-precision floating-point number to a standard Caché floating-point number.

$DOUBLE generates floating-point numeric values that accord with the IEEE double-precision (64-bit) binary floating point standard. It is primarily intended for interchange and compatibility with applications that use this data type standard. IEEE floating-point numbers are represented using binary notation. They have a precision of 53 binary bits, which corresponds to 15.95 decimal digits. (Note that the binary representation does not correspond exactly to a decimal fraction.)

IEEE floating-point numbers have greater min/max value range than standard Caché floating-point numbers. However, standard Caché floating-point numbers have a greater degree of precision. In most cases, standard Caché floating-point numbers are preferable.

Note: A Caché numeric string literal that exceeds the min/max range supported by Caché floating-point data types (for example, “1E128”) is automatically converted to an IEEE double-precision floating-point number. This conversion is only performed on numeric literals; it is not performed on the results of mathematical operations. This automatic conversion can be controlled on a per-process basis using the TruncateOverflow() method of the %SYSTEM.Process class. The system-wide default behavior can be established by setting the TruncateOverflow property of the Config.Miscellaneous class.

The num value can be specified as a number or a numeric string. It is resolved to canonical form (leading and trailing zeros removed, multiple plus and minus signs resolved, etc.) before $DOUBLE conversion. Specifying a nonnumeric string to num returns 0. Specifying a mixed-numeric string (for example ”7dwarves” or ”7.5.4”) to num truncates the input value at the first nonnumeric character then converts the numeric portion. A $DOUBLE numeric value supplied to a JSON array or JSON object follows different validation and conversion rules.

The Caché SQL data types DOUBLE and DOUBLE PRECISION represent IEEE floating-point numbers; the FLOAT data type represents standard Caché floating-point numbers.

Equality Comparisons and Mixed Arithmetic

Because numbers generated by $DOUBLE are converted to a binary representation that does not correspond exactly to decimal digits, equality comparisons between $DOUBLE values and standard Caché floating-point number values may yield unexpected results and should generally be avoided. Comparisons between $DOUBLE values and standard Caché floating-point number values are performed exactly, without rounding.
Mixed arithmetic operations involving a $DOUBLE value and one or more standard Caché numbers return a $DOUBLE value. In mixed arithmetic, Caché automatically converts all of the numbers to $DOUBLE values before performing the arithmetic operation. Caché handles conversions to/from $DOUBLE numeric representations and comparisons between numeric representations; these operations are, therefore, the same on all platforms. However, arithmetic operations involving $DOUBLE values are governed by the underlying operating system, and thus may occasionally differ between platforms. For further details on arithmetic operations involving IEEE double numbers, see the appendix “Numeric Computing in InterSystems Applications” in the Caché Programming Orientation Guide.

### Integer Divide

With certain values, Caché decimal floating-point and IEEE double numbers yield a different integer divide product. For example:

```plaintext
WRITE !,"Divide operations:"
WRITE !,"Cache /: ",4.1/.01 // 410
WRITE !,"Double /: ",$DOUBLE(4.1)/.01 // 410
WRITE !,"Integer divide operations:"
WRITE !,"Cache \: ",4.1\.01 // 410
WRITE !,"Double \: ",$DOUBLE(4.1)\.01 // 409
```

### Platform Independence

Standard Caché decimal floating-point numbers ($DECIMAL numbers) have an approximate precision of 18.96 decimal digits. This precision is consistent across all system platforms that Caché supports.

IEEE double-precision floating-point numbers ($DOUBLE numbers) have a standard internal representation that is platform-independent. Conversions and comparisons between $DOUBLE and $DECIMAL numbers are consistent across all system platforms that Caché supports. However, other computations on $DOUBLE numbers may show slight differences based on the system platform.

### INF and NAN

Following the IEEE standard, $DOUBLE can return the strings INF (infinity) and NAN (not a number). INF can be positive or negative (INF and -INF); NAN is always unsigned. While these are valid IEEE return values, they are not actual numbers.

#### INF and NAN as Input Values

One way to cause $DOUBLE to return INF and NAN is to specify the corresponding string as the num input value. These input strings are not case-sensitive, and can take leading plus and minus signs (INF resolves signs, NAN ignores signs). To return NAN, specify “NAN”, “sNAN”, “+NAN”, “-NAN”. To return INF, specify “INF”, “+INF”, “Infinity”. To return -INF, specify “-INF”, “+-INF”.

#### IEEEError

IEEEError controls how $DOUBLE responds to a numeric conversion that cannot be resolved. If IEEEError is set to 0, $DOUBLE returns INF and NAN when it cannot resolve a conversion. If IEEEError is set to 1, $DOUBLE generates standard Caché error codes when it cannot resolve a conversion. The default is 1.

This behavior can be controlled on a per-process basis using the IEEEError() method of the %SYSTEM.Process class. The system-wide default behavior can be established by setting the IEEEError property of the Config.Miscellaneous class.

### Returning INF and NAN

$DOUBLE can return INF and NAN when you specify an extremely large number, or when you specify an unresolvable arithmetic operation. These values are only returned when IEEEError is set to return INF and NAN.

Extremely large floating-point numbers are not supported. The maximum supported value for a $DOUBLE binary floating-point number is 1.7976931348623158079e308. The minimum supported value for a $DOUBLE binary floating-point number is 1.0E-323. A num value smaller than this returns 0.
**Note:** The maximum supported value for a Caché decimal floating-point number is 9.223372036854775807e145. The minimum supported value for a Caché decimal floating-point number is either 2.2250738585072013831e-308 (normal) or 4.9406564584124654417e-324 (denormalized).

The following table shows the value returned or error generated by unresolvable arithmetic operations:

<table>
<thead>
<tr>
<th>Input Value</th>
<th>IEEEError=0</th>
<th>IEEEError=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 1.0E308</td>
<td>INF</td>
<td>&lt;MAXNUMBER&gt;</td>
</tr>
<tr>
<td>&lt; 1.0E-323</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1/$DOUBLE(0)</td>
<td>INF</td>
<td>&lt;DIVIDE&gt;</td>
</tr>
<tr>
<td>1/$DOUBLE(–0)</td>
<td>–INF</td>
<td>&lt;DIVIDE&gt;</td>
</tr>
<tr>
<td>$DOUBLE(1)/0</td>
<td>INF</td>
<td>&lt;DIVIDE&gt;</td>
</tr>
<tr>
<td>$DOUBLE(0)/0</td>
<td>NAN</td>
<td>&lt;ILLEGAL VALUE&gt;</td>
</tr>
<tr>
<td>$ZLOG($DOUBLE(0))</td>
<td>–INF</td>
<td>&lt;DIVIDE&gt;</td>
</tr>
</tbody>
</table>

**Comparing INF and NAN**

INF can be compared as if it were a numerical value. Thus INF = INF, INF’ = –INF, –INF = –INF, and INF > –INF.

NAN cannot be compared as if it were a numerical value. Because NAN (Not A Number) cannot be meaningfully compared using numerical operators, Caché operations (such as equal to, less than, or greater than) that attempt to compare $DOUBLE(“NAN”) to another $DOUBLE(“NAN”) fail. Comparisons with NAN <= or >= are a special case, which is described in the appendix “Numeric Computing in InterSystems Applications” in the Caché Programming Orientation Guide.

$LISTSAME does consider a $DOUBLE(“NAN”) list element to be identical to another $DOUBLE(“NAN”) list element.

Caché does not distinguish between different NAN representations (NAN, sNAN, etc.). Caché considers all NANs to be the same, regardless of their binary representation.

**$ISVALIDNUM, $INUMBER, and $FNUMBER**

These ObjectScript functions provide support for $DOUBLE numbers.

$ISVALIDNUM supports INF and NAN. Although these strings are not numbers, $ISVALIDNUM returns 1 for these values, just as if they were numbers. When $DOUBLE is specified with a nonnumeric string, for example $DOUBLE(“”), Caché returns a value of 0. For this reason, $ISVALIDNUM($DOUBLE(“”)) returns 1, because 0 is a number.

$INUMBER and $FNUMBER provide a “D” format option that supports $DOUBLE values. $INUMBER converts a numeric to a IEEE floating-point number. $FNUMBER “D” support includes case conversion of INF and NAN, and choosing whether $DOUBLE(–0) should return 0 or -0.

**INF and NAN with Operators**

You can perform arithmetic and logical operations on INF and NAN. Use of operators with INF and NAN is not recommended; if such an operation is performed, the following are the results:

Arithmetic operators:
Division (/, \, or # operators)

<table>
<thead>
<tr>
<th>Addition</th>
<th>Subtraction</th>
<th>Multiplication</th>
<th>Division (/, , or # operators)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN+NAN=NAN</td>
<td>NAN-NAN=NAN</td>
<td>NAN*NAN=NAN</td>
<td>NAN/NAN=NAN</td>
</tr>
<tr>
<td>NAN+INF=INF</td>
<td>INF-NAN=NAN</td>
<td>INF*INF=INF</td>
<td>INF/INF=NAN</td>
</tr>
<tr>
<td>INF+INF=INF</td>
<td>INF-INF=NAN</td>
<td>INF*INF=INF</td>
<td>INF/INF=NAN</td>
</tr>
</tbody>
</table>

Logical operators:

<table>
<thead>
<tr>
<th>Equality (=)</th>
<th>NAN</th>
<th>INF</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>INF</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Less Than (&lt;) or Greater Than (&gt;)</th>
<th>NAN</th>
<th>INF</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>INF</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Other operators, such as pattern matching and concatenation, treat NAN and INF as three-character alphabetic strings.

For further details on operations involving IEEE double numbers, see the appendix “Numeric Computing in InterSystems Applications” in the Caché Programming Orientation Guide.

INF and NAN Examples

$DOUBLE returns an INF value (or a -INF for negative numbers) when the numeric value exceeds the available precision, as shown in the following example:

```objectscript
SET rtn=##class(%SYSTEM.Process).IEEEError(0)
SET x=$DOUBLE(1.2e300)
WRITE !,"Double: ",x
WRITE !,"Is number? ",$ISVALIDNUM(x)
SET y= $DOUBLE(x*x)
WRITE !,"Double squared: ",y
WRITE !,"Is number? ",$ISVALIDNUM(y)
```

$DOUBLE returns a NAN (not a number) value when the numeric value is invalid. For example, when an arithmetic expression involves two INF values, as shown in the following example. (An arithmetic expression involving a single INF value returns INF.)

```objectscript
SET rtn=##class(%SYSTEM.Process).IEEEError(0)
SET x=$DOUBLE(1.2e500)
WRITE !,"Double: ",x
WRITE !,"Is number? ",$ISVALIDNUM(x)
SET y= $DOUBLE(x-x)
WRITE !,"Double INF minus INF: ",y
WRITE !,"Is number? ",$ISVALIDNUM(y)
```

JSON Numeric Literals

JSON validation of numeric literals is described in the SET command. $DOUBLE numeric literals specified in a JSON array or JSON object are subject to the following additional rules:

- INF, -INF, and NAN values can be stored in JSON structures, but cannot be returned by %ToJSON(). Attempting to do so results in an <ILLEGAL VALUE> error, as shown in the following example:
SET jary=[123,($DOUBLE("INF"))]  // executes successfully
WRITE jary.%ToJSON()            // fails with <ILLEGAL VALUE> error

• $DOUBLE(−0) is stored in a JSON structure as −0.0. $DOUBLE(0) is stored in a JSON structure as 0. This is shown in the following example:

SET jary=[0,−0,($DOUBLE(0)),($DOUBLE(−0))]  
WRITE jary.%ToJSON() // returns [0,−0,0,−0.0]

Examples

The following example returns floating-point numbers of 20 digits:

WRITE !,$DOUBLE(999.12345678987654321)
WRITE !,$DOUBLE(99912345678987654321)
WRITE !,$DOUBLE(999123456789.87654321)

The following example returns the value of pi as a $DOUBLE value and as a standard Caché numeric value. This example shows that equality operations should not be attempted between $DOUBLE and standard Caché numbers, and that the number of digits returned is greater for standard Caché numbers:

SET x=$ZPI
SET y=$DOUBLE($ZPI)
IF x=y { WRITE !,"Same" } 
ELSE { WRITE !,"Different"
   WRITE !,"standard: ",x
   WRITE !,"IEEE float: ",y }

The following examples show that a floating-point number is not necessarily equivalent to a numeric string of the same value:

SET x=123.4567891234560 
SET y=123.4567891234567
IF x=$DOUBLE(x) { WRITE !,"Same" } 
ELSE { WRITE !,"Different" }
IF y=$DOUBLE(y) { WRITE !,"Same" } 
ELSE { WRITE !,"Different" }

SET x=1234567891234560
SET y=1234567891234567
IF x=$DOUBLE(x) { WRITE !,"Same" } 
ELSE { WRITE !,"Different" }
IF y=$DOUBLE(y) { WRITE !,"Same" } 
ELSE { WRITE !,"Different" }

See Also

• ZZDUMP command
• $DECIMAL function
• $FNUMBER function
• $NUMBER function
• Data Types in Caché SQL Reference
• Operators in Using Caché ObjectScript
• Numeric Computing in InterSystems Applications in Caché Programming Orientation Guide
$\text{EXTRACT}$

Extracts a substring from a character string by position, or replaces a substring by position.

$\text{EXTRACT}(\text{string}, \text{from}, \text{to})$

$\text{SET} \ \text{EXTRACT}(\text{string}, \text{from}, \text{to}) = \text{value}$

$\text{SET} \ \text{E}(\text{string}, \text{from}, \text{to}) = \text{value}$

**Parameters**

- **string**
  - The target string in which substrings are identified. Specify `string` as an expression that evaluates to a quoted string or a numeric value. In `SET $\text{EXTRACT}$` syntax, `string` must be a variable or a multi-dimensional property.

- **from**
  - Optional — Specifies the starting position within the target `string`. Characters are counted from 1. Permitted values are `n` (a positive integer specifying the character count from the beginning of `string`), `*` (specifying the last character in `string`), and `*-n` (offset integer count of characters backwards from end of `string`). `SET $\text{EXTRACT}$` syntax also supports `*+n` (offset integer count of characters to append beyond the end of `string`). A `from` without a `to` specifies a single character. A `from` with a `to` specifies a range of characters. If `from` is not specified, it defaults to 1.

- **to**
  - Optional — Specifies the end position (inclusive) for a range of characters. Must be used with `from`. Permitted values are `n` (a positive integer specifying the character count from the beginning of `string`), `*` (specifying the last character in `string`), and `*-n` (offset integer count of characters backwards from end of `string`). `SET $\text{EXTRACT}$` syntax also supports `*+n` (offset integer count of the end of a range of characters to append beyond the end of `string`).

**Description**

$\text{EXTRACT}$ identifies substrings within `string` by character count, either from the beginning of `string` or the end of `string`. A substring can be a single character or a range of characters.

$\text{EXTRACT}$ can be used in two ways:

- To return a substring from `string`. This uses the $\text{EXTRACT}(\text{string}, \text{from}, \text{to})$ syntax.
- To replace a substring within `string`. The replacement substring may be the same length, longer, or shorter than the original substring. This uses the `SET $\text{EXTRACT}(\text{string}, \text{from}, \text{to}) = \text{value}$` syntax.

**Returning a Substring**

$\text{EXTRACT}$ returns a substring by character position from `string`. The nature of this substring extraction depends on the parameters used.

- $\text{EXTRACT}(\text{string})$ extracts the first character in the string.

  ```objectscript
  SET mystr="ABCD"
  WRITE $\text{EXTRACT}(\text{mystr})$
  ```

- $\text{EXTRACT}(\text{string}, \text{from})$ extracts a single character in the position specified by `from`. The `from` value can be an integer count from the beginning of the string, an asterisk specifying the last character of the string, or an asterisk with a negative integer specifying a count backwards from the end of the string.

  The following example extracts single letters from the string “ABCD”:
SET mystr="ABCD"
WRITE !,$EXTRACT (mystr,2)  // *B* the 2nd character
WRITE !,$EXTRACT (mystr,* )  // *D* the last character
WRITE !,$EXTRACT (mystr,=*-2) // *B* the offset 2 characters from end
WRITE !,$EXTRACT (mystr,=*-0)  // *D* the last character by 0 offset

- **$EXTRACT**(string,from,to) extracts the range of characters starting with the **from** position and ending with the **to** position (inclusive). For example, if variable var2 contains the string “1234Alabama567”, the following $EXTRACT functions both return the string “Alabama”:

```
SET var2="1234Alabama567"
WRITE !,$EXTRACT (var2,5,11)
WRITE !,$EXTRACT (var2,=*-9,*-3)
```

### Parameters

#### string

The target string in which the substring is identified.

When **$EXTRACT** is used to **return a substring**, **string** can be a string literal enclosed in quotation marks, a canonical numeric, a variable, an object property, or any valid ObjectScript expression that evaluates to a string or a numeric. If you specify a null string ("" ) as the target string, **$EXTRACT** always returns the null string, regardless of the other parameter values.

When **$EXTRACT** is used with **SET** on the left hand side of the equals sign to **replace a substring**, **string** can be a variable name or a multidimensional property reference; it cannot be a non-multidimensional object property.

#### from

The **from** parameter can specify a single character, or the beginning of a range of characters.

- If **from** is *n* (a positive integer), **$EXTRACT** counts characters from the beginning of **string**.
- If **from** is *(asterisk)*, **$EXTRACT** returns the last character in **string**.
- If **from** is *-n* (an asterisk followed by a negative number), **$EXTRACT** counts characters by offset from the end of **string**. Thus, *-0* is the last character in **string**, *-1* is the next-to-last character in **string** (an offset of 1 from the end).
- For **SET $EXTRACT** syntax only — If **from** is *+n* (an asterisk followed by a positive number), **SET $EXTRACT** appends characters by offset beyond the end of **string**. Thus, *+1* appends a character beyond the end of **string**, *+2* appends a character two positions beyond the end of **string**, padding the skipped position with a blank space. *+0* is the last character in **string**.

If the **from** integer value is greater than the number of characters in the string, **$EXTRACT** returns a null string. With a **from** *-n* value, if *n* is equal to or greater than the number of characters in the string, **$EXTRACT** returns a null string. If the **from** value is 0 or a negative number, **$EXTRACT** returns a null string; however, if **from** is used with **to**, a **from** value of 0 or a negative number is treated as a value of 1.

If **from** is used with the **to** parameter, **from** identifies the start of the range to be extracted and must be less than the value of **to**. If **from** equals **to**, **$EXTRACT** returns the single character at the specified position. If **from** is greater than **to**, **$EXTRACT** returns a null string. If used with the **to** parameter, a **from** value less than 1 (zero, or a negative number) is treated as if it were the number 1.

#### to

The **to** parameter must be used with the **from** parameter. It must be a positive integer, *(asterisk)*, or *-n* (an asterisk followed by a negative integer). If the **to** value is an integer greater than or equal to the **from** value, **$EXTRACT** returns the specified substring. If the **to** value is an asterisk, **$EXTRACT** returns the substring beginning with the **from** character through the end of the string. If **to** is an integer greater than the length of the string, **$EXTRACT** also returns the substring beginning with the **from** character through the end of the string.
If the *from* and *to* positions are the same, \$EXTRACT returns a single character. If the *to* position is closer to the beginning of the string than the *from* position, \$EXTRACT returns the null string.

If you omit the *to* parameter, only one character is returned. If *from* is specified, \$EXTRACT returns the character identified by *from*. If both *to* and *from* are omitted, \$EXTRACT returns the first character of *string*.

For SET \$EXTRACT syntax only — If *to* is *+n*, SET \$EXTRACT appends a range of characters by offset beyond the end of *string*, padding with blank spaces as needed. If *from* represents a character position after the end of *string*, SET \$EXTRACT appends characters. If *from* represents a character position before the end of *string*, SET \$EXTRACT may both replace and append characters.

**Specifying *-n* and *+n* Parameter Values**

When using a variable to specify *-n* or *+n*, you must always specify the asterisk and a sign character in the parameter itself.

The following are valid specifications of *-n*:

```plaintext
SET count=2
SET alph="abcd"
WRITE \$EXTRACT(alph, *-count)
```

```plaintext
SET count=-2
SET alph="abcd"
WRITE \$EXTRACT(alph, *+count)
```

The following is a valid specification of *+n*:

```plaintext
SET count=2
SET alph="abcd"
SET \$EXTRACT(alph, *+count)="F"
WRITE alph
```

Whitespace is permitted within these parameter values.

**Examples: Returning a Substring**

The following example returns “D”, the fourth character in the string:

```plaintext
SET x="ABCDEFGH IJK"
WRITE \$EXTRACT(x, 4)
```

The following example returns “K”, the last character in the string:

```plaintext
SET x="ABCDEFGH IJK"
WRITE \$EXTRACT(x, *)
```

In the following example, all the \$EXTRACT functions return “J” the next-to-last character in the string:

```plaintext
SET n=-1
SET m=1
SET x="ABCDEFGH IJK"
WRITE 1, \$EXTRACT(x, *-1)
WRITE 1, \$EXTRACT(x, *-m)
WRITE 1, \$EXTRACT(x, *+n)
WRITE 1, \$EXTRACT(x, *-1, *-1)
```

Note that a minus or plus sign is needed between the asterisk and the integer variable.

The following example shows that the one-argument format is equivalent to the two-argument format when the *from* value is “1”. Both \$EXTRACT functions return “H”.

```plaintext
SET x="HELLO"
WRITE 1, \$EXTRACT(x)
WRITE 1, \$EXTRACT(x, 1)
```

The following example returns a substring “THIS IS” which is composed of the first through seventh characters.
The following example also returns the substring “THIS IS”. When the from variable contains a value less than 1, \texttt{EXTRACT} treats that value as 1. Thus, the following example returns a substring composed of the first through seventh characters.

\begin{verbatim}
SET X="THIS IS A TEST"
WRITE $EXTRACT(X, -1, 7)
\end{verbatim}

The following example returns the last four characters of the string:

\begin{verbatim}
SET X="THIS IS A TEST"
WRITE $EXTRACT(X, -*3, *)
\end{verbatim}

The following example also returns the last four characters of the string:

\begin{verbatim}
SET X="THIS IS A TEST"
WRITE $EXTRACT(X, -*3, 14)
\end{verbatim}

The following example extracts a substring from an object property:

\begin{verbatim}
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SchemaPath="MyTests,Sample,Cinema"
WRITE "whole schema path: ",tStatement.%SchemaPath,!
WRITE "start of schema path: ",$EXTRACT(tStatement.%SchemaPath,1,10),!
\end{verbatim}

\section*{Replacing a Substring Using \texttt{SET $EXTRACT}}

You can use \texttt{$EXTRACT} with the \texttt{SET} command to replace a specified character or range of characters with another value. You can also use it to append characters to the end of a string.

When \texttt{$EXTRACT} is used with \texttt{SET} on the left hand side of the equals sign, \textit{string} can be a valid variable name. If the variable does not exist, \texttt{SET $EXTRACT} defines it. The \textit{string} parameter can also be a multidimensional property reference; it cannot be a non-multidimensional object property. Attempting to use \texttt{SET $EXTRACT} on a non-multidimensional object property results in an \texttt{<OBJECT DISPATCH>} error.

You cannot use \texttt{SET (a,b,c,...)=value} syntax with \texttt{$EXTRACT} (or \texttt{PIECE} or \texttt{LIST}) on the left of the equals sign, if the function uses relative offset syntax: * representing the end of a string and *n or *n representing relative offset from the end of the string. You must instead use \texttt{SET a=value,b=value,c=value,...} syntax.

The simplest form of \texttt{SET $EXTRACT} is a one-for-one substitution:

\begin{verbatim}
SET alph="ABZD"
SET $EXTRACT(alph,3)="C"
WRITE alph ; "ABCD"
\end{verbatim}

You can append characters to \textit{string} either by specifying \textit{to} as a positive integer that is 1 larger than the length of \textit{string}, or by specifying \textit{to} as *+1, as shown in the following examples:

\begin{verbatim}
SET alph="ABCD"
SET $EXTRACT(alph,5)="E"
WRITE alph ; "ABCDE"
\end{verbatim}

\begin{verbatim}
SET alph="ABCD"
SET $EXTRACT(alph,*+1)="E"
WRITE alph ; "ABCDE"
\end{verbatim}

If you specify \textit{to} larger than the string plus 1, \texttt{$EXTRACT} pads with blank spaces:

\begin{verbatim}
SET alph="ABCD"
SET len=$LENGTH(alph)
SET $EXTRACT(alph,len+2)="F"
WRITE alph ; "ABCD F"
\end{verbatim}

\begin{verbatim}
SET alph="ABCD"
SET $EXTRACT(alph,*+2)="F"
WRITE alph ; "ABCD F"
\end{verbatim}
You can also extract a string and replace it with a string of a different length. For example, the following command extracts the string “Rhode Island” from $foo and replaces it with the string “Texas”, with no padding.

```objectscript
SET foo="Deep in the heart of Rhode Island"
SET $EXTRACT(foo,22,33)="Texas"
WRITE foo ; "Deep in the heart of Texas"
```

You can extract a string and set it to the null string, removing the extracted characters from the string:

```objectscript
SET alph="ABCzzzzzzD"
SET $EXTRACT(alph,4,8)="
WRITE alph ; "ABCD"
```

If you specify from larger than to, no replacement occurs:

```objectscript
SET alph="ABCD"
SET $EXTRACT(alph,4,3)="X"
WRITE alph ; "ABCD"
```

In the following example, assume that variable $x does not exist.

```objectscript
KILL x
SET $EXTRACT(x,1,4)="ABCD"
WRITE x ; "ABCD"
```

The SET command creates variable $x and assigns it the value “ABCD”.

$EXTRACT performs leading padding with blank spaces as required, but does not perform trailing padding. The following example inserts the value “F” in the sixth position past the end of the string, but inserts no additional characters in positions 7 and 8:

```objectscript
SET alph="ABCD"
SET $EXTRACT(alph,6,8)="F"
WRITE alph ; "ABCD F"
```

The following example inserts the value “F” in the sixth position and adds characters past the specified range:

```objectscript
SET alph="ABCD"
SET $EXTRACT(alph,6,8)="FGHIJ"
WRITE alph ; "ABCD FGHIJ"
```

The following example shortens a character string by extracting a from, to range larger than the number of values in the replacement string.

```objectscript
SET x="ABCDEFGH"
SET $EXTRACT(x,3,6)="Z"
WRITE x
```

inserts the value “Z” in the third position and removes positions 4, 5 and 6. Variable $x now contains the value “ABZGH” and has a length of 5.

**Notes**

$EXTRACT and Unicode

The $EXTRACT function operates on characters, not bytes. Therefore, Unicode strings are handled the same as ASCII strings, as shown in the following example using the Unicode character for “π” ($CHAR(960)):

```objectscript
Caché ObjectScript Reference 285
```
IF $SYSTEM.Version.IsUnicode()  {
    SET a="QT PIE"
    SET b="#CHAR(960)"
    SET a1=$EXTRACT(a,-33,4)
    SET a2=$EXTRACT(a,4,4)
    SET a3=$EXTRACT(a,4,99)
    SET b1=$EXTRACT(b,-33,4)
    SET b2=$EXTRACT(b,4,4)
    SET b3=$EXTRACT(b,4,99)
    WRITE !,"ASCII form returns ",!,a1,!,a2,!,a3
    WRITE !,"Unicode form returns ",!,b1,!,b2,!,b3
}  
ELSE {WRITE "This example requires a Unicode installation of Caché"}

For further details on Caché Unicode support, refer to Unicode in Using Caché ObjectScript.

Surrogate Pairs

$EXTRACT does not recognize surrogate pairs. Surrogate pairs are used to represent some Chinese characters and to support the Japanese JIS2004 standard. You can use the $WISWIDE function to determine if a string contains a surrogate pair. The $WEXTRACT function recognizes and correctly parses surrogate pairs. $EXTRACT and $WEXTRACT are otherwise identical. However, because $EXTRACT is generally faster than $WEXTRACT, $EXTRACT is preferable for all cases where a surrogate pair is not likely to be encountered.

$EXTRACT in DTM Modes

In the DTM and DTM-J modes, $EXTRACT supports two additional arguments, as follows:

$EXTRACT(string,from,to,replace,pad)

The optional replace argument replaces the substring specified by from and to with the replace substring, and returns the result. The original string is not changed.

The optional pad argument specifies a padding character. This is used when the from argument specifies a position beyond the end of string. The returned string is padded to the location specified by from followed by the replace substring. The pad value may be any single character; a nonnumeric character must be enclosed in quotes. To specify a quote character as the pad character literal, double it.

You can use the LanguageMode() method of the %SYSTEM.Process class to set DTM mode (2) or DTM-J mode (7).

The following example shows the four-argument replace syntax:

SET x="ABCDEFGH"
DO ##class(%SYSTEM.Process).LanguageMode(2)
WRITE $EXTRACT(x,3,6,"##") /* returns "AB##GH" */

The following example use the four-argument syntax to append the replace string:

SET x="ABCDEFGH"
DO ##class(%SYSTEM.Process).LanguageMode(2)
WRITE $EXTRACT(x,1,0,"##") /* returns "##ABCDEFGH" */

The following example shows the five-argument pad and replace syntax:

SET x="ABCDEFGH"
DO ##class(%SYSTEM.Process).LanguageMode(2)
WRITE $EXTRACT(x,12,16,"##","*") /* returns "ABCDEFGH***##" */

Note: When using four-argument or five-argument syntax, the $EXTRACT from and to arguments do not support asterisk syntax.

SET $EXTRACT cannot be used with four-argument or five-argument syntax.
$EXTRACT Compared with $PIECE and $LIST

$EXTRACT determines a substring by counting characters from the beginning of a string. $EXTRACT takes as input any ordinary character string. $PIECE and $LIST both work on specially prepared strings.

$PIECE determines a substring by counting user-defined delimiter characters within the string.

$LIST determines an element from an encoded list by counting elements (not characters) from the beginning of the list. $LIST cannot be used on ordinary strings, and $EXTRACT cannot be used on encoded lists.

See Also

- SET command
- $FIND function
- $LENGTH function
- $PIECE function
- $REVERSE function
- $WEXTRACT function
- $WFIND function
- $WISWIDE function
**$FACTOR**

Converts an integer to a $BIT bitstring.

\[ $FACTOR(num, scale) \]

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>An expression that evaluates to a number. num is converted to a positive integer before bitstring conversion. A negative number is converted to a positive number (its absolute value). A fractional number is rounded to an integer.</td>
</tr>
<tr>
<td>scale</td>
<td>Optional — An integer used as a power-of-ten exponent (scientific notation) multiplier for num. The default is 0.</td>
</tr>
</tbody>
</table>

**Description**

$FACTOR returns the $BIT format bitstring that corresponds to the binary representation of the supplied integer. It performs the following operations:

- If you specify a negative number, $FACTOR takes the absolute value of the number.
- If you specify a scale $FACTOR multiplies the integer by \(10^{scale}\).
- If you specify a fractional number $FACTOR rounds this number to an integer. When rounding numbers, Caché rounds the fraction .5 up to the next highest integer.
- $FACTOR converts the integer to its binary representation.
- $FACTOR converts this binary number to $BIT encoded binary format.

The binary string returned specifies bit positions starting from the least significant bit at position 1 (one's place at position 1). This corresponds to the bitstrings used by the various $BIT functions.

**Parameters**

**num**

A number (or an expression that evaluates to a number). $FACTOR applies the scale parameter (if supplied), converts this number to an integer by rounding, and then returns the corresponding bitstring. num can be positive or negative. If num is a mixed numeric string (for example “7dwarves” or “5.6.8”) $FACTOR converts the numeric part of the string (in our example, 7 and 5.6) until it encounters a nonnumeric character. If num is zero, or rounds to zero, or is the null string (""), or a nonnumeric string, $FACTOR returns an empty string. The $DOUBLE values INF, –INF, and NAN return the empty string.

**scale**

An integer that specifies the scientific notation exponent to apply to num. For example, if scale is 2, then scale represents 10 exponent 2, or 100. This scale value is multiplied by num. For example, $FACTOR(7,2) returns the bitstring that corresponds to the integer 700. This multiplication is done before rounding num to an integer. By default, scale is 0.

**Examples**

The following example show the conversion of the integers 1 through 9 to bitstrings:
SET x=1
WHILE x<10 {
  WRITE !,x,="="
  FOR i=1:1:8 {
    WRITE $BIT($FACTOR(x),i) }
  SET x=x+1 }

The following example show $FACTOR conversion of negative numbers and fractions to positive integers:

FOR i=1:1:8 {WRITE $BIT($FACTOR(17),i)}
WRITE " Positive integer",!
FOR i=1:1:8 {WRITE $BIT($FACTOR(-17),i)}
WRITE " Negative integer (absolute value)",!
FOR i=1:1:8 {WRITE $BIT($FACTOR(16.5),i)}
WRITE " Positive fraction (rounded up)",!
FOR i=1:1:8 {WRITE $BIT($FACTOR(-16.5),i)}
WRITE " Negative fraction (rounded up)"

The following example show the bitstring returned when the scale parameter is specified:

SET x=2.7
WRITE !,x," scaled then rounded to an integer:"","!!
FOR i=1:1:12 {
  WRITE $BIT($FACTOR(x),i) }
WRITE " binary = ",$NORMALIZE(x,0)," decimal",!
SET scale=1
SET y=x*(10**scale)
FOR i=1:1:12 {
  WRITE $BIT($FACTOR(x,scale),i) }
WRITE " binary = ",$NORMALIZE(y,0)," decimal","!
SET scale=2
SET y=x*(10**scale)
FOR i=1:1:12 {
  WRITE $BIT($FACTOR(x,scale),i) }
WRITE " binary = ",$NORMALIZE(y,0)," decimal"

See Also

• $BIT function
• $BITCOUNT function
• $BITFIND function
• $BITLOGIC function
• $DOUBLE function
$FIND

Finds a substring by value and returns an integer specifying its end position in the string.

$FIND(string,substring,position)
$F(string,substring,position)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>string</em></td>
<td>The target string that is to be searched. It can be a variable name, a numeric value, a string literal, or any valid ObjectScript expression that resolves to a string.</td>
</tr>
<tr>
<td><em>substring</em></td>
<td>The substring that is to be searched for. It can be a variable name, a numeric value, a string literal, or any valid ObjectScript expression that resolves to a string.</td>
</tr>
<tr>
<td><em>position</em></td>
<td>Optional — A position within the target string at which to start the search. It must be a positive integer.</td>
</tr>
</tbody>
</table>

**Description**

$FIND returns an integer specifying the end position of a substring within a string. $FIND searches *string* for *substring*. $FIND is case-sensitive. If *substring* is found, $FIND returns the integer position of the first character following *substring*. If *substring* is not found, $FIND returns a value of 0.

Because $FIND returns the position of the character following the *substring*, when *substring* is a single character that matches the first character of *string* $FIND returns 2. When *substring* is the null string (""), $FIND returns 1.

You can include the *position* option to specify a starting position for the search. If *position* is greater than the number of characters in *string*, $FIND returns a value of 0.

$FIND counts characters, not bytes. Therefore, it can be used with strings containing 8-bit or 16-bit (Unicode) characters. For further details on Caché Unicode support, refer to Unicode in Using Caché ObjectScript.

**Examples**

For example, if variable *var1* contains the string “ABCDEFG” and variable *var2* contains the string “BCD,” the following $FIND returns the value 5, indicating the position of the character (“E”) that follows the *var2* string:

```objectscript
SET var1="ABCDEFG",var2="BCD"
WRITE $FIND(var1,var2)
```

The following example returns 4, the position of the character immediately to the right of the substring “FOR”.

```objectscript
SET X="FOREST"
WRITE $FIND(X,"FOR")
```

In the following examples, $FIND searches for a substring that is not in *string*, for a null substring, and for a substring that is the first character of *string*. The examples return 0, 1, and 2, respectively:

```objectscript
WRITE 1,$FIND("aardvark","z") ; returns 0
WRITE 1,$FIND("aardvark",""") ; returns 1
WRITE 1,$FIND("aardvark","a") ; returns 2
```

The following examples show what happens when *string* is a null string:

```objectscript
WRITE 1,$FIND("","z") ; returns 0
WRITE 1,$FIND("",""") ; returns 1
```

The following example returns 14, the position of the character immediately to the right of the first occurrence of “R” after the seventh character in X.

```objectscript
WRITE 1,$FIND("","R")
```
In the following example, \$FIND begins its search after the last character in string. It returns zero (0):

```
SET X="EVERGREEN FOREST",Y="R"
WRITE $FIND(X,Y,7)
```

The following example uses \$FIND with \$REVERSE to perform a search operation from the end of the string. This example locates the last example of a string within a line of text. It returns the position of that string as 33:

```
SET line="THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG."
SET position=$LENGTH(line)+2-$FIND($REVERSE(line),$REVERSE("THE"))
WRITE "The last THE in the line begins at ",position
```

For more information, refer to Indirection in Using Caché ObjectScript.

**Notes**

**\$FIND, \$EXTRACT, \$PIECE, and \$LIST**

- \$FIND locates a substring by value and returns a position.
- \$EXTRACT locates a substring by position and returns the substring value.
- \$PIECE locates a substring by a delimiter character or delimiter string, and returns the substring value.
- \$LIST operates on specially encoded strings. It locates a substring by substring count and returns the substring value.

The \$FIND, \$EXTRACT, \$LENGTH, and \$PIECE functions operate on standard character strings. The various \$LIST functions operate on encoded character strings, which are incompatible with standard character strings. The sole exception is the one-argument and two-argument forms of \$LIST, which take an encoded character string as input, but output a single element value as a standard character string.

**Surrogate Pairs**

\$FIND does not recognize surrogate pairs. Surrogate pairs are used to represent some Chinese characters and to support the Japanese JIS2004 standard. You can use the \$WISWIDE function to determine if a string contains a surrogate pair. The \$WFIND function recognizes and correctly parses surrogate pairs. \$FIND and \$WFIND are otherwise identical. However, because \$FIND is generally faster than \$WFIND, \$FIND is preferable for all cases where a surrogate pair is not likely to be encountered.

**See Also**

- \$EXTRACT function
- \$LENGTH function
- \$PIECE function
- \$REVERSE function
- \$SWEXTRACT function
- \$WFIND function
• **SWISWIDE** function
$FNUMBER

Formats a numeric value with a specified format; optionally rounds or zero fills to a specified precision.

$FNUMBER(inumber,format,decimal)
$FN(inumber,format,decimal)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inumber</td>
<td>The number to be formatted. It can be a numeric literal, a variable, or any valid ObjectScript expression that evaluates to a numeric value.</td>
</tr>
<tr>
<td>format</td>
<td>Optional — Specifies how the number is to be formatted. Specified as a quoted string consisting of zero or more format codes, in any order. Format codes are described below. Note that some format codes are incompatible and result in an error. For default formatting, with or without the decimal parameter, you can specify the empty string (&quot;&quot;&quot;). If omitted, defaults to the empty string (&quot;&quot;).</td>
</tr>
<tr>
<td>decimal</td>
<td>Optional — The number of fractional decimal digits to be included in the returned number. If format is omitted, include a placeholder comma before specifying decimal.</td>
</tr>
</tbody>
</table>

Description

$FNUMBER returns the number specified by inumber in the specified format.

Parameters

inumber

An expression that resolves to a number. Before $FNUMBER performs any operation, Caché performs its standard numeric resolution on inumber as follows: it resolves variables, performs string operations such as concatenation, converts strings to numerics, performs numeric expression operations, then converts the resulting numeric to canonical form. This is the number that $FNUMBER formats.

If inumber is a string, Caché first converts it to a number, truncating at the first non-numeric character. If the first character of the string is a non-numeric character, Caché converts the string to 0.

format

The possible format codes are as follows. You can specify them singly or in combination. Alphabetic codes are not case-sensitive.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;&quot;</td>
<td>Empty string. Returns inumber in canonical number format. This format is the same as &quot;L&quot; format.</td>
</tr>
<tr>
<td>+</td>
<td>Returns a nonnegative number prefixed by the PlusSign property of the current locale (+&quot; by default). If the number is negative, it returns the number prefixed by the MinusSign property of the current locale (-&quot; by default).</td>
</tr>
<tr>
<td>-</td>
<td>Returns the absolute value of a number. Always returns a negative number without the MinusSign character. Returns a positive number without the PlusSign character. When combined with the &quot;+&quot; format code (+=&quot;+, returns positive numbers with a plus sign, negative numbers with no sign. This code cannot be used with the &quot;P&quot; format code; attempting to do so results in a &lt;SYNTAX&gt; error.</td>
</tr>
</tbody>
</table>

Caché ObjectScript Reference
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>,</td>
<td>Returns the number with the value of the <code>NumericGroupSeparator</code> property of the current locale inserted every <code>NumericGroupSize</code> numerals to the left of the decimal point. Combining &quot;,&quot; with either &quot;,&quot; or &quot;N&quot; formats results in a <code>&lt;FUNCTION&gt;</code> error.</td>
</tr>
<tr>
<td>.</td>
<td>Returns the number using standard European formatting, regardless of the current locale settings. Sets <code>DecimalSeparator</code> to comma (,), <code>NumericGroupSeparator</code> to period (.), <code>NumericGroupSize</code> to 3, <code>PlusSign</code> to plus (+), <code>MinusSign</code> to minus (-). Combining &quot;,&quot; with either &quot;,&quot; or &quot;O&quot; formats results in a <code>&lt;FUNCTION&gt;</code> error.</td>
</tr>
</tbody>
</table>
| D    | $DOUBLE special formatting. This code has two effects:  
|      | “D” specifies that $DOUBLE(-0) should return -0; otherwise, $DOUBLE(-0) returns 0. However, 
|      | “-D” overrides the negative sign and returns 0. 
|      | You can specify “D” or “d” for this code; a returned INF or NAN will be expressed in the corresponding uppercase or lowercase letters. The default is upper case. |
| E    | E-notation (scientific notation). Returns the number in scientific notation. If you omit the `decimal` number of fractional digits, 6 is used as the default. You can specify “E” or “e” for this code; the returned value will contain the corresponding uppercase or lowercase symbol. The exponent portion of the returned value is two digits in length with a leading sign, unless three exponent digits are required. “E” and “G” are incompatible and result in a `<FUNCTION>` error. |
| G    | E-notation or fixed decimal notation. If the number of fractional digits that would result from conversion to scientific notation is larger than the `decimal` value (or the default of 6 decimal digits), the number is returned in scientific notation. For example, $FNUMBER(1234.99,"G",2) returns 1.23E+03. If the number of fractional digits that would result from conversion to scientific notation is equal to or smaller than the `decimal` value (or the default of 6 decimal digits), the number is returned in fixed decimal (standard) notation. For example, $FNUMBER(1234.99,"G",3) returns 1235. You can specify “G” or “g” for this code; the returned scientific notation value will contain the corresponding uppercase “E” or lowercase “e”. “E” and “G” are incompatible and result in a `<FUNCTION>` error. |
| L    | Leading sign. Sign, if present, must precede the numerical portion of `inumber`. Parentheses are not permitted. This code cannot be used with the “P” or “T” format codes; attempting to do so results in a `<SYNTAX>` or `<FUNCTION>` error. Leading sign is the default format. |
| N    | No `NumericGroupSeparator`. Does not allow the use of a numeric group separator. This format code is incompatible with the comma (,) format code. When used with the dot format code ("N. ") the number is formatted with the European decimal separator but no numeric group separators. |
| O    | ODBC locale. Overrides the current locale, and instead uses the standard ODBC locale with the following values: `PlusSign`=+; `MinusSign`=-; `DecimalSeparator`=.; `NumericGroupSeparator`=,; `NumericGroupSize`=3. By itself, the “O” format code uses only the ODBC `MinusSign` and `DecimalSeparator`. This format code is incompatible with the dot (.) format code. When used with the comma format code ("O, ") the number is formatted with the ODBC decimal separator and ODBC numeric group separators. |
| P    | Parentheses sign. Returns a negative number in parentheses and without a leading `MinusSign` locale property value. Otherwise, it returns the number without parentheses, but with a leading and trailing space character. This code cannot be used with the “+”, “-”, “L”, or “T” format codes; attempting to do so results in a `<SYNTAX>` error. |
Trailing sign. Returns the number with a trailing sign if a prefix sign would otherwise have been generated. However, it does not force a trailing sign. To produce a trailing sign for a nonnegative number (positive or zero), you must also specify the "+" format code. To produce a trailing sign for a negative number, you must not specify the "-" format code. The trailing sign used is determined by the PlusSign and MinusSign properties of the current locale respectively. A trailing space character, but no sign, is inserted in the case of a nonnegative number with "+" omitted or in the case of a negative number with "-" specified. Parentheses are not permitted. This code cannot be used with the "L" or "P" format codes; attempting to do so results in a <SYNTAX> or <FUNCTION> error.

In Caché, fractional numbers less than 1 are represented in Caché canonical form without a zero integer: 0.66 becomes .66. This is the $FNUMBER default. However, most $FNUMBER format options return fractional numbers less than 1 with a leading zero integer: .66 becomes 0.66. Two-parameter $FNUMBER with a format of "" (empty string), "L" (which is functionally identical to empty string), and "D" return fractional numbers less than 1 in canonical form: .66. All other two-parameter $FNUMBER format options, and all three-parameter $FNUMBER format options, return fractional numbers less than 1 with a single leading zero integer: 000.66 or .66 both becomes 0.66. This is the fractional number format for JSON numbers.

The $DOUBLE function can return the values INF (infinite) and NAN (not a number). INF can take a negative sign; format codes represent INF as if it were a number. For example: +INF, INF-, (INF). NAN does not take a sign; the only format code that affects NAN is “d”, which returns it in lowercase letters. The “E” and “G” codes have no effect on INF and NAN values.

**decimal**

The decimal parameter specifies the number of fractional digits to include in the returned value. Specify decimal as a positive integer, or any valid ObjectScript variable or expression that evaluates to a positive integer. If decimal is a negative number, Caché treats it as a 0 value. If decimal is a fractional number, Caché truncates to its integer component.

- If decimal is greater than the number of fractional digits in inumber, the remaining positions are zero filled.
- If decimal is less than the number of fractional digits in inumber, Caché rounds inumber to the appropriate number of fractional digits.
- If decimal is 0, inumber is returned as an integer with no decimal separator character. Caché rounds inumber to the appropriate integer.

If inumber is less than 1 and decimal greater than 0, $FNUMBER always returns a single zero in the integer position before the decimal separator character, regardless of the format value. This representation of fractional numbers differs from Caché canonical form.

You can specify the decimal parameter to control the number of fractional digits returned, after rounding is performed. For example, assume that variable c contains the number 6.25198.

```
SET c="6.25198"
SET x=$FNUMBER(c,"+",3)
SET y=$FNUMBER(c,"+",8)
WRITE !,x,!,y
```

The first $FNUMBER returns +6.252 and the second returns +6.25198000.

**Examples**

The following examples show how the different formatting designations can affect the behavior of $FNUMBER. These examples assume that the current locale is the default locale.

The following example shows the effects of sign codes on a positive number:
SET a=1234
WRITE $FNUMBER(a),! ; returns 1234
WRITE $FNUMBER(a,""),! ; returns 1234
WRITE $FNUMBER(a,"+"),! ; returns +1234
WRITE $FNUMBER(a,"-"),! ; returns 1234
WRITE $FNUMBER(a,"L"),! ; returns 1234
WRITE $FNUMBER(a,"T"),! ; returns 1234 (with a trailing space)
WRITE $FNUMBER(a,"T+"),! ; returns 1234+

The following example shows the effects of sign codes on a negative number:

SET b=-1234
WRITE $FNUMBER(b,""),! ; returns -1234
WRITE $FNUMBER(b,"+"),! ; returns -1234
WRITE $FNUMBER(b,"-"),! ; returns -1234
WRITE $FNUMBER(b,"L"),! ; returns -1234
WRITE $FNUMBER(b,"T"),! ; returns 1234-

The following example shows the effects of the “P” format code on positive and negative numbers. This example writes asterisks before and after the number to show that a positive number is returned with a leading and a trailing blank:

WRITE "**",$FNUMBER(-123,"P"),"**",! ; returns *(123)*
WRITE "**",$FNUMBER(123,"P"),"**",! ; returns * 123 *

The following example returns 1,234,567.81. The "," format returns x in American format, inserting commas as numeric group separators and a period as the decimal separator:

SET x=1234567.81
WRITE $FNUMBER(x,",,")

The following example returns 1.234.567,81. The "." format returns x in European format, inserting periods as numeric group separators and a comma as the decimal separator:

SET x=1234567.81
WRITE $FNUMBER(x,".")

The following 3-parameter example returns 124,329.00. $FNUMBER inserts a comma as numeric group separator, adds a period as the decimal separator, and appends two zeros as fractional digits to the value of x.

SET x=124329
WRITE $FNUMBER(x,",,",2)

The following 3-parameter example returns 124329.00. The omitted format is represented by a placeholder comma; decimal appends two zeros as fractional digits to the value of x.

SET x=124329
WRITE $FNUMBER(x,",,",2)

The following 3-parameter example returns 0.78. The omitted format is represented by a placeholder comma; decimal rounds to 2 fractional digits; decimal also appends the integer 0, overriding the format default:

SET x=.7799
WRITE $FNUMBER(x,",,",2)

Notes

Decimal Separator

$FNUMBER uses the DecimalSeparator property value for the current locale ("." by default) as the delimiter character between the integer part and the fractional part of the returned number. When the "," format code is specified, this delimiter is a "," regardless of the current locale setting.

To determine the DecimalSeparator character for your locale, invoke the GetFormatItem() method:

WRITE #class(%SYS.NLS.Format).GetFormatItem("DecimalSeparator")
Numeric Group Separator and Size

When the format string includes “,” $FNUMBER uses the NumericGroupSeparator property value from the current locale as the delimiter between groups of digits in the integer part of the returned number. The size of these groups is determined by the NumericGroupSize property of the current locale.

The English language locale defaults to a comma (”,”) as the NumericGroupSeparator and 3 as the NumericGroupSize. Many European locales use a period (“.”) as the NumericGroupSeparator. The Russian (rusw), Ukrainian (ukrw), and Czech (csyw) locales use a blank space as the NumericGroupSeparator. The NumericGroupSize defaults to 3 for all locales, including Japanese. (Users of Japanese may wish to group integer digits in units of either 3 or 4, depending upon context.)

When the format string includes ‘.’ (and does not include “N”) $FNUMBER uses NumericGroupSeparator=”,“ and NumericGroupSize=3 to format the return value, regardless of your current locale settings.

To determine the NumericGroupSeparator character and NumericGroupSize number for your locale, invoke the GetFormatItem() method:

WRITE <<-##class(%SYS.NLS.Format).GetFormatItem("NumericGroupSeparator"),!
WRITE <<-##class(%SYS.NLS.Format).GetFormatItem("NumericGroupSize")

Plus Sign and Minus Sign

$FNUMBER uses the PlusSign and MinusSign property values for the current locale (“+” and “-” by default). When the “.” format code is specified, these signs are set to “+” and “-”, regardless of the current locale.

To determine the PlusSign and MinusSign characters for your locale, invoke the GetFormatItem() method:

WRITE <<-##class(%SYS.NLS.Format).GetFormatItem("PlusSign"),!
WRITE <<-##class(%SYS.NLS.Format).GetFormatItem("MinusSign")

Differences between $FNUMBER and $INUMBER

Most format codes have similar meanings in the $FNUMBER and $INUMBER functions, but the exact behavior triggered by each code differs by function because of the nature of the validations and conversions being performed.

In particular, the “-” and “+” format codes do not have quite the same meaning for $FNUMBER as they do for $INUMBER. With $FNUMBER, “-” and “+” are not mutually exclusive, and “-” only affects the MinusSign (by suppressing it), and “+” only affects the PlusSign (by inserting it). With $INUMBER, “-” and “+” are mutually exclusive. “-” means no sign is permitted, and “+” means there must be a sign.

See Also

- $DOUBLE function
- $JUSTIFY function
- $INUMBER function
- $ISVALIDNUM function
- $NORMALIZE function
- $NUMBER function
$GET

Returns the data value of a specified variable.

$GET(variable,default)
$G(variable,default)

Parameters

| variable | A local variable, global variable, or process-private global variable, subscripted or unsubscripted. The variable may be undefined. variable may be specified as a multidimensional object property with the syntax obj.property. |
| default | Optional — The value to be returned if the variable is undefined. If a variable, it must be defined. |

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.

Parameters

variable

The variable whose data value is to be returned.

- variable can be a local variable, a global variable, or a process-private global (PPG) variable. It can be subscripted or unsubscripted. It cannot be an ObjectScript special variable or a structured system variable (SSVN).

  The variable does not need to be a defined variable. $GET returns the null string for an undefined variable; it does not define the variable. A 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.

- variable can be a multidimensional object property; it cannot be a non-multidimensional object property. Attempting to use $GET on a non-multidimensional object property results in an <OBJECT DISPATCH> error.

- $GET cannot return a property value for a Proxy Object property. Caché instead issues a message that the specified property does not exist. This property access limitation is unique to the class %ZEN.proxyObject, which is defined in the InterSystems Class Reference.

default

The data value to be returned if variable is undefined. It can any expression, including a local variable or a global variable, either subscripted or unsubscripted. 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. If present, default resets the naked indicator, affecting future naked global references, as described below.
If default is an undefined variable, by default $GET issues an <UNDEFINED> error, even when variable is defined. You can change Caché behavior to not generate an <UNDEFINED> error when referencing an undefined variable by setting the %SYSTEM.Process.Undef() method. If the Undefined() method is set to not generate an <UNDEFINED> error, $GET returns variable when default is undefined.

You can specify an ObjectScript special variable as default. However, specifying $ZORDER may result in an <UNDEFINED> error, even when variable is defined.

**Examples**

In the following example, the variable test is defined and the variable xtest is undefined. (The ZWRITE command is used because it explicitly returns a null string value.)

```objectscript
KILL xtest
SET test="banana"
SET tdef=$GET(test),tundef=$GET(xtest)
ZWRITE tdef    ; $GET returned value of test
ZWRITE tundef ; $GET returned null string for xtest
WRITE !,$GET(xtest,"none") ; $GET returns default of "none" for undefined variable
```

In the following example, a multidimensional property is used as the variable value. This example returns the names of all defined namespaces:

```objectscript
SET obj = ##class(%ResultSet).%New("%SYS.Namespace:List")
DO obj.Execute()
WRITE !,$GET(obj.Data,"none") // returns "none"
SET x=1
WHILE x"="
   DO obj.Next()
   SET x=$GET(obj.Data("Nsp"))
   IF x"="
      WRITE !,"Namespace: ",x }
   }
WRITE !,"Done!"
```

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. For example, you can recode the following lines:

```objectscript
IF $DATA(^client(i))=10 {
   WRITE !!,"Name: No Data"
   GOTO Level1+3
}
```

as:

```objectscript
IF $GET(^client(i))="" {
   WRITE !!,"Name: No Data"
   GOTO Level1+3
}
```

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. For example, the lines:

```objectscript
IF $DATA(^client(i))=0 { QUIT }
ELSEIF $DATA(^client(i))=10 {
   WRITE !!,"Name: No Data"
   GOTO Level1+3
}
```

could not be re-coded as:
The two lines perform different actions depending on whether the array element is undefined or a downward pointer without data. If $GET were used here, only the first action (QUIT) would ever be performed. You could use $DATA for the first test and $GET for the second, but not the reverse ($GET for the first test and $DATA for the second).

### Defaults with $GET and $SELECT

$GET(variable,default) allows you to return a default value when a specified variable is undefined. The same operation can be performed using a $SELECT function.

However, unlike $SELECT, the second argument in $GET is always evaluated.

The fact that $GET always evaluates both of its arguments is significant if variable and default both make subscripted global references and thus both modify the naked indicator. Because the arguments are evaluated in left-to-right sequence, the naked indicator is set to the default global reference, regardless of the whether $GET returns the default value. For further details on using $GET with global variables and the naked indicator, see Using Multidimensional Storage ( Globals) in Using Caché Globals.

### 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.

You can define handling behavior for all undefined variables on a per-process basis using the Undefined() method of the %SYSTEM.Process class. The system-wide default behavior can be established by setting the Undefined property of the Config.Miscellaneous class. Setting Undefined has no effect on $GET or $DATA handling of specified variables.

### See Also

- $DATA function
- $SELECT function
- Using Multidimensional Storage (Globals) in Using Caché Globals
### $INCREMENT

Adds a specified increment to the numeric value of a variable.

```
$INCREMENT(variable,num)
$I(variable,num)
```

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>The variable whose value is to be incremented. It can specify a local variable, a process-private global, or a global variable and can be either subscripted or unsubscripted. The variable need not be defined. If the variable is not defined, or is set to the null string (&quot;&quot;), $INCREMENT treats it as having an initial value of zero and increments accordingly. A literal value cannot be specified here. You cannot specify a simple object property reference as <code>variable</code>; you can specify a multidimensional property reference as <code>variable</code> with the syntax <code>obj.property</code>.</td>
</tr>
<tr>
<td>num</td>
<td>Optional — The numeric increment you want to add to <code>variable</code>. The value can be a number (integer or non-integer, positive or negative), a string containing a number, or any expression which evaluates to a number. Leading and trailing blanks and multiple signs are evaluated. A string is evaluated until the first nonnumeric character is encountered. The null string (&quot;&quot;&quot;) is evaluated as zero. If you do not specify <code>num</code> for the second argument, Caché defaults to incrementing <code>variable</code> by 1.</td>
</tr>
</tbody>
</table>

#### Description

$INCREMENT resets the value of a variable by adding a specified increment to the existing value of the variable and returning the incremented value. This is shown in the following example:

```ObjectScript
SET a=7
SET result=$INCREMENT(a)
WRITE !,result /* result is 8 (a+1) */
WRITE !,a /* variable a is also now 8 */
```

You can use the $GET function to return the current value of a variable.

$INCREMENT performs this increment as an atomic operation, which does not require the use of the LOCK command.

If multiple processes simultaneously increment the same global through $INCREMENT, each process receives a unique, increasing number (or decreasing number if num is negative). In some situations, certain numbers may be skipped due to timing issues. For further details on using $INCREMENT with global variables, see Using Multidimensional Storage (Globals) in Using Caché Globals.

Caché does not restore the original, non-incremented value if $INCREMENT is in a transaction that is rolled back.

$INCREMENT and $ZINCREMENT have the same syntax and effects. You can use $ZINCREMENT in any situation in which you would use $INCREMENT.

#### Parameters

- **variable**

  The variable whose data value is to be incremented. It must be a variable, it cannot be a literal. The variable does not need to be defined. $INCREMENT defines an undefined variable, setting its value to `num` (1, by default).
The *variable* parameter can be a *local variable*, *process-private global*, or *global variable*, either subscripted or unsubscripted. 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*.

The *variable* parameter can be a *multidimensional property* reference. For example, `$INCREMENT(..Count)`. It cannot be a non-multidimensional object property. Attempting to increment a non-multidimensional object property results in an `<OBJECT DISPATCH>` error.

*$INCREMENT* cannot increment *special variables*, even those that can be modified using *SET*. Attempting to increment a special variable results in a `<SYNTAX>` error.

*num*

The amount to increment (or decrement) by. The *num* parameter can be a positive number, incrementing the value of *variable*, or a negative number, decrementing the value of *variable*. It can be an integer or a fractional number. *num* can be zero (no increment). A numeric string is treated as a number. An empty string (""") or a non-numeric string is treated as an increment of zero. If you do not specify an increment, Caché uses the default increment of one (1).

**$INCREMENT or $SEQUENCE**

*$SEQUENCE* and *$INCREMENT* can be used as alternatives, or can be used in combination with each other. *$SEQUENCE* is intended specifically for integer increment operations involving multiple simultaneous processes. *$INCREMENT* is a more general increment/decrement function.

- *$SEQUENCE* increments global variables. *$INCREMENT* increments local variables, global variables, or process-private globals.
- *$SEQUENCE* increments an integer by 1. *$INCREMENT* increments or decrements any numeric value by any specified numeric value.
- *$SEQUENCE* can allocate a range of increments to a process. *$INCREMENT* allocates only a single increment.
- *SET* $SEQUENCE can be used to change or undefine (kill) a global. *$INCREMENT* cannot be used on the left side of the *SET* command.

**$INCREMENT and Global Variables**

You can use *$INCREMENT* on a global variable or a subscript node of a global variable. You can access a global variable mapped to another namespace using an *extended global reference*. You can access a subscripted global variable using a *naked global reference*.

Caché evaluates parameters in left-to-right order. If *num* (the amount to increment) is a subscripted global, Caché uses this global reference to set the naked indicator, affecting all subsequent naked global references.

**Incrementing Strings**

*$INCREMENT* is generally used for incrementing a variable containing a numeric value. However, it does accept a *variable* containing a string. The following rules apply when using *$INCREMENT* on a string:

- A null string (""") is treated as having a value of zero.
- A numeric string ("123" or "+0012.30") is treated as having that numeric value. The string is converted to canonical form: leading and trailing zeros and the plus sign are removed.
- A mixed numeric/nonnumeric string ("12AB" or "1,000") is treated as the numeric value up to the first nonnumeric character and then truncated at that point. (Note that a comma is a nonnumeric character.) The resulting numeric substring is converted to canonical form: leading and trailing zeros and the plus sign are removed.
- A nonnumeric string ("ABC" or "$12") is treated as having a value of zero.
• Scientific notation conversion is performed. For example, if \texttt{strvar=}"3E2", \texttt{$INCREMENT}$ treats it as having a value of 300.

• Arithmetic operations are not performed. For example, if \texttt{strvar=}"3+7", \texttt{$INCREMENT$} will truncate the string at the plus sign (treating it as a nonnumeric character) and increment \texttt{strvar} to 4.

• Multiple uses of a string variable in a single \texttt{$INCREMENT$} statement should be avoided. For example, avoid concatenating a string variable to the increment of that variable: \texttt{strvar \_ $INCREMENT$ (strvar)}. This returns unpredictable results.

\textbf{Failure to Increment}

If \texttt{$INCREMENT$} cannot increment \texttt{variable}, it issues a \texttt{<MAXINCREMENT>} error. This only occurs when the \texttt{num} increment value is extremely small, and/or the \texttt{variable} value is extremely large.

An increment by zero (\texttt{num=}0) always returns the original number, regardless of its size. It does not issue a \texttt{<MAXINCREMENT>} error.

\texttt{<MAXINCREMENT>} occurs when the numeric types of the parameters differ and the resulting type conversion and rounding would result in no increment. If you use \texttt{$INCREMENT$} on a very large number, the default increment of 1 (or some other small positive or negative value of \texttt{num}) is too small to be significant. Similarly, if you specify a very small fractional \texttt{num} value, its value is too small to be significant. Rather than returning the original \texttt{variable} number without incrementing it, \texttt{$INCREMENT$} generates a \texttt{<MAXINCREMENT>} error.

In the following example, 1.2E18 is a number that can be incremented or decremented by 1; 1.2E20 is a number that is too large to be incremented or decremented by 1. The first three \texttt{$INCREMENT$} functions successfully increment or decrement the number 1.2E18. The fourth and fifth \texttt{$INCREMENT$} functions increment by zero, and so always return the original number unchanged, regardless of the size of the original number. The sixth and seventh \texttt{$INCREMENT$} functions provide a \texttt{num} increment sufficiently large to successfully increment or decrement the number 1.2E20. The eighth \texttt{$INCREMENT$} function attempts to increment 1.2E20 by 1, and thus generates a \texttt{<MAXINCREMENT>} error.

\begin{verbatim}
SET x=1.2E18
WRITE "E18      :",x,"!
WRITE "E18+1    :",$INCREMENT(x),"!
WRITE "E18+4    :",$INCREMENT(x,4),"!
WRITE "E18-6    :",$INCREMENT(x,-6),"!
WRITE "E18+0    :",$INCREMENT(x,0),"!
SET y=1.2E20
WRITE "E20      :",y,"!
WRITE "E20+0    :",$INCREMENT(y,0),"!
WRITE "E20-10000:",$INCREMENT(y,-10000),"!
WRITE "E20+10000:",$INCREMENT(y,10000),"!
WRITE "E20+1    :",$INCREMENT(y),"!
\end{verbatim}

\textbf{Locking and Simultaneous Global Increments}

\texttt{$INCREMENT$} and \texttt{$ZINCREMENT$} do not perform a lock operation when they increment \texttt{variable} nor does using the \texttt{LOCK} command on \texttt{variable} prevent \texttt{$INCREMENT$} from incrementing or decrementing its value. For example, suppose process 1 executes a lock on \texttt{^COUNTER}:

\begin{verbatim}
    LOCK ^COUNTER
\end{verbatim}

Then suppose, process 2 increments \texttt{^COUNTER}:

\begin{verbatim}
    SET x=$INCREMENT (^COUNTER, VAL)
\end{verbatim}

Process 2 is not prevented from incrementing \texttt{^COUNTER} by the lock held by process 1.

The two processes are not guaranteed their own unique \texttt{^COUNTER} values unless both are using \texttt{$INCREMENT$}.

\texttt{$INCREMENT$}
$INCREMENT and Transaction Processing

The common usage for $INCREMENT is to increment a counter before adding a new entry to a database. $INCREMENT provides a way to do this very quickly, avoiding the use of the LOCK command.

The trade off for this is that the counter is not locked. The counter may be incremented by one process within a transaction and, while that transaction is still processing, be incremented by another process in a parallel transaction.

In the event either transaction (or any other transaction that uses $INCREMENT) must be rolled back (with the TROLLBACK command), counter increments are ignored. The counter variables are not decremented since it is not clear whether the resulting counter value would be valid. In all likelihood, such a rollback would be disastrous for other transactions.

For further details on using $INCREMENT in a distributed database environment, refer to “The $INCREMENT Function and Application Counters” in the Developing Distributed Applications chapter of the Caché Distributed Data Management Guide.

Examples

The following example increments the value of myvar by n. Note that myvar does not have to be a prior defined variable:

```
SET n=4
KILL myvar
SET VAL=$INCREMENT(myvar,n)       ; returns 4
WRITE !,myvar
SET VAL=$INCREMENT(myvar,n)       ; returns 8
WRITE !,myvar
SET VAL=$INCREMENT(myvar,n)       ; returns 12
WRITE !,myvar
```

The following example adds incremental values to the process-private variable ^||xyz using $INCREMENT. The one-argument form of $INCREMENT increments by 1; the two-argument form increments by the value specified in the second argument. In this case, the second argument is a non-integer value.

```
KILL ^||xyz
WRITE ,,$INCREMENT(^||xyz)       ; returns 1
WRITE ,,$INCREMENT(^||xyz)       ; returns 2
WRITE ,,$INCREMENT(^||xyz)       ; returns 3
WRITE ,,$INCREMENT(^||xyz,3.14)  ; returns 6.14
```

The following example shows the effects of incrementing by zero (0) and incrementing by a negative number:

```
KILL xyz
WRITE ,,$INCREMENT(xyz,0)  ; initialized as zero
WRITE ,,$INCREMENT(xyz,0)  ; still zero
WRITE ,,$INCREMENT(xyz)    ; increments by 1 (default)
WRITE ,,$INCREMENT(xyz,1)   ; increments by 1 (=2)
WRITE ,,$INCREMENT(xyz,-1)  ; decrements by -1 (=1)
WRITE ,,$INCREMENT(xyz,-1)  ; decrements by -1 (=0)
WRITE ,,$INCREMENT(xyz,-1)  ; decrements by -1 (=0)
```

The following example shows the effects of incrementing using mixed (numeric and nonnumeric) num strings and the null string:

```
KILL xyz
WRITE ,,$INCREMENT(xyz,"")    ; null string initializes to 0
WRITE ,,$INCREMENT(xyz,2)     ; increments by 2
WRITE ,,$INCREMENT(xyz,"")    ; null string increments by 0 (xyz=2)
WRITE ,,$INCREMENT(xyz,"3A4") ; increments by 3 (rest of string ignored)
WRITE ,,$INCREMENT(xyz,"A4")  ; nonnumeric string evaluates as zero (xyz=5)
WRITE ,,$INCREMENT(xyz,"1E2") ; increments by 100 (scientific notation)
```
See Also

- $SEQUENCE function
- $ZINCREMENT function
- $GET function
- TROLLBACK command
- Using ObjectScript for Transaction Processing in Using Caché ObjectScript
### $INUMBER

Validates a numeric value and converts it to internal format.

$INUMBER(fnumber, format, erropt)

\[\$IN(fnumber, format, erropt)\]

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fnumber</td>
<td>The numeric value to be converted to the internal format. It can be a numeric or string value, a variable name, or any valid ObjectScript expression.</td>
</tr>
<tr>
<td>format</td>
<td>A format specification indicating which external numeric formats are valid representations of numbers. Specified as a quoted string consisting of zero or more format codes, in any order. Format codes are described below. Note that some format codes are incompatible and result in an error. For default formatting, with or without the erropt parameter, you can specify the empty string (&quot;&quot;).</td>
</tr>
<tr>
<td>erropt</td>
<td>Optional — The expression returned if fnumber is considered invalid based on format.</td>
</tr>
</tbody>
</table>

#### Description

The $INUMBER function validates the numeric value fnumber using the formats specified in format. It then converts it to the internal Caché format.

If fnumber does not correspond to the specified format and you have not specified erropt, the system generates an <ILLEGAL VALUE> error. If you have specified erropt, an invalid numeric value returns the erropt string.

#### Parameters

**format**

The possible format codes are as follows. You can specify them singly or in combination to instruct $INUMBER to adhere strictly to the format rules. If no format codes are entered, $INUMBER will be as flexible as possible in validating fnumber (see Null Format Provides Maximum Flexibility for more information).
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Mandatory sign. The <code>fnumber</code> value must have an explicit sign. Even the number 0 must be signed (+0 or -0). The sign can be either leading or trailing, unless restricted by either an “L” or a “T” format code. Parentheses cannot be used. The only value that does not require a sign is NAN, which can be specified with or without a sign when using code “D+”.</td>
</tr>
<tr>
<td>-</td>
<td>Unsigned. No sign may be present in <code>fnumber</code>.</td>
</tr>
<tr>
<td>D</td>
<td>$DOUBLE numbers. This code converts <code>fnumber</code> to an IEEE floating point number. This is equivalent to <code>$DOUBLE(fnumber)</code>. If “D” is specified, you can input the quoted strings “INF” and “NAN” as an <code>fnumber</code> value. INF and NAN can be specified in any combination of uppercase and lowercase letters, with or without leading or trailing signs or enclosing parentheses. (Signs are accepted, but ignored, for NAN.) The variant forms INFINITY and SNAN are also supported.</td>
</tr>
<tr>
<td>E or G</td>
<td>E-notation (scientific notation). This code allows you to specify <code>fnumber</code> as a string in scientific notation format. This code permits, but does not require, that you specify <code>fnumber</code> in scientific notation.</td>
</tr>
<tr>
<td>N</td>
<td>No NumericGroupSeparator. Does not allow the use of a numeric group separator. This format code is incompatible with the comma (,) format code.</td>
</tr>
<tr>
<td>O</td>
<td>ODBC locale. Overrides the current locale, and instead uses the standard ODBC locale with the following values: PlusSign=+; MinusSign=-; DecimalSeparator=.; NumericGroupSeparator=,; NumericGroupSize=3. This format code is incompatible with the dot (.) format code.</td>
</tr>
<tr>
<td>P</td>
<td>Negative numbers must be enclosed in parentheses. Nonnegative numbers must be unsigned, and may have or omit leading and trailing spaces.</td>
</tr>
<tr>
<td>L</td>
<td>Leading sign. Sign, if present, must precede the numerical portion of <code>fnumber</code>. Parentheses are not permitted.</td>
</tr>
<tr>
<td>T</td>
<td>Trailing sign. Sign, if present, must follow the numerical portion of <code>fnumber</code>. Parentheses are not permitted.</td>
</tr>
<tr>
<td>,</td>
<td>Expects <code>fnumber</code> to use the format specified by properties in the current locale. The NumericGroupSeparator (&quot;_&quot; by default) may or may not appear in <code>fnumber</code>, but if present, it must consistently appear every NumericGroupSize (3 by default) digits to the left of the decimal point.</td>
</tr>
<tr>
<td>.</td>
<td>Requires standard European formatting, regardless of the current locale settings. Requires DecimalSeparator as comma (,), NumericGroupSeparator as period (.), NumericGroupSize as 3, PlusSign as plus (+), MinusSign as minus (-). Periods are optional, but if present must consistently appear every three digits to the left of the decimal comma.</td>
</tr>
</tbody>
</table>

**When “+”, “-” and “P” Format Codes are Absent**

When `format` does not include any of the “+”, “-”, or “P” codes, then `fnumber` may contain any one of the following:

- No sign or parentheses.
- Either the PlusSign locale property (“+” by default) or the MinusSign locale property (“-” by default) but not both. The position of this sign is determined by the “L” or the “T” format code if specified.
- Leading and trailing parentheses.
When “L”, “T” and “P” Format Codes are Absent

When format does not include any of the “L”, “T”, or “P” format codes, any sign present in fnumber may be either leading or trailing (but not both).

When “,” and “.” Format Codes are Absent

When format does not include either the “,” or “.” format codes, fnumber may optionally have NumericGroupSeparator symbols appear anywhere to the left or right of the DecimalSeparator, if any. However, each NumericGroupSeparator must have at least one digit to its immediate left and one to its immediate right. When format includes “N”, no NumericGroupSeparator symbols are permitted.

Mutually Exclusive Format Codes

Some format codes conflict with each other. Each of the following pairs of format codes are mutually exclusive and result in an error:

- “-+” results in a <FUNCTION> error
- “-P” or “+P” result in a <SYNTAX> error
- “TP” or “LP” result in a <SYNTAX> error
- “TL” results in a <FUNCTION> error
- “.” results in a <FUNCTION> error
- “.N” results in a <FUNCTION> error
- “.O” results in a <FUNCTION> error

A <FUNCTION> error is also generated if you specify an invalid format code character.

Null Format Provides Maximum Flexibility

You can specify format as a null string. This is called a null format. When a null format is specified, $INUMBER accepts a fnumber value with any one of the following sign conventions:

- No sign or parentheses.
- Either a leading or trailing MinusSign, but not both.
- Either a leading or trailing PlusSign, but not both.
- Leading and trailing parentheses.

When a null format is specified, fnumber may optionally have NumericGroupSeparator symbols appear anywhere to the left or right of the DecimalSeparator, if any. However, each NumericGroupSeparator must have at least one digit to its immediate left and one to its immediate right. Sign rules are flexible, and leading and trailing blanks and zeros are ignored. Thus, the following two commands:

```
WRITE !,$INUMBER("+1,23,456,7.8,9,100","")
WRITE !,$INUMBER("0012,3456,7.891+","")
```

are both valid and return the same number, formatted according to the default locale. However,

```
WRITE $INUMBER("1,23,,345,7.,8,9,,","")
```

is invalid because of the adjacent commas, the adjacent period and comma, and the trailing comma. It generates an <ILLEGAL VALUE> error.
Behavior Common to All Formats

Regardless of the specified format codes, $INUMBER always ignores leading and trailing blank spaces or zeros, but considers $fnumber to be invalid if it has any of the following characteristics:

- Both a PlusSign and a MinusSign
- More than one PlusSign or MinusSign
- Parentheses and a PlusSign
- Parentheses and a MinusSign
- More than one DecimalSeparator
- Embedded Spaces
- Any characters other than the following:
  - Numeric digits
  - "(" or ")"
  - Leading or trailing spaces
  - The DecimalSeparator specified by the current locale (if format does not include ".")
  - The NumericGroupSeparator specified by the current locale (if format does not include ".")
  - The PlusSign property specified by the current locale (if format does not include ".")
  - The MinusSign property specified by the current locale (if format does not include ".")
  - "." (if format includes ".")
  - "," (if format includes ",")
  - "+" (if format includes "+")
  - "-" (if format includes "-")
- The strings "INF" and "NAN" (and their variants) if format includes "D".

Examples

These examples illustrate how different formats affect the behavior of $INUMBER. All of these examples assume the current locale is the default locale.

In the following example, $INUMBER accepts a leading minus sign because of the “L” format code and returns -123456789.12345678:

```objectscript
WRITE $INUMBER("-123,4,56,789.1234,5678","L")
```

In the following example, $INUMBER generates an <ILLEGAL VALUE> error because the sign is leading but the “T” format code specifies that trailing signs must be used:

```objectscript
WRITE $INUMBER("-123,4,56,789.1234,5678","T")
```

In the following example, the first $INUMBER succeeds and returns a negative number. The second $INUMBER generates an <ILLEGAL VALUE> error because $fnumber includes a sign but the “P” format code specifies that negative numbers must be enclosed in parentheses rather than signed:

```objectscript
WRITE !,$INUMBER("(123,4,56,789.1234,5678)","P")
WRITE !,$INUMBER("-123,4,56,789.1234,5678","P")
```
In the following example, $INUMBER generates an <ILLEGAL VALUE> error because a sign is present but the “−” format code specifies that numbers must be unsigned:

```
WRITE $INUMBER("-123,4,56,789.1234,5678","-")
```

In the following example, $INUMBER fails but does not generate an error due to the illegal use of a sign, but instead returns as its value the string “ERR” specified as the erropt:

```
WRITE $INUMBER("-123,4,56,789.1234,5678", ", " , "ERR")
```

The following example returns -23456789.123456789; $INUMBER accepts the specified fnumber as valid because the leading sign follows the formatting specified by “L” and the strict spacing of commas every three digits to the left of the decimal place with no commas to its right follows the strict formatting specified by the “,” code:

```
WRITE $INUMBER("-23,456,789.123456789","L,")
```

In the following example, the “E” code permits conversion of a scientific notation string to a number. Note that all format codes support scientific notation as a numeric literal, but only “E” (or “G”) support scientific notation as a string. This example uses variables and concatenation to provide the scientific notation string values:

```
SET num=1.234
SET exp=-14
WRITE $INUMBER(1.234E-14,"E","E-lit-err"),!
WRITE $INUMBER(num_"E"_exp,"E","E-string-err"),!
WRITE $INUMBER(1.234E-14,"L","L-lit-err"),!
WRITE $INUMBER(num_"E"_exp,"L","L-string-err"),!
```

The following example compares the values returned by “L” code and a “D” code for a fractional number and for the constant pi. The “D” code converts to an IEEE floating point ($DOUBLE) number:

```
WRITE $INUMBER(1.23E-23,"L"),!
WRITE $INUMBER(1.23E-23,"D"),!
WRITE $INUMBER($ZPI,"L"),!
WRITE $INUMBER($ZPI,"D"),!
```

### Notes

#### Differences between $INUMBER and $FNUMBER

Most format codes have similar meanings in the $INUMBER and $FNUMBER functions, but the exact behavior triggered by each code differs by function because of the nature of the validations and conversions being performed.

In particular, the “−” and “+” format codes do not have quite the same meaning for $INUMBER as they do for $FNUMBER. With $FNUMBER, “−” and “+” are not mutually exclusive, and “−” only affects the MinusSign (by suppressing it), and “+” only affects the PlusSign (by inserting it). With $INUMBER, “−” and “+” are mutually exclusive. “−” means no sign is permitted, and “+” means there must be a sign.

#### Decimal Separator

$INUMBER uses the DecimalSeparator property value for the current locale (“.” by default) as the delimiter character between the integer part and the fractional part of fnumber. When the “." format code is specified, this delimiter is a “,” regardless of the current locale.

To determine the DecimalSeparator character for your locale, invoke the GetFormatItem() method:

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DecimalSeparator")
```

#### Numeric Group Separator and Size

$INUMBER uses the NumericGroupSeparator property value from the current locale (“.” by default) as the delimiter between groups of digits in the integer part of fnumber. The size of these groups is determined by the NumericGroupSize
property of the current locale (‘3’ by default). When the ‘.’ format code is specified, this delimiter is a ‘.’ and appears every three digits regardless of the current locale.

To determine the NumericGroupSeparator character and NumericGroupSize number for your locale, invoke the GetFormatItem() method:

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("NumericGroupSeparator"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("NumericGroupSize")
```

**Plus Sign and Minus Sign**

$NUMBER uses the PlusSign and MinusSign property values from the current locale (‘+’ and ‘-’ by default). When the ‘.’ format code is specified, these signs are set to ‘+’ and ‘-’, regardless of the current locale.

To determine the PlusSign and MinusSign characters for your locale, invoke the GetFormatItem() method:

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("PlusSign"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("MinusSign")
```

**See Also**

- $DOUBLE function
- $NUMBER function
- $ISVALIDNUM function
- $NORMALIZE function
- $NUMBER function
- More information on locales in the section on “System Classes for National Language Support” in *Caché Specialized System Tools and Utilities*. 
$ISOBJECT

Returns whether an expression is an object reference (OREF).

\[ \text{$ISOBJECT(expr)$} \]

Parameter

| expr | A ObjectScript expression. |

Description

$ISOBJECT returns 1 if \( expr \) is an object reference (OREF). $ISOBJECT returns 0 if \( expr \) is not an object reference (OREF).

$ISOBJECT returns –1 if \( expr \) is a reference to an invalid object. Invalid objects should not occur in normal operations; an invalid object could be caused, for example, by recompiling the class while instances of the class are active.

To remove an object reference, set the variable to the null string (“”). The obsolete %Close() method cannot be used to remove an object reference. %Close() performs no operation and always returns successful completion. Do not use %Close() when writing new code.

For information on OREFs, see “OREF Basics” in Using Caché Objects.

Parameter

expr

Any ObjectScript expression.

Examples

The following example shows the values returned by $ISOBJECT for an object reference and a non-object reference (in this case, a string reference):

```
SET a="certainly not an object"
SET o=##class(%SQL.Statement).%New()
WRITE !,"non-object a: ",$ISOBJECT(a)
WRITE !,"object ref o: ",$ISOBJECT(o)
```

The following example shows that JSON values are object references:

```
SET a=["apple","banana","orange"]
SET b={"fruit":"orange","color":"orange"}
WRITE !,"JSON array: ",$ISOBJECT(a)
WRITE !,"JSON object: ",$ISOBJECT(b)
```

The following Dynamic SQL example shows that a stream field is an OID, not an object reference. You need to use the SQL %OBJECT function to return the object reference:

```
ZSPACE "SAMPLES"
SET myquery=2
SET myquery1="SELECT TOP 1 Name,Notes,%OBJECT(Notes) AS NoteObj "
SET myquery2="FROM Sample.Employee WHERE Notes IS NOT NULL"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery1)
IF qStatus=0 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WHILE rset.%Next() {
  WRITE "Stream field oid: ",$ISOBJECT(rset.Notes),!
  WRITE "Stream field oref: ",$ISOBJECT(rset.NoteObj),!
}
The following example shows how to remove an object reference. The %Close() method does not change the object reference. Setting an object reference to the null string deletes the object reference:

```ObjectScript
SET o=#class(%SQL.Statement).%New()
WRITE !,"objref o: ",$ISOBJECT(o)
DO o.%Close() ; this is a no-op
WRITE !,"objref o: ",$ISOBJECT(o)
SET o=""
WRITE !,"objref o: ",$ISOBJECT(o)
```

**See Also**

- `$SYSTEM` special variable
**$ISVALIDDOUBLE**

Validates a $DOUBLE numeric value and returns a boolean; optionally provides range checking.

### $ISVALIDDOUBLE (num, scale, min, max)

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>num</strong></td>
<td>The numeric value to be validated. It can be a numeric or string value, a variable name, or any valid ObjectScript expression. If a valid number, <em>num</em> is converted to a IEEE double-precision floating point type.</td>
</tr>
<tr>
<td><strong>scale</strong></td>
<td>Optional — The number of significant decimal digits for <em>min</em> and <em>max</em> range comparisons.</td>
</tr>
<tr>
<td><strong>min</strong></td>
<td>Optional — The minimum permitted numeric value. The value you supply is converted to a IEEE double-precision floating point type. If not specified, <em>min</em> defaults to $DOUBLE(&quot;-INF&quot;).</td>
</tr>
<tr>
<td><strong>max</strong></td>
<td>Optional — The maximum permitted numeric value. The value you supply is converted to a IEEE double-precision floating point type. If not specified, <em>max</em> defaults to $DOUBLE(&quot;INF&quot;).</td>
</tr>
</tbody>
</table>

#### Description

The $ISVALIDDOUBLE function determines whether *num* passes validation tests for an IEEE double-precision floating point number and returns a boolean value. It optionally performs a range check using *min* and *max* values, which are automatically converted to IEEE numbers. The *scale* parameter is used during range checking to specify how many fractional digits to compare. A boolean value of 1 means that *num* is a properly formed IEEE double-precision number value and passes the range check, if one is specified. $ISVALIDDOUBLE is a validation function; it does not determine if *num* is a number of data type DOUBLE, or if it has been generated using the $DOUBLE function.

$ISVALIDDOUBLE validates American format numbers, which use a period (.) as the decimal separator. It does not validate European format numbers, which use a comma (,) as the decimal separator. $ISVALIDDOUBLE does not consider valid a number that contains numeric group separators; it returns 0 (invalid) for any number containing a comma or a blank space, regardless of the current locale.

#### Parameters

**num**

The number to be validated may be an integer, a fractional number, a number in scientific notation (with the letter “E” or “e”). It may be a string, expression, or variable that resolves to a number. It may be signed or unsigned, and may contain leading or trailing zeros.

ObjectScript converts a number to canonical form before supplying it to $ISVALIDDOUBLE for validation. Therefore, any arithmetic expression, numeric concatenation, or multiple leading + and – signs are resolved prior to evaluation by the function. ObjectScript does not convert a numeric string to canonical form prior to evaluation. However, prefacing a numeric string with a + sign forces numeric evaluation (and thus canonical conversion) of a numeric string.

Validation fails ($ISVALIDDOUBLE returns 0) if:

- *num* contains any characters other than the digits 0–9, a leading + or – sign, a decimal point (.), and a letter “E” or “e”.
- *num* contains more than one + or – sign, decimal point, or letter “E” or “e”. If *num* is a number, ObjectScript resolves multiple leading + and – signs. If *num* is a numeric string, it cannot contain more than one leading + or – sign.
- The optional + or – sign is not the first character of *num*.
- The letter “E” or “e” indicating a base-10 exponent is not followed by an integer in a numeric string. With a number, “E” not followed by an integer results in a $SYNTAX error.
num is the null string.

If num is INF (with or without a + or – sign), it is a valid $DOUBLE number; the boolean value returned by $ISVALIDDOUBLE depends on whether num passes the specified range check.

If num is NAN $ISVALIDDOUBLE returns 1.

scale

The scale parameter is used during range checking to specify how many fractional digits to compare. Specify an integer value for scale; any fractional digits in the scale value are ignored. You can specify a scale value larger than the number of fractional digits specified in the other parameters. You can specify a scale value of –1; all other negative scale values result in a <FUNCTION> error.

A nonnegative scale value causes num to be rounded to that number of fractional digits before performing min and max range checking. A scale value of 0 causes num to be rounded to an integer value (3.9 = 4) before performing range checking. A scale value of –1 causes num to be truncated to an integer value (3.9 = 3) before performing range checking. To compare all specified digits without rounding or truncating, omit the scale parameter. A scale value that is nonnumeric or the null string is equivalent to a scale value of 0.

Rounding is performed for all scale values except –1. A value of 5 or greater is always rounded up.

The scale parameter value causes evaluation using rounded or truncated versions of the num value. The actual value of the num variable is not changed by $ISVALIDDOUBLE processing.

If you omit the scale parameter, retain the comma as a place holder.

min and max

You can specify a minimum allowed value, a maximum allowed value, neither, or both. If specified, the num value (after the scale operation) must be greater than or equal to the min value, and less than or equal to the max value. The values are converted to IEEE floating point numbers before being used for range checking. A null string as a min or max value is equal to zero. If a value does not meet these criteria, $ISVALIDDOUBLE returns 0.

The NAN value is always valid, regardless of the min or max value.

If you omit a parameter, retain the comma as a place holder. For example, when omitting scale and specifying min or max, or when omitting min and specifying max. Trailing commas are ignored.

Examples

In the following example, each invocation of $ISVALIDDOUBLE returns 1 (valid number):

```
WRITE 1,$ISVALIDDOUBLE(0) ; All integers OK
WRITE 1,$ISVALIDDOUBLE(4.567) ; Fractional numbers OK
WRITE 1,$ISVALIDDOUBLE("4.567") ; Numeric strings OK
WRITE 1,$ISVALIDDOUBLE(-.0) ; Signed numbers OK
WRITE 1,$ISVALIDDOUBLE(+004.500) ; Leading/trailing zeroes OK
WRITE 1,$ISVALIDDOUBLE(4E2) ; Scientific notation OK
```

In the following example, each invocation of $ISVALIDDOUBLE returns 0 (invalid number):

```
WRITE 1,$ISVALIDDOUBLE("") ; Null string is invalid
WRITE 1,$ISVALIDDOUBLE("4,567") ; Commas are not permitted
WRITE 1,$ISVALIDDOUBLE("4A") ; Invalid character
WRITE 1,$ISVALIDDOUBLE("+-123") ; Multiple signs not resolved for strings
```

In the following example, each invocation of $ISVALIDDOUBLE returns 1 (valid number), even though INF (infinity) and NAN (Not A Number) are, strictly speaking, not numbers:

```
WRITE 1,$ISVALIDDOUBLE($DOUBLE($ZPI)) ; DOUBLE numbers OK
WRITE 1,$ISVALIDDOUBLE($DOUBLE("INF")) ; DOUBLE INF OK
WRITE 1,$ISVALIDDOUBLE($DOUBLE("NAN")) ; DOUBLE NAN OK
```
In the following example, specifying a min value eliminates -INF but not INF:

```objectscript
WRITE !,$ISVALIDDOUBLE($DOUBLE("-INF"),,99999999999)
WRITE !,$ISVALIDDOUBLE($DOUBLE("INF"),,99999999999)
```

The following example shows the use of the min and max parameters. All of the following return 1 (number is valid and also passes the range check):

```objectscript
WRITE !,$ISVALIDDOUBLE(4,,3,5)  ; scale can be omitted
WRITE !,$ISVALIDDOUBLE(4,2,3,5) ; scale can be larger than
                                 ; number of fractional digits
WRITE !,$ISVALIDDOUBLE(4,0,,5)  ; min or max can be omitted
WRITE !,$ISVALIDDOUBLE(4,0,4,4) ; min and max are inclusive
WRITE !,$ISVALIDDOUBLE(-4,0,-5,5) ; negative numbers
WRITE !,$ISVALIDDOUBLE(4.00,2,04,05) ; leading/trailing zeros
WRITE !,$ISVALIDDOUBLE(.4E3,0,3E2,400) ; base-10 exponents expanded
```

The following example shows the use of the scale parameter with min and max. All of the following return 1 (number is valid and also passes the range check):

```objectscript
WRITE !,$ISVALIDDOUBLE(4.55,,4.54,4.551)
    ; When scale is omitted, all digits of num are checked.
WRITE !,$ISVALIDDOUBLE(4.1,0,4,4.01)
    ; When scale=0, num is rounded to an integer value
    ; (0 fractional digits) before min & max check.
WRITE !,$ISVALIDDOUBLE(3.85,1,3.9,5)
    ; num is rounded to 1 fractional digit,
    ; (with values of 5 or greater rounded up)
    ; before min check.
WRITE !,$ISVALIDDOUBLE(4.01,17,3,5)
    ; scale can be larger than number of fractional digits.
WRITE !,$ISVALIDDOUBLE(3.9,-1,2,3)
    ; When scale=-1, num is truncated to an integer value
```

Notes

**$ISVALIDDOUBLE and $ISVALIDNUM Compared**

The **$ISVALIDDOUBLE** and **$ISVALIDNUM** functions both validate American format numbers and return a boolean value (0 or 1).

- Both functions accept as valid numbers the INF, –INF, and NAN values returned by **$DOUBLE**. **$ISVALIDDOUBLE** also accepts as valid numbers the not case-sensitive strings “NAN” and “INF”, as well as the variants “Infinity” and “sNAN”, and any of these strings beginning with a single plus or minus sign. **$ISVALIDNUM** rejects all of these strings as invalid, and returns 0.

```objectscript
WRITE !,$ISVALIDNUM($DOUBLE("NAN"))    ; returns 1
WRITE !,$ISVALIDDOUBLE($DOUBLE("NAN")) ; returns 1
WRITE !,$ISVALIDNUM("NAN")             ; returns 0
WRITE !,$ISVALIDDOUBLE("NAN")          ; returns 1
```

- Both functions parse signed and unsigned integers (including –0), scientific notation numbers (with “E” or “e”), real numbers (123.45) and numeric strings (“123.45”).

- Neither function recognizes the European DecimalSeparator character (comma (,)) or the NumericGroupSeparator character (American format: comma (,); European format: period (.)). For example, both reject the string “123,456” as an invalid number, regardless of the current locale setting.

- Both functions parse multiple leading signs (+ and –) for numbers. Neither accepts multiple leading signs in a quoted numeric string.

See Also

- **$DOUBLE function**
- **$FNUMBER function**
• $NUMBER function
• $ISVALIDNUM function
• $NORMALIZE function
• $NUMBER function
• More information on locales in the section on “System Classes for National Language Support” in Caché Specialized System Tools and Utilities.
$ISVALIDNUM

Validates a numeric value and returns a boolean; optionally provides range checking.

$ISVALIDNUM(num,scale,min,max)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>The numeric value to be validated. It can be a numeric or string value, a variable name, or any valid ObjectScript expression.</td>
</tr>
<tr>
<td>scale</td>
<td>Optional — The number of significant fractional digits for min and max range comparisons.</td>
</tr>
<tr>
<td>min</td>
<td>Optional — The minimum permitted numeric value.</td>
</tr>
<tr>
<td>max</td>
<td>Optional — The maximum permitted numeric value.</td>
</tr>
</tbody>
</table>

**Description**

The $ISVALIDNUM function validates *num* and returns a boolean value. It optionally performs a range check using *min* and *max* values. The *scale* parameter is used during range checking to specify how many fractional digits to compare. A boolean value of 1 means that *num* is a properly formed number and passes the range check, if one is specified.

$ISVALIDNUM validates American format numbers, which use a period (.) as the decimal separator. It does not validate European format numbers, which use a comma (,) as the decimal separator. $ISVALIDNUM does not consider valid a number that contains numeric group separators; it returns 0 (invalid) for any number containing a comma or a blank space, regardless of the current locale.

**Parameters**

**num**

The number to be validated may be an integer, a real number, or a scientific notation number (with the letter “E” or “e”). It may be a string, expression, or variable that resolves to a number. It may be signed or unsigned, and may contain leading or trailing zeros. Validation fails ($ISVALIDNUM returns 0) if:

- *num* is the empty string (“”).
- *num* contains any characters other than the digits 0–9, a leading + or – sign, a decimal point (.), and a letter “E” or “e”.
- *num* contains more than one + or – sign, decimal point, or letter “E” or “e”.
- The optional + or – sign is not the first character of *num*.
- The letter “E” or “e” indicating a base-10 exponent is not followed by an integer in a numeric string.

With the exception of $ISVALIDNUM, specifying a base-10 number with a non-integer exponent in any expression results in a <SYNTAX> error. For example, `WRITE 7E3.5`.

The *scale* parameter value causes evaluation using rounded or truncated versions of the *num* value. The actual value of the *num* variable is not changed by $ISVALIDNUM processing.

If *num* is the INF, –INF, or NAN value returned by $DOUBLE, $ISVALIDNUM returns 1.

The largest floating point number that Caché supports is 1.7976931348623157081E308. Specifying a larger number in any Caché numeric operation generates a <MAXNUMBER> error. The largest Caché decimal floating point number specified as a string that $ISVALIDNUM supports is 9.223372036854775807E145. For floating point number strings larger than this, use SIVALDDOUBLE. For further details, refer to “Extremely Large Numbers” in the “Data Types and Values” chapter of *Using Caché ObjectScript*. 
The `scale` parameter is used during range checking to specify how many fractional digits to compare. Specify an integer value for `scale`; fractional digits in the `scale` value are ignored. You can specify a `scale` value larger than the number of fractional digits specified in the other parameters. You can specify a `scale` value of −1; all other negative `scale` values result in a <FUNCTION> error.

A nonnegative `scale` value causes `num` to be rounded to that number of fractional digits before performing `min` and `max` range checking. A `scale` value of 0 causes `num` to be rounded to an integer value (3.9 = 4) before performing range checking. A `scale` value of −1 causes `num` to be truncated to an integer value (3.9 = 3) before performing range checking. To compare all specified digits without rounding or truncating, omit the `scale` parameter. A `scale` value that is nonnumeric or the null string is equivalent to a `scale` value of 0.

Rounding is performed for all `scale` values except −1. A value of 5 or greater is always rounded up.

If you omit the `scale` parameter, retain the comma as a place holder.

When rounding numbers, be aware that IEEE floating point numbers and standard Caché fractional numbers differ in precision. `$DOUBLE` IEEE floating point numbers are encoded using binary notation. They have a precision of 53 binary bits, which corresponds to 15.95 decimal digits of precision. (Note that the binary representation does not correspond exactly to a decimal fraction.) Because most decimal fractions cannot be exactly represented in this binary notation, an IEEE floating point number may differ slightly from the corresponding standard Caché floating point number. Standard Caché fractional numbers have a precision of 18 decimal digits on all supported Caché system platforms. When an IEEE floating point number is displayed as a fractional number, the binary bits are often converted to a fractional number with far more than 18 decimal digits. This does not mean that IEEE floating point numbers are more precise than standard Caché fractional numbers.

### min and max

You can specify a minimum allowed value, a maximum allowed value, neither, or both. If specified, the `num` value (after the `scale` operation) must be greater than or equal to the `min` value, and less than or equal to the `max` value. A null string as a `min` or `max` value is equal to zero. If a value does not meet these criteria, `$ISVALIDNUM` returns 0.

If you omit a parameter, retain the comma as a place holder. For example, when omitting `scale` and specifying `min` or `max`, or when omitting `min` and specifying `max`. Trailing commas are ignored.

If the `num`, `min`, or `max` value is a `$DOUBLE` number, then all three of these numbers are treated as a `$DOUBLE` number for this range check. This prevents unexpected range errors caused by the small generated fractional part of a `$DOUBLE` number.

### Examples

In the following example, each invocation of `$ISVALIDNUM` returns 1 (valid number):

```plaintext
WRITE 1,$ISVALIDNUM(0) ; All integers OK
WRITE 1,$ISVALIDNUM(4.567) ; Real numbers OK
WRITE 1,$ISVALIDNUM("4.567") ; Numeric strings OK
WRITE 1,$ISVALIDNUM(0.0) ; Signed numbers OK
WRITE 1,$ISVALIDNUM(+004.500) ; Leading/trailing zeroes OK
WRITE 1,$ISVALIDNUM(4E2) ; Scientific notation OK
```

In the following example, each invocation of `$ISVALIDNUM` returns 0 (invalid number):

```plaintext
WRITE 1,$ISVALIDNUM("") ; Null string is invalid
WRITE 1,$ISVALIDNUM("4.56") ; Commas are not permitted
WRITE 1,$ISVALIDNUM("AA") ; Invalid character
```

In the following example, each invocation of `$ISVALIDNUM` returns 1 (valid number), even though INF (infinity) and NAN (Not A Number) are, strictly speaking, not numbers:
DO %%class(%SYSTEM.Process).IEEEError(0)
WRITE ,,ISVALIDNUM($DOUBLE($ZPI)) ; DOUBLE numbers OK
WRITE ,,ISVALIDNUM($DOUBLE("INF")) ; DOUBLE INF OK
WRITE ,,ISVALIDNUM($DOUBLE("NAN")) ; DOUBLE NAN OK
WRITE ,,ISVALIDNUM($DOUBLE(1)/0) ; generated INF OK

The following example shows the use of the min and max parameters. All of the following return 1 (number is valid and also passes the range check):

WRITE ,,ISVALIDNUM(4,,3,5) ; scale can be omitted
WRITE ,,ISVALIDNUM(4,2,3,5) ; scale can be larger than number of fractional digits
WRITE ,,ISVALIDNUM(4,0,,5) ; min or max can be omitted
WRITE ,,ISVALIDNUM(4,0,4,4) ; min and max are inclusive
WRITE ,,ISVALIDNUM(-4,0,-5,5) ; negative numbers
WRITE ,,ISVALIDNUM(4.00,2,04,05) ; leading/trailing zeros
WRITE ,,ISVALIDNUM(.4E3,0,3E2,400) ; base-10 exponents expanded

The following example shows the use of the scale parameter with min and max. All of the following return 1 (number is valid and also passes the range check):

WRITE ,,ISVALIDNUM(4.55,,4.54,4.551)
; When scale is omitted, all digits of num are checked.
WRITE ,,ISVALIDNUM(4.1,0,4.0,4.01)
; When scale=0, num is rounded to an integer value
; (0 fractional digits) before min & max check.
WRITE ,,ISVALIDNUM(3.85,1,3.9,5)
; num is rounded to 1 fractional digit,
; (with values of 5 or greater rounded up)
; before min check.
WRITE ,,ISVALIDNUM(4.01,17,3,5)
; scale can be larger than number of fractional digits.
WRITE ,,ISVALIDNUM(3.9,-1,2,3)
; When scale=--1, num is truncated to an integer value

Notes

$ISVALIDNUM and $ISVALIDDDOUBLE Compared

The $ISVALIDNUM and $ISVALIDDDOUBLE functions both validate numbers and return a boolean value (0 or 1).

- Both functions accept as valid numbers the INF, –INF, and NAN values returned by $DOUBLE. $ISVALIDDDOUBLE also accepts as valid numbers the not case-sensitive strings “NAN” and “INF”, as well as the variants “Infinity” and “sNAN”, and any of these strings beginning with a single plus or minus sign. $ISVALIDNUM rejects all of these strings as invalid, and returns 0.

WRITE ,,ISVALIDNUM($DOUBLE("NAN")) ; returns 1
WRITE ,,ISVALIDDDOUBLE($DOUBLE("NAN")) ; returns 1
WRITE ,,ISVALIDNUM("NAN") ; returns 0
WRITE ,,ISVALIDDDOUBLE("NAN") ; returns 1

- Both functions parse signed and unsigned integers (including –0), scientific notation numbers (with “E” or “e”), real numbers (123.45) and numeric strings (“123.45”).

- Neither function recognizes the European DecimalSeparator character (comma (,)) or the NumericGroupSeparator character (American format: comma (,); European format: period (.)). For example, both reject the string “123,456" as an invalid number, regardless of the current locale setting.

- Both functions parse multiple leading signs (+ and –) for numbers. Neither accepts multiple leading signs in a quoted numeric string.

If a numeric string is too big to be represented by a Caché floating point number, the default is to automatically convert it to an IEEE double-precision number. However, such large numbers fail the $ISVALIDNUM test, as shown in the following example:
WRITE !,"E127 no IEEE conversion required"
WRITE !,$ISVALIDNUM("9223372036854775807E127")
WRITE !,$ISVALIDDOUBLE("9223372036854775807E127")
WRITE !,"E128 automatic IEEE conversion"
WRITE !,$ISVALIDNUM("9223372036854775807E128")
WRITE !,$ISVALIDDOUBLE("9223372036854775807E128")

$ISVALIDNUM, $NORMALIZE, and $NUMBER Compared

The $ISVALIDNUM, $NORMALIZE, and $NUMBER functions all validate numbers. $ISVALIDNUM returns a boolean value (0 or 1). $NORMALIZE and $NUMBER return a validated version of the specified number.

These three functions offer different validation criteria. Select the one that best meets your needs.

- American format numbers are validated by all three functions. European format numbers are only validated by the $NUMBER function.
- All three functions parse signed and unsigned integers (including –0), scientific notation numbers (with “E” or “e”), and numbers with a fractional part. However, $NUMBER can be set (using the “I” format) to reject numbers with a fractional part (including scientific notation with a negative base-10 exponent). All three functions parse both numbers (123.45) and numeric strings (“123.45”).
- Leading and trailing zeroes are stripped out by all three functions. The decimal character is stripped out unless followed by a nonzero value.
- DecimalSeparator: $NUMBER validates the decimal character (American format: period (.) or European format: comma (,)) based on its format parameter (or the default for the current locale). The other functions only validate American format decimal numbers, regardless of the current locale setting.
- NumericGroupSeparator: $NUMBER accepts NumericGroupSeparator characters (in American format: comma (,) or blank space; in European format: period (.) or blank space). It accepts and strips out any number of NumericGroupSeparator characters, regardless of position. For example, in American format it validate “12 3,4,56,9,9” as the number 123456.99. $NORMALIZE does not recognize NumericGroupSeparator characters. It validates character-by-character until it encounters a nonnumeric character; for example, it validates “123,456.99” as the number 123. $ISVALIDNUM rejects the string “123,456” as an invalid number.
- Multiple leading signs (+ and –) are interpreted by all three functions for numbers. However, only $NORMALIZE accepts multiple leading signs in a quoted numeric string.
- Trailing + and – signs: All of the three functions reject trailing signs in numbers. However, in a quoted numeric string $NUMBER parses one (and only one) trailing sign, $NORMALIZE parses multiple trailing signs, and $ISVALIDNUM rejects any string containing a trailing sign as an invalid number.
- Parentheses: $NUMBER parses parentheses surrounding an unsigned number in a quoted string as indicating a negative number. $NORMALIZE and $ISVALIDNUM reject parentheses.
- Numeric strings containing multiple decimal characters: $NORMALIZE validates character-by-character until it encounters the second decimal character. For example, in American format it validates “123.4.56” as the number 123.4. $NUMBER and $ISVALIDNUM reject any string containing more than one decimal character as an invalid number. Numeric strings containing other nonnumeric characters: $NORMALIZE validates character-by-character until it encounters an alphabetic character. It validates “123A456” as the number 123. $NUMBER and $ISVALIDNUM validate the entire string, they reject “123A456” as an invalid number.
- The null string: $NORMALIZE parses the null string as zero (0). $NUMBER and $ISVALIDNUM reject the null string.

The $ISVALIDNUM and $NUMBER functions provide optional min/max range checking.

$ISVALIDNUM, $NORMALIZE, and $NUMBER all provide rounding of numbers to a specified number of fractional digits. $ISVALIDNUM and $NORMALIZE can round fractional digits, and round or truncate a number with a fractional
part to return an integer. For example, $NORMALIZE can round 488.65 to 488.7 or 489, or truncate it to 488. $NUMBER can round both fractional digits and integer digits. For example, $NUMBER can round 488.65 to 488.7, 489, 490 or 500.

See Also

- $DOUBLE function
- $FNUMBER function
- $INUMBER function
- $ISVALIDDOUBLE function
- $NORMALIZE function
- $NUMBER function
$JUSTIFY

Right-aligns an expression within a specified width, rounding to a specified number of fractional digits.

$JUSTIFY(expression,width,decimal)
$J(expression,width,decimal)

Parameters

| expression | The value that is to be right-aligned. It can be a numeric value, a string literal, the name of a variable, or any valid ObjectScript expression. |
| width      | The number of characters within which expression is to be right-aligned. A positive integer or an expression that evaluates to a positive integer. |
| decimal    | Optional — The number of fractional digits. A positive integer or an expression that evaluates to a positive integer. Caché rounds or pads the number of fractional digits in expression to this value. If you specify decimal, Caché treats expression as a numeric. |

Description

$JUSTIFY returns the value specified by expression right-aligned within the specified width. You can include the decimal parameter to decimal-align numbers within width.

- $JUSTIFY(expression,width): the 2-parameter syntax right-justifies expression within width. It does not perform any conversion of expression. The expression can be a numeric or a nonnumeric string.
- $JUSTIFY(expression,width,decimal): the 3-parameter syntax converts expression to a canonical number, rounds or zero pads fractional digits to decimal, then right-justifies the resulting numeric value within width. If expression is a nonnumeric string, Caché converts it to 0, pads it, then right-justifies it.

$JUSTIFY recognizes the DecimalSeparator character for the current locale. It adds or deletes a DecimalSeparator character as needed. The DecimalSeparator character depends upon the locale; commonly it is either a period (.) for American-format locales, or a comma (,) for European-format locales. To determine the DecimalSeparator character for your locale, invoke the following method:

WRITE ##class(%SYS.NLS.Format).GetFormatItem("DecimalSeparator")

Commonly, $JUSTIFY is used to format numbers with fractional digits: every number is given the same number of fractional digits, and the numbers are right-aligned so that the DecimalSeparator characters align in a column of numbers. $JUSTIFY is especially useful for outputting formatted values using the WRITE command.

Parameters

expression

The value to be right-justified, and optionally expressed as a numeric with a specified number of fractional digits.

- If string justification is desired, do not specify decimal. The expression can contain any characters. $JUSTIFY right-justifies expression, as described in width.
- If numeric justification is desired, specify decimal. If decimal is specified, $JUSTIFY converts expression to a canonical number. It resolves leading plus and minus signs and removes leading and trailing zeros. It truncates expression at the first nonnumeric character. If expression begins with a nonnumeric character (such as a currency symbol), $JUSTIFY converts the expression value to 0. For further details on how Caché converts a numeric to a canonical
number, and Caché handling of a numeric string containing nonnumeric characters, refer to the Numbers section of the “Data Types and Values” chapter of Using Caché ObjectScript.

After $JUSTIFY converts expression to a canonical number, it zero-pads or rounds this canonical number to decimal number of fractional digits, then right-justifies the result, as described in width. $JUSTIFY does not recognize NumericGroupSeparator characters, currency symbols, multiple DecimalSeparator characters, or trailing plus or minus signs.

**width**

The width in which to right-justify the converted expression. If width is greater than the length of expression (after numeric and fractional digit conversion), Caché right-justifies to width, left-padding as needed with blank spaces. If width is less than the length of expression (after numeric and fractional digit conversion), Caché sets width to the length of the expression value.

Specify width as a positive integer. A width value of 0, the null string (""), or a nonnumeric string is treated as a width of 0, which means that Caché sets width to the length of the expression value.

**decimal**

The number of fractional digits. If expression contains more fractional digits, $JUSTIFY rounds the fractional portion to this number of fractional digits. If expression contains fewer fractional digits, $JUSTIFY pads the fractional portion with zeros to this number of fractional digits, adding a DecimalSeparator character, if needed. If decimal=0, $JUSTIFY rounds expression to an integer value and deletes the Decimal Separator character.

If the expression value is less than 1, $JUSTIFY inserts a leading zero before the DecimalSeparator character.

The $DOUBLE values INF, -INF, and NAN are returned unchanged by $JUSTIFY, regardless of the decimal value.

### $JUSTIFY and $FNUMBER

You can use $FNUMBER to format a number for display. Both $JUSTIFY and $FNUMBER can round (or zero pad) to a specified number of fractional digits. $FNUMBER can also be used to add NumericGroupSeparator characters. However, note the following:

- $FNUMBER cannot format a number once it has been right-aligned using $JUSTIFY. ($FNUMBER interprets the leading spaces as nonnumeric characters.)
- $JUSTIFY cannot perform numeric justification on a number once you have added NumericGroupSeparator characters or have prepended a currency symbol. ($JUSTIFY interprets NumericGroupSeparators or currency symbols as non-numeric characters.)

Therefore, to properly add NumericGroupSeparators, round fractional digits, prepend a currency symbol, and right-align the resulting number, you use $FNUMBER to perform rounding and inserting of NumericGroupSeparators. You then use $JUSTIFY with 2-parameter syntax to right-align the resulting string:

```objectscript
SET num=123456.789
SET fmtnum=$FNUMBER(num","",2)
SET money="$"_fmtnum
SET rmoney=$JUSTIFY(money,15)
WRITE ">",rmoney,"<"
```

### Examples

The following example performs right-justification on strings. No numeric conversion is performed:

```objectscript
WRITE ">",$JUSTIFY("right",10),"<",!
WRITE ">",$JUSTIFY("aligned",10),"<",!
WRITE ">",$JUSTIFY("+0123.456",10),"<",!
WRITE ">",$JUSTIFY("string longer than width",10),"<",!
```

The following example performs numeric right-justification with a specified number of fractional digits:
return the following lines:

250.51
875.00

TOTAL 1125.51

The following example performs numeric right-justification with the $DOUBLE values INF and NAN:

SET rtn=##class(%SYSTEM.Process).IEEEError(0)
SET x=$DOUBLE(1.2e500)
WRITE !,"Double: ",x
WRITE !,">",$JUSTIFY(x,12,2),"<"
SET y=$DOUBLE(x-x)
WRITE !,"Double INF minus INF: ",y
WRITE !,">",$JUSTIFY(y,12,2),"<"

See Also

• $FNUMBER function
• $X special variable
$LENGTH

Returns the number of characters or delimited substrings in a string.

$LENGTH(expression, delimiter)
$L(expression, delimiter)

Parameters

<table>
<thead>
<tr>
<th>expression</th>
<th>The target string. It can be a numeric value, a string literal, a variable name, or any valid expression that resolves to a string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>delimiter</td>
<td>Optional — A string that demarcates separate substrings in the target string. It can be a variable name, a numeric value, a string literal, or any valid expression that resolves to a string.</td>
</tr>
</tbody>
</table>

Description

$LENGTH returns the number of characters in a specified string or the number of delimited substrings in a specified string, depending on the parameters used. Note that length counts the number of characters; an 8-bit character and a 16-bit wide (Unicode) character are both counted as one character. For further details on Caché Unicode support, refer to Unicode in Using Caché ObjectScript.

- $LENGTH(expression) returns the number of characters in the string. If the expression is a null string, $LENGTH returns a 0. If expression is a numeric expression, it is converted to canonical form before determining its length. If expression is a string numeric expression, no conversion is performed. If expression is the $DOUBLE values INF, -INF, or NAN, the lengths returned are 3, 4, and 3, respectively.

  This syntax can be used with the $EXTRACT function, which locates a substring by position and returns the substring value.

- $LENGTH(expression, delimiter) returns the number of substrings within the string. $LENGTH returns the number of substrings separated from one another by the indicated delimiter. This number is always equal to the number of delimiters in the string, plus one.

  This syntax can be used with the $PIECE function, which locates a substring by a delimiter and returns the substring value.

  If the delimiter is the null string, $LENGTH returns a 0. If the delimiter is any other valid string literal and the string is a null string, $LENGTH returns a 1.

Encoded Strings

Caché supports strings that contain internal encoding. Because of this encoding, $LENGTH should not be used to determine the data content of a string.

- $LENGTH should not be used for a List structure string created using $LISTBUILD or $LIST. Because a Caché List string is encoded, the length returned does not meaningfully indicate the number of characters in the list elements. The sole exception is the one-argument and two-argument forms of $LIST, which take an encoded Caché List string as input, but outputs a single List element value as a standard character string. You can use the $LISTLENGTH function to determine the number of substrings (list elements) in an encoded list string.

- $LENGTH should not be used for a bit string. Because a Caché bit string is encoded, the length returned does not meaningfully indicate the number of bits in the bit string. You can use the $BITCOUNT function, which returns the number of bits in the string.
$LENGTH should not be used for a JSON string. The value assigned by setting a variable to a JSON object or a JSON array is an object reference. Therefore the length of that variable value would be the length of the object reference, which has no connection to the length of the data encoded in the JSON string.

**Surrogate Pairs**

$LENGTH does not recognize surrogate pairs. Surrogate pairs are used to represent some Chinese characters and to support the Japanese JIS2004 standard. You can use the $WISWIDE function to determine if a string contains a surrogate pair. The $WLENGTH function recognizes and correctly parses surrogate pairs. $LENGTH and $WLENGTH are otherwise identical. However, because $LENGTH is generally faster than $WLENGTH, $LENGTH is preferable for all cases where a surrogate pair is not likely to be encountered.

**Examples**

In the following example, both $LENGTH functions return 4, the number of characters in the string.

```ObjectScript
IF $SYSTEM.Version.IsUnicode() {
    SET roman="test"
    WRITE 1,$LENGTH(roman)," characters in: ",roman
    SET greek=$CHAR(964,949,963,964)
    WRITE 1,$LENGTH(greek)," characters in: ",greek
} 
ELSE (WRITE "This example requires a Unicode installation of Caché")
```

In the following example, the first $LENGTH returns 5. This is the length of 74000, the canonical version of the specified number. The second $LENGTH returns 8, the length of the string “+007.4e4”.

```ObjectScript
WRITE 1,$LENGTH(+007.4e4)
WRITE 1,$LENGTH("+007.4e4")
```

In the following example, the first WRITE returns 11 the number of characters in var1 (including, of course, the space character). The second WRITE returns 2, the number of substrings in var1 using the space character as the substring delimiter.

```ObjectScript
SET var1="HELLO WORLD"
WRITE 1,$LENGTH(var1)
WRITE 1,$LENGTH(var1, " ")
```

The following example returns 3, the number of substrings within the string, as delimited by the dollar sign ($) character.

```ObjectScript
SET STR="ABC$DEF$EFG",DELIM="$"
WRITE $LENGTH(STR,DELIM)
```

If the specified delimiter is not found in the string $LENGTH returns 1, because the only substring is the string itself.

The following example returns a 0 because the string tested is the null string.

```ObjectScript
SET Nstring = ""
WRITE $LENGTH(Nstring)
```

The following example shows the values returned when a delimiter or its string is the null string.

```ObjectScript
SET String = "ABC"
SET Nstring = ""
SET Delim = ""
SET Ndelim = ""
WRITE 1,$LENGTH(String,Delim) ; returns 1
WRITE 1,$LENGTH(Nstring,Delim) ; returns 1
WRITE 1,$LENGTH(String,Ndelim) ; returns 0
WRITE 1,$LENGTH(Nstring,Ndelim) ; returns 0
```

**See Also**

- $SEXTRACT function
- $SPICE function
- $WISWIDE function
- $WLENGTH function
$LIST

Returns or replaces elements in a list.

$LIST(list,position,end)
$LI(list,position,end)

SET $LIST(list,position,end)=value
SET $LI(list,position,end)=value

Parameters

| list | An expression that evaluates to a valid list. Because lists contain encoding, list must be created using $LISTBUILD or $LISTFROMSTRING, or extracted from another list using $LIST. In SET $LIST syntax, list must be a variable or a multi-dimensional property. |
| position | Optional — An integer code specifying the starting position in list from which to retrieve a sublist. Permitted values are \(n\) (integer count from beginning of list), * (last element in list), and *\(-n\) (relative offset count backwards from end of list). SET $LIST syntax also supports *+n (relative offset integer count of elements to append beyond the end of list). Thus, the first element in the list is 1, the second element is 2, the last element in the list is *, and the next-to-last element is *-1. If position is omitted, it defaults to 1. 
-1 may be used in older code to specify the last element in the list. This deprecated use of -1 should not be combined with *, *-n, or *+n relative offset syntax. |
| end | Optional — An integer code specifying the ending position of a sublist of list. Can be a positive integer specifying position count from the beginning of list or a symbolic code counting from the end of list. Used with position and uses the same code values as position. If omitted, only the single element specified by position is returned. |

Description

$LIST can be used in two ways:

- To return an element (or elements) from list. This uses the $LIST(list,position,end) syntax. It locates elements by offset from either the beginning or the end of list. It returns a single element as a string. It return a range of elements as a list.

- To replace an element (or elements) within list. The replacement element may be the same length, longer, or shorter than the original element. This uses the SET $LIST(list,position,end)=value syntax.

Returning List Elements

$LIST returns a single element or a range of elements from a list. The elements returned depend on the parameters used.

- $LIST(list) returns the first element in list as a string.
- $LIST(list,position) returns the element of list specified by position as a string. The specified position cannot be a positive integer count beyond the end of list or a negative integer count before the beginning of list.
- $LIST(list,position,end) returns a “sublist” (an encoded list string) containing a range of elements retrieved from list. The range is from the specified start position through the specified end position (inclusive). If position and end specify the same element, $LIST returns this element as an encoded list.
The specified position cannot be a positive integer count beyond the end of list or a negative integer count before the beginning of list. The specified end can be a positive integer count beyond the end of list but only the existing list elements are returned; no element padding is performed.

Note: $LIST should not be used in a loop structure to return multiple successive element values. While this will work, it is highly inefficient, because $LIST must evaluate the list from the beginning with each iteration. The $LIST-NEXT function is a far more efficient way to return multiple successive element values.

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 empty string (“”) and certain $CHAR non-printing character combinations, such as $CHAR(1), $CHAR(2,1), and $CHAR(3,1,asciicode) can also return an encoded empty or one-element list.)

When returning an element (or elements), list can be a variable or an object property.

When $LIST is used with SET on the left hand side of the equals sign to replace an element (or elements), list can be a variable or a multidimensional property reference; it cannot be a non-multidimensional object property.

The following are valid list arguments:

```
SET myList = $LISTBUILD("Red","Blue","Green","Yellow")
WRITE 1,$LIST(myList,2) ; prints Blue
SET subList = $LIST(myList,2,4)
WRITE 1,$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:

```
SET myList = $LISTBUILD("Red","Blue","Green","Yellow")
SET subList = $LIST(myList,2)
WRITE $LIST(subList,1)
```

In SET $LIST syntax form, list cannot be a non-multidimensional object property.

position

The position (element count) of the list element to return (or replace). A single element is returned as a string. List elements are counted from 1. If position is omitted, $LIST returns the first element.

• If position is a positive integer, $LIST counts elements from the beginning of list. If position is greater than the number of elements in list, Caché issues a <NULL VALUE> error.

• If position is * (asterisk), $LIST returns the last element in list.

• If position is *-n (an asterisk followed by a negative number), $LIST counts elements by offset backwards from the end of list. Thus, *-0 is the last element in the list, *-1 is the next-to-last list element (an offset of 1 from the end). If the position relative offset count is equal to the number of elements in list (and thus specifies the 0th element), Caché issues a <NULL VALUE> error. If the position relative offset count is greater than the number of elements in list, Caché issues a <RANGE> error.

• For SET $LIST syntax only — If position is +n (an asterisk followed by a positive number), SET $LIST appends elements by offset beyond the end of list. Thus, *+1 appends an element beyond the end of list. *+2 appends an element two positions beyond the end of list, padding with a null string element.

• If position is 0 or -0, Caché issues a <NULL VALUE> error.
If the *end* parameter is specified, *position* specifies the first element in a range of elements. A range of elements is always returned as an encoded list string. 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 the same element, but not the same data value as $LIST(x,2,2).

**end**

The position of the last element in a range of elements, specified as an integer. You must specify *position* to specify *end*. If *end* is a fractional number, it is truncated to its integer part.

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.

- **position < end**: If *end* and *position* are positive integers, and *position < end*, $LIST returns an encoded sublist containing the specified list of elements, inclusive of the *position* and *end* elements. If *position* is 0 or 1, the sublist begins with the first element in list. If *end* is greater than the number of elements in list, $LIST returns an encoded sublist containing all of the elements from *position* through the end of the list. If *end* is *-n*, *position* can be a positive integer or a *-n* value greater than or equal to this *end* position. Thus, $LIST(fourlist,*-1,*), $LIST(fourlist,*-3,*-2), $LIST(fourlist,2,*-1) are all valid sublists.

- **position = end**: If *end* and *position* evaluate to the same element, $LIST returns an encoded sublist containing that single element. For example, in a list with four elements, $LIST(fourlist,3,2), $LIST(fourlist,7,*), or $LIST(fourlist,*,*-2,*-2) all return the null string.

- **position > end**: If *position > end*, $LIST returns the null string (""). For example, in a list with four elements, $LIST(fourlist,3,2), $LIST(fourlist,7,*), or $LIST(fourlist,*,*-2,*-2) all return the null string.

- Position *0*, with *end*: If *end* is specified, and *position* is zero (0) or a negative offset that evaluates to position zero, $LIST $LIST returns an encoded sublist containing the elements from *position 1* through the *end* position. If *end* also evaluates to element position zero, $LIST returns the null string (""), because *position > end*.

- For SET $LIST syntax only — If *end* is *++n* (an asterisk followed by a positive number), SET $LIST appends a range of elements by offset beyond the end of list. If *position* is *++n*, SET $LIST appends a range of values. If *position* is a positive integer, or *-n* SET $LIST both replaces and appends values. To replace the last element and append elements, specify SET $LIST(mylist,*+0,*+n). If the start of the specified range is beyond the end of list, the list is padded with a null string elements as needed. If *end* is larger than the supplied range of values, trailing padding is not performed.

**Deprecated –1 Values**

In older code, a *position* or *end* value of -1 represents the last element in the list. A value of -1 cannot be used with *, *++n, or *-n* syntax.

**Specifying *-n and *+n Parameter Values**

When using a variable to specify *-n or *+n*, you must always specify the asterisk and a sign character in the parameter itself.

The following are valid specifications of *-n:  

```
SET count=2
SET alph=$LISTBUILD("a","b","c","d")
WRITE $LIST(alph,*-count)
```

```
SET count=-2
SET alph=$LISTBUILD("a","b","c","d")
WRITE $LIST(alph,*+count)
```

The following is a valid specification of *++n:  

```
SET count=2
SET alph=$LISTBUILD("a","b","c","d")
WRITE $LIST(alph,*+0,*+n)
```
SET count=2
SET alph=$LISTBUILD("a","b","c","d")
SET $LIST(alph,"+count")="F"
WRITE $LISTTOSTRING(alph,"^",1)

Whitespace is permitted within these parameter values.

$LIST Errors

The following $LIST parameter values generate an error:

- If the list parameter does not evaluate to a valid list, $LIST generates a <LIST> error. You can use the $LISTVALID function to determine if a list is valid.
- If the list parameter evaluates to a valid list that contains a null value, or concatenates a list and a null value, $LIST(list) syntax generates a <NULL VALUE> error because this syntax is trying to return a null value as a string. All of the following are valid lists (according to $LISTVALID) for which $LIST generate a <NULL VALUE> error:

  WRITE $LIST(""),!
  WRITE $LIST($LB()),!
  WRITE $LIST($LB(UndefinedVar)),!
  WRITE $LIST($LB()),
  WRITE $LIST($LB())_SLB("a","b","c")

If the $LIST(list,position) syntax position parameter specifies a null (non-existent) element, $LIST generates a <NULL VALUE> error because this syntax is trying to return a null value as a string:

  SET mylist=$LISTBUILD("A","C")
  ZZDUMP $LIST(mylist,2) ; generates a <NULL VALUE> error
  SET mylist2=$LISTBUILD("A","B","C")
  WRITE $LIST(mylist2,4) ; generates a <NULL VALUE> error

- If the $LIST(list,position) syntax specifies a position parameter of 0, or a negative relative offset that specifies the 0th element, $LIST generates a <NULL VALUE> error. If end is specified, 0 is a valid position value and is parsed as 1.

- If the -*n value of the position or end parameter specifies an n value larger than the number of element positions in list, $LIST generates a <RANGE> error.

  SET list2=$LISTBUILD("Brown","Black")
  WRITE $LIST(list2,"*-2") ; generates a <NULL VALUE> error
  WRITE $LIST(list2,"*-3") ; generates a <RANGE> error

  Because $LISTLENGTH("") is 0, a position or end of -*1 or greater results in a <RANGE> error:

  WRITE $LIST("",*-0) ; generates a <NULL VALUE> error
  WRITE $LIST("",-1) ; generates a <RANGE> error
  WRITE $LIST("",0,-1) ; generates a <RANGE> error

- If the value of the position parameter or the end parameter is less than -1, $LIST generates a <RANGE> error.

Replacing Elements Using SET $LIST

- You can use SET $LIST(list,position) to remove an element, replace an element’s value, or append an element to the list. In this two-parameter form, you specify the new element value as a string.

- You can use SET $LIST(list,position,end) to remove one or more elements, replace one or more element values, or append one or more elements to the list. In this three-parameter form, you must specify the new element value(s) as an encoded list.

When $LIST is used with SET on the left hand side of the equals sign, list can be a valid variable name. If the variable does not exist, SET $LIST defines it. The list parameter can also be a multidimensional property reference; it cannot be a non-multidimensional object property. Attempting to use SET $LIST on a non-multidimensional object property results in an <OBJECT DISPATCH> error.
You cannot use `SET (a,b,c,...)=value` syntax with `$LIST` (or `$PIECE` or `$EXTRACT`) on the left of the equals sign, if the function uses relative offset syntax: * representing the end of a string and *-n or *+n representing relative offset from the end of the string. You must instead use `SET a=value,b=value,c=value,...` syntax.

You can also use `$LISTUPDATE` to replace one or more elements in a list or append element to a list by element position. `$LISTUPDATE` replaces list elements, performing a boolean test for each element replacement. Unlike `SET $LIST`, `$LISTUPDATE` does not modify the initial list, but returns a copy of that list with the specified element replacements.

**Two Parameter Operations**

You can perform the following two-parameter operations. Note that two-parameter operations specify an element value as a string. Specifying an element value as a list creates a sublist within the list.

- Replace one element value with a new value:
  
  ```
  SET $LIST(fruit,2)="orange" ; count from beginning of list
  SET $LIST(fruit,*)="pear" ; element at end of list
  SET $LIST(fruit,*-2)="peach" ; offset from end of list
  ```

- Remove an element value (this sets the value to the null string; it does not remove the element position):
  
  ```
  SET $LIST(fruit,2)="
  ```

- Append an element to a list. You can append to the end of the list, or to a location past the end of the list, by using **+n syntax. **SET $LIST** inserts null value elements as needed to pad to the specified position:
  
  ```
  SET $LIST(fruit,*+1)="plum"
  ```

- Replace one element with a sublist of elements:
  
  ```
  SET $LIST(fruit,3)=$LISTBUILD("orange","banana")
  ```

**Three Parameter Operations**

You can perform the following three-parameter (range) operations. Note that range operations specify an element values as a list, even when specifying a single element value.

- Replace one element with several elements:
  
  ```
  SET $LIST(fruit,3,3)=$LISTBUILD("orange","banana")
  ```

- Replace a range of element values with the same number of new values:
  
  ```
  SET $LIST(fruit,2,3)=$LISTBUILD("orange","banana")
  ```

- Replace a range of element values with a larger or smaller number of new values:
  
  ```
  SET $LIST(fruit,2,3)=$LISTBUILD("orange","banana","peach")
  ```

- Remove a range of element values (this sets the element values to the null string; it does not remove the element positions):
  
  ```
  SET $LIST(fruit,2,3)=$LISTBUILD("","")
  ```

- Remove a range of element values and their positions:
  
  ```
  SET $LIST(fruit,2,3)="
  ```

- Append a range of element to a list. You can append to the end of the list, or to a location past the end of the list, by using **+n syntax. **SET $LIST** inserts null string value elements as needed to pad to the specified position:
  
  ```
  SET $LIST(fruit,**+1,**+2)=$LISTBUILD("plum","pear")
  ```
$LIST$ only appends the specified element values. If the end position is larger than the specified elements, empty trailing element positions are not created.

Examples

Examples of Returning Elements with $LIST$

The following examples use the 2-parameter form of $LIST$ to return a list element as a string:

The following two $LIST$ statements return “Red”, the first element in the list. The first returns the first element by default, the second returns the first element because the position parameter is set to 1. The value is returned as a string:

```
SET colorlist=$LISTBUILD("Red","Orange","Yellow","Green","Blue","Violet")
WRITE $LIST(colorlist),!
WRITE $LIST(colorlist,1)
```

The following two $LIST$ statements return “Orange”, the second element in the list. The first counts from the beginning of the list, the second counts backwards from the end of the list. The value is returned as a string:

```
SET colorlist=$LISTBUILD("Red","Orange","Yellow","Green","Blue","Violet")
WRITE $LIST(colorlist,2),!
WRITE $LIST(colorlist,*-4)
```

The following examples use the 3-parameter form of $LIST$ to return one or more elements as an encoded list string. Because a list contains non-printing encoding character, you must use $LISTTOSTRING$ to convert the sublist to a printable string.

The following two $LIST$ statements return “Blue”, the fifth element in the list as an encoded list string. The first counts from the beginning of the list, the second counts backwards from the end of the list. Because the element is specified as a range, it is retrieved as a list consisting of one element:

```
SET colorlist=$LISTBUILD("Red","Orange","Yellow","Green","Blue","Violet")
WRITE $LISTTOSTRING($LIST(colorlist,5,5))
WRITE $LISTTOSTRING($LIST(colorlist,*-1,*-1))
```

The following example returns “Red Orange Yellow”, a three-element list string beginning with the first element and ending with the third element in the list:

```
SET colorlist=$LISTBUILD("Red","Orange","Yellow","Green","Blue","Violet")
WRITE $LISTTOSTRING($LIST(colorlist,1,3))
```

The following example returns “Green Blue Violet”, a three-element list string beginning with the fourth element and ending with the last element in the list:

```
SET colorlist=$LISTBUILD("Red","Orange","Yellow","Green","Blue","Violet")
WRITE $LISTTOSTRING($LIST(colorlist,4,*))
```

The following example returns a list element from a property:

```
SET cfg=##class(%iKnow.Configuration).%New("Trilingual",1,$LB("en","fr","es"))
WRITE $LIST(cfg.Languages,2)
```

Examples of Replacing, Removing, or Appending Elements with SET $LIST$

The following example shows SET $LIST$ replacing the second element:

```
SET fruit=$LISTBUILD("apple","onion","banana","pear")
WRITE !,$LISTTOSTRING(fruit,"/")
SET $LIST(fruit,2)="orange"
WRITE !,$LISTTOSTRING(fruit,"/")
```

The following example shows SET $LIST$ replacing the second and third elements:
The following example shows SET $LIST replacing the second and third elements with four elements:

```
SET fruit=$LISTBUILD("apple","potato","onion","pear")
WRITE !,$LISTTOSTRING(fruit,"/"
SET $LIST(fruit,2,3)=$LISTBUILD("orange","banana")
WRITE !,$LISTTOSTRING(fruit,"/"
```

The following example shows SET $LIST appending an element to the end of the list:

```
SET fruit=$LISTBUILD("apple","orange","banana","peach")
WRITE $LL(fruit)," ",$LISTTOSTRING(fruit,"/",1),!
SET $LIST(fruit,*+1)="pear"
WRITE $LL(fruit)," ",$LISTTOSTRING(fruit,"/",1)
```

The following example shows SET $LIST appending an element three positions past the end of the list:

```
SET fruit=$LISTBUILD("apple","orange","banana","peach")
WRITE $LL(fruit)," ",$LISTTOSTRING(fruit,"/",1),!
SET $LIST(fruit,*+3)="tangerine"
WRITE $LL(fruit)," ",$LISTTOSTRING(fruit,"/",1)
```

The following four examples show SET $LIST using *-n syntax to replace elements by offset from the end of the list. Note that SET $LIST(x,*-n) and SET $LIST(x,n,*-n) perform different operations: SET $LIST(x,*-n) replaces the value of the specified element; SET $LIST(x,n,*-n) deletes the specified range of elements, then appends the specified list.

To replace the next-to-last element with a single value, use SET $LIST(x,*-1):

```
SET fruit=$LISTBUILD("apple","banana","orange","potato","pear")
WRITE !,"list length is ",$LISTLENGTH(fruit)," 
WRITE $LL(fruit)," ",$LISTTOSTRING(fruit,"/"
SET $LIST(fruit,*-1)="peach"
WRITE !,"list length is ",$LISTLENGTH(fruit)," 
WRITE $LISTTOSTRING(fruit,"/"
```

To remove a single element by offset from the end of the list, use SET $LIST(x,*-n,*-n)="":

```
SET fruit=$LISTBUILD("apple","banana","orange","potato","peach")
WRITE !,"list length is ",$LISTLENGTH(fruit)," 
WRITE $LL(fruit)," ",$LISTTOSTRING(fruit,"/"
SET $LIST(fruit,*-1,-1)=""
WRITE !,"list length is ",$LISTLENGTH(fruit)," 
WRITE $LISTTOSTRING(fruit,"/"
```

To replace a single element by offset from the end of the list with a list of elements, use SET $LIST(x,*-n,*-n)=list:

```
SET fruit=$LISTBUILD("apple","banana","potato","orange","pear")
WRITE !,"list length is ",$LISTLENGTH(fruit)," 
WRITE $LL(fruit)," ",$LISTTOSTRING(fruit,"/"
SET $LIST(fruit,*-2,*-2)=$LISTBUILD("peach","plum","quince")
WRITE !,"list length is ",$LISTLENGTH(fruit)," 
WRITE $LISTTOSTRING(fruit,"/"
```

To replace a single element by offset from the end of the list with a sublist, use SET $LIST(x,*-n)=list:

```
SET fruit=$LISTBUILD("apple","banana","potato","orange","pear")
WRITE !,"list length is ",$LISTLENGTH(fruit)," 
WRITE $LL(fruit)," ",$LISTTOSTRING(fruit,"/"
SET $LIST(fruit,*-2)=$LISTBUILD("peach","plum","quince")
WRITE !,"list length is ",$LISTLENGTH(fruit)," 
WRITE $LISTTOSTRING(fruit,"/"
```

The following example shows SET $LIST removing elements from the list, beginning with the third element through the end of the list:
SET fruit=$LISTBUILD("apple","orange","onion","peanut","potato")
WRITE !,"list length is ",$LISTLENGTH(fruit)," 
WRITE $LISTTOSTRING(fruit,"/")
SET $LIST(fruit,3,*)=""
WRITE !,"list length is ",$LISTLENGTH(fruit)," 
WRITE $LISTTOSTRING(fruit,"/")

Notes

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.

IF $SYSTEM.Version.IsUnicode()  {
  SET y=$LISTBUILD("ABC"_$CHAR(68),"XYZ")
  SET z=$LISTBUILD("ABC"_$CHAR(960),"XYZ")
  WRITE !,"The ASCII list y elements: 
  ZZDUMP $LIST(y,1)
  ZZDUMP $LIST(y,2)
  WRITE !,"The Unicode list z elements: 
  ZZDUMP $LIST(z,1)
  ZZDUMP $LIST(z,2)
}  ELSE {WRITE "This example requires a Unicode installation of Caché"

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.

$LIST Compared with $EXTRACT and $PIECE

$LIST determines an element from an encoded list by counting elements (not characters) from the beginning (or end) of the list.

$EXTRACT determines a substring by counting characters from the beginning (or end) of a string. $EXTRACT takes as input an ordinary character string.

$PIECE determines a substring by counting user-defined delimiter characters within the string. $PIECE takes as input an ordinary character string containing multiple instances of a character (or string) intended for use as a delimiter.

$LIST cannot be used on ordinary strings. $PIECE and $EXTRACT cannot be used on encoded lists.

See Also

- $LISTBUILD function
- $LISTDATA function
- $LISTFIND function
- $LISTFROMSTRING function
- $LISTGET function
- $LISTLENGTH function
- $LISTNEXT
- $LISTSAME function
- $LISTTOSTRING function
- $LISTUPDATE function
• $LISTVALID function
$LISTBUILD

Builds a list of elements from the specified expressions.

```objectscript
$LISTBUILD(element,...)
$LB(element,...)
SET $LISTBUILD(var1,var2,...)=list
SET $LB(var1,var2,...)=list
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>element</code></td>
<td>An expression that specifies a list element value. Can be a single expression or an expression in a comma-separated list of expressions. A placeholder comma can be specified for an omitted element.</td>
</tr>
<tr>
<td><code>var</code></td>
<td>A variable, specified as a single variable or as a variable in a comma-separated list of variables. A placeholder comma can be specified for an omitted variable. A <code>var</code> may be a variable of any type: local, process-private, or global, unsubscripted or subscripted.</td>
</tr>
<tr>
<td><code>list</code></td>
<td>An expression that evaluates to a valid list. Because lists contain encoding, <code>list</code> must be created using <code>$LISTBUILD</code> or <code>$LISTFROMSTRING</code>, or extracted from another list using <code>$LIST</code>.</td>
</tr>
</tbody>
</table>

### Description

$LISTBUILD has two syntax forms: `$LISTBUILD` and `SET $LISTBUILD`:

- `$LISTBUILD(element1,element2,...)` takes one or more expressions and returns an encoded list structure with one element for each expression. Elements are placed in the list in the order specified. Elements are counted from 1.
- `SET $LISTBUILD(var1,var2,...)=list` extracts multiple elements from a list into variables. Each element of `list` is copied into the `var` variable that corresponds to its position in the `$LISTBUILD` parameter list.

**$LISTBUILD(element1,element2,...)**

$LISTBUILD takes one or more expressions and returns an encoded list structure with one element for each expression. Elements are placed in the list in the order specified. Elements are counted from 1.

The following functions can be used to create a list:

- `$LISTBUILD`, which creates a list from multiple data items (strings or numerics), one list element per data item. `$LISTBUILD` can also be used to create list elements containing no data.
- `$LISTFROMSTRING`, which creates a list from a single string containing multiple delimited elements.
- `$LIST`, which extracts a sublist from an existing list.
- The null string (""") is also considered to be a valid list. The null string (""") is used to represent a null list, a list containing no elements. Because it contains no list elements, `$LISTLENGTH""` returns an element count of 0.
- Certain `$CHAR` non-printing character combinations, such as `$CHAR(1)`, `$CHAR(2,1)`, and `$CHAR(3,1,asciicode)` can also return an encoded empty or one-element list.

You can use the `$LISTVALID` function to determine if an expression is a valid list.

$LISTBUILD is used with the other $LIST functions: `$LISTDATA`, `$LISTFIND`, `$LISTGET`, `$LISTNEXT`, `$LISTLENGTH`, `$LISTSAME`, and `$LISTTOSTRING`. 
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 the same way, regardless of the hardware platform. Wide characters are represented as byte strings.

**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 when comparing encoded lists. 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") and $LISTBUILD(1.0) is not equal to $LISTBUILD("1.0"). However, list display functions, such as $LIST and $LISTTOSTRING return numeric list element values in canonical form. Therefore $LIST($LISTBUILD(1),1)=$LIST($LISTBUILD("1"),1).

For the same reason, an encoded list 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.

### SET $LISTBUILD

When used on the left side of the equal sign in a SET command, the $LISTBUILD function extracts multiple elements from a list as a single operation. The syntax is as follows:

```
SET $LISTBUILD(var1,var2,...)=list
```

The `var` arguments of $LISTBUILD are a comma-separated list of variables, each of which receives the element of `list` that corresponds to its position in the $LISTBUILD parameter list. The `var` arguments do not have to be existing variables; the variable is defined when SET $LISTBUILD assigns it a value.

- The number of `var` arguments may be less than or greater than the number of `list` elements. Unspecified `var` values retain their prior value; if previously undefined they remain undefined. Excess `list` elements are ignored.
- The `var` arguments and/or the `list` elements may contain omitted values, represented by placeholder commas. An omitted `var` argument is undefined. An omitted `list` element causes the corresponding `var` value to retain its prior value; if previously undefined it remains undefined.

$LISTBUILD is an atomic operation. The maximum number of `var` arguments is 128. Attempting to exceed this number results in a <SYNTAX> error.

If a `var` argument is an object property (object.property) the property must be multidimensional. Any property may be referenced as an $%property instance variable within an object method.

In the following examples, $LISTBUILD (on the right side of the equal sign) creates a list with four elements.

In the following example, SET $LISTBUILD extracts the first two elements from a list into two variables.

```
SET colorlist=$LISTBUILD("red","blue","green","white")
SET $LISTBUILD(a,b)=colorlist
WRITE "a="a," b="b   /* a="red" b="blue" */
```

In the following example, SET $LISTBUILD extracts elements from a list into five variables. Because the specified `list` does not have a 5th element, the corresponding `var` variable (`e`) contains its prior value:

```
SET (a,b,c,d,e)=0
SET colorlist=$LISTBUILD("red","blue","green","white")
SET $LISTBUILD(a,b,c,d,e)=colorlist
WRITE "a="a," b="b," c="c," d="d," e="e   /* a="red" b="blue" c="green" d="white" e=0 */
```

In the following example, SET $LISTBUILD extracts elements from a list into four variables. Because the specified `list` does not have a 3rd element, the corresponding `var` variable (`c`) contains its prior value:

```
SET (a,b,c,d,e)=0
SET colorlist=$LISTBUILD("red","blue","green","white")
SET $LISTBUILD(a,b,c,d)=colorlist
WRITE "a="a," b="b," c="c," d="d," e="e   /* a="red" b="blue" c="green" d="white" */
```

```
```
In the following example, $LISTBUILD extracts elements from a list into four variables. Because the 3rd list element value is a nested list, the corresponding var variable (c) contains a list value:

```objectscript
SET (a,b,c,d)=0
SET colorlist=$LISTBUILD("red","blue","green","yellow","white")
SET $LISTBUILD(a,b,c,d)=colorlist
WRITE "a="a," b="b," c="c," d="d
/* a="red" b="blue" c=$LB("green","yellow") d="white" */
```

**Examples**

Many of the examples shown here use the $LISTTOSTRING function to convert the $LISTBUILD return value for display; $LISTBUILD returns an encoded string that cannot be displayed directly.

The following example produces the three-element list "Red,Blue,Green":

```objectscript
SET colorlist=$LISTBUILD("Red","Blue","Green")
WRITE $LISTTOSTRING(colorlist,"^")
```

The following example creates a list of six numeric elements that display as "3^0^4^4^5^6^3^3^4^4^00". Note that $LISTBUILD encodes numeric element values based on an optimized binary representation, which may not be the same as canonical form. List display functions such as $LIST and $LISTTOSTRING return numeric element values in canonical form:

```objectscript
SET numlist=$LISTBUILD(003,0.00,44.0000000,5.6,33,4E2)
WRITE $LISTTOSTRING(numlist,"^")
```

**Omitting Elements**

Omitting an element expression defines an encoded element, but the data value of that element is undefined.

In the following example, the $LISTBUILD statements both produce a valid three-element list whose second element has an undefined value. Omitting an element and specifying an undefined variable for an element produces exactly the same result. In either case, referencing the second element with any list function (such as $LIST or $LISTTOSTRING) generates a <NULL VALUE> error:

```objectscript
KILL a
SET list1=$LISTBUILD("Red",,,"Green")
SET list2=$LISTBUILD("Red",a,"Green")
WRITE "List lengths:","LISTLENGTH(list1),","LISTLENGTH(list2),!
IF $LISTVALID(list1)=1,$LISTVALID(list2)=1 {
  WRITE "These are valid lists",!
} IF list1=list2 [WRITE "and they're identical"] ELSE [WRITE "They're not identical"]
```

The following example shows that an undefined element can be specified at the end of a list, as well as within a list. A list with trailing undefined elements is a valid list. However, referencing this undefined element with any list function generates a <NULL VALUE> error:

```objectscript
KILL z
SET list3=$LISTBUILD("Red",)
SET list4=$LISTBUILD("Red",z)
WRITE "List lengths:","LISTLENGTH(list3),","LISTLENGTH(list4),!
IF $LISTVALID(list3)=1,$LISTVALID(list4)=1 {
  WRITE "These are valid lists",!
} IF list3=list4 [WRITE "and they're identical"] ELSE [WRITE "They're not identical"]
```

However, the following example produces a three-element list whose second element has a data value: the empty string. No error condition occurs when referencing the second element:
SET list5=$LISTBUILD("Red","","Green")
SET list5len=$LISTLENGTH(list5)
WRITE "List length: ",list5len,
FOR i=1:1:list5len {
  WRITE "Element ",i," value: ",$LIST(list5,i),!
}

LISTS with No Data or Null String Data

Any list created using $LISTBUILD contains at least one encoded list element. That element may or may not contain data. Because $LISTLENGTH counts elements, not data, any list created using $LISTBUILD has a list length of at least 1.

Referencing a $LISTBUILD element whose data value is undefined generates a <NULL VALUE> error. The following are all valid $LISTBUILD statements that create "empty" lists. However, attempting to reference an element is such a list results in a <NULL VALUE> error:

TRY {
  SET x=$LISTBUILD(UndefinedVar)
  SET y=$LISTBUILD(,)
  SET z=$LISTBUILD()
  IF $LISTVALID(x)=1,$LISTVALID(y)=1,$LISTVALID(z)=1 {
    WRITE "These are valid lists",!
    WRITE "$LB(UndefinedVar) contains ",$LISTLENGTH(x)," elements",!
    WRITE "$LB(,) contains ",$LISTLENGTH(y)," elements",!
    WRITE "$LB() contains ",$LISTLENGTH(z)," elements",!
    /* Attempt to use null lists */
    WRITE "$LB(UndefinedVar) list value ",$LISTTOSTRING(x,"^"),!
    WRITE "$LB(,) list value ",$LISTTOSTRING(y,"^"),!
    WRITE "$LB() list value ",$LISTTOSTRING(z,"^"),!
  }
  CATCH exp {
    WRITE !,"In the CATCH block",!
    IF 1=exp.%IsA("%Exception.SystemException") {
      WRITE "System exception",!
      WRITE "Name: ",$ZCVT(exp.Name,"O","HTML"),!
      WRITE "Location: ",exp.Location,!
      WRITE "Code: 
    }
    ELSE {
      WRITE "Some other type of exception",! RETURN }
    WRITE exp.Code,!
    WRITE "Data: ",exp.Data,!
    RETURN
  }
}

The following are valid $LISTBUILD statements that create a list element that contains data, though this data has a null string value:

SET x=$LISTBUILD(""
WRITE "list contains ",$LISTLENGTH(x)," elements",!
WRITE "list value is ",$LISTTOSTRING(x,"^"),!
SET y=$LISTBUILD($CHAR(0))
WRITE "list contains ",$LISTLENGTH(y)," elements",!
WRITE "list value is ",$LISTTOSTRING(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":

SET nlist=$LISTBUILD("Apple","Pear",$LISTBUILD("Walnut","Pecan"))
WRITE "Nested list length is ",$LISTLENGTH($LIST(nlist,3)),!
WRITE "Full list length is ",$LISTLENGTH(nlist),!
WRITE "List is ",$LISTTOSTRING(nlist,"^")

Concatenating Lists

The result of concatenating two lists with the Concatenate operator (_) is a list that combines the two lists.

In the following example, concatenating two lists creates a list that is identical to a list with the same elements created using LISTBUILD:
You cannot concatenate a string to a list. Attempting to do so generates a <LIST> error the first time you attempt to access the result:

TRY {
  SET list1=$LISTBUILD("A","B")_"C"
  WRITE "$LISTBUILD completed without error",!
  SET listlen=$LISTLENGTH(list)
  WRITE "$LISTLENGTH completed without error",!
  SET listval=$LISTTOSTRING(list,"\^")
  WRITE "$LISTTOSTRING completed without error",!
}
CATCH exp { WRITE !!,"In the CATCH block",!
  IF 1=exp.%IsA("%Exception.SystemException") {
    WRITE "System exception",!
    WRITE "Name: ",ZCVT(exp.Name,"O","HTML"),!
    WRITE "Location: ",exp.Location,!
    WRITE "Code: 
  }
  ELSE { WRITE "Some other type of exception",! RETURN }
  WRITE exp.Code,!
  WRITE "Data: ",exp.Data,!
  RETURN
}

For further details on concatenation, see Operators in Using Caché ObjectScript.

See Also

- SET command
- ZZDUMP command
- $LIST function
- $LISTDATA function
- $LISTFROMSTRING function
- $LISTGET function
- $LISTLENGTH function
- $LISTNEXT
- $LISTSAME function
- $LISTTOSTRING function
- $LISTUPDATE function
- $LISTVALID function
$LISTDATA

Indicates whether the specified element exists and has a data value.

```
$LISTDATA(list,position,var)
$LD(list,position,var)
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>An expression that evaluates to a valid list.</td>
</tr>
<tr>
<td>position</td>
<td>Optional — An expression interpreted as a position in the specified list. Either a positive, non-zero integer or -1.</td>
</tr>
<tr>
<td>var</td>
<td>Optional — A variable that contains the element value at the specified list position. If $LISTDATA returns a value of a 1, var is written; if $LISTDATA returns a value of a 0, var is unchanged.</td>
</tr>
</tbody>
</table>

**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.

Optionally, $LISTDATA can write the element value to the `var` variable.

**Note:** $LISTDATA should not be used in a loop structure to return multiple successive element values. While this will work, it is highly inefficient, because $LISTDATA must evaluate the list from the beginning with each iteration. The $LISTNEXT function is a far more efficient way to return multiple successive element values.

**Parameters**

`list`

A list is an encoded string containing multiple elements. A list must have been created using $LISTBUILD or $LISTFROM-STRING, or extracted from another list using $LIST.

You can use the $LISTVALID function to determine if an expression is a valid list. If the expression in the `list` parameter does not evaluate to a valid list, a <LIST> error occurs. If a valid list contains no data at the specified position, $LISTDATA returns 0.

`position`

The integer position of the element in the list, counting from 1. 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.

$LISTDATA returns 0 if `position` refers to a nonexistent list member. A `position` of 0 always returns 0. If the value of `position` is less than -1, invoking the $LISTDATA function generates a <RANGE> error.

`var`

If $LISTDATA returns a value of a 1, Caché writes the value of the requested element to `var`. If $LISTDATA returns a value of a 0, `var` is unchanged. The `var` parameter can be a local, global, or process-private variable, with or without subscripts. It does not need to be defined; the first call to $LISTDATA that returns 1 defines and sets `var`. If the first call to $LISTDATA returns 0, `var` remains undefined.
The \textit{var} parameter cannot be a non-multidimensional object property. Attempting to write a value to a non-multidimensional object property results in an \texttt{<OBJECT DISPATCH>} error.

The \textit{var} parameter cannot be a special variable. Attempting to write a value to a special variable results in a \texttt{<SYNTAX>} error.

**Examples**

The following two examples show the results of the various values of the \textit{position} parameter.

The following $\texttt{LISTDATA}$ statements return a value of 0:

```
KILL y
SET x=$LISTBUILD("Red",y,"","Green",)
WRITE 1,$LISTDATA(x,2) ; second element is undefined
WRITE 1,$LISTDATA(x,3) ; third element is a killed variable
WRITE 1,$LISTDATA(x,-1) ; the last element is undefined
WRITE 1,$LISTDATA(x,0) ; the 0th position
WRITE 1,$LISTDATA(x,6) ; 6th position in 5-element list
```

The following $\texttt{LISTDATA}$ statements return a value of 1:

```
SET x=$LISTBUILD("Red",y,"","Green",)
WRITE 1,$LISTDATA(x) ; first element (by default)
WRITE 1,$LISTDATA(x,1) ; first element specified
WRITE 1,$LISTDATA(x,4) ; fourth element, value=null string
WRITE 1,$LISTDATA(x,5) ; fifth element
```

The following 3-parameter $\texttt{LISTDATA}$ statement tests for the presence of an element value and updates the \textit{evalue} variable with that value. Note that when $\texttt{LISTDATA}$ returns 0, \textit{evalue} remains unchanged:

```
SET x=$LISTBUILD("Red",y,"","Green",)
FOR i=1:1:$LISTLENGTH(x) {
    WRITE "element ",i," data? ",LISTDATA(x,i,evalue)," value ",evalue,!
}
WRITE " list elements"
```

All of the following $\texttt{LISTDATA}$ statements return a value of 0:

```
WRITE 1,$LISTDATA($LB()) ; null list
WRITE 1,$LISTDATA($LB(UndefinedVar)) ; null list
WRITE 1,$LISTDATA("") ; null string is a valid list
WRITE 1,$LISTDATA($LB(,)) ; but contain no data
WRITE 1,$LISTDATA($LB(,)) ; two-element null list
```

The following $\texttt{LISTDATA}$ statements return a value of 1:

```
WRITE 1,$LISTDATA($LB("")) ; data is null string
WRITE 1,$LISTDATA($LB($CHAR(0))) ; data is non-display character
```

**See Also**

- \texttt{LIST} function
- \texttt{LISTBUILD} function
- \texttt{LISTFIND} function
- \texttt{LISTFROMSTRING} function
- \texttt{LISTGET} function
- \texttt{LISTLENGTH} function
- \texttt{LISTNEXT}
- \texttt{LISTSAME} function
- \texttt{LISTTOSTRING} function
• $LISTUPDATE function
• $LISTVALID function
**$LISTFIND**

Searches a specified list for the requested value.

```
$LISTFIND(list,value,startafter)
$LF(list,value,startafter)
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>list</code></td>
<td>An expression that evaluates to a valid list. A list is an encoded string containing one or more elements. A list must be created using <code>$LISTBUILD</code> or <code>$LISTFROMSTRING</code>, or extracted from another list using <code>$LIST</code>.</td>
</tr>
<tr>
<td><code>value</code></td>
<td>An expression containing the desired element value.</td>
</tr>
<tr>
<td><code>startafter</code></td>
<td>Optional — An integer expression interpreted as a list position. The search starts with the element after this position; thus 0 means to start with position 1, 1 means to start with position 2. <code>startafter=-1</code> is a valid value, but always returns no match. Only the integer portion of the <code>startafter</code> value is used.</td>
</tr>
</tbody>
</table>

### Description

**$LISTFIND** searches the specified `list` for the first instance of the requested `value`. A match must be exact and consist of the full element value. Letter comparisons are case-sensitive. Numbers are compared in canonical form. If an exact match is found, **$LISTFIND** returns the position of the matching element. If `value` is not found, **$LISTFIND** returns a 0.

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 no match is found, **$LISTFIND** returns a 0. **$LISTFIND** will also return a 0 if the value of the `startafter` parameter refers to a nonexistent list member.

You can use the **$LISTVALID** function to determine if `list` is a valid list. If `list` is not a valid list, the system generates a `<LIST>` error.

If the value of the `startafter` parameter is less than -1, invoking the **$LISTFIND** function generates a `<RANGE>` error.

### Empty Strings and Empty Lists

The **$LISTFIND** function can be used to locate an empty string value, as shown in the following example:

```
SET x=$LISTBUILD("A","","C","D")
WRITE $LISTFIND(x,""); returns 2
```

**$LISTFIND** can be used with lists containing omitted elements, but cannot be used to locate an omitted element. The following example finds a value in a list with omitted elements:

```
SET x=$LISTBUILD("A","","C","D")
WRITE $LISTFIND(x,"C"); returns 3
```

The following **$LISTFIND** example returns 1:

```
WRITE $LISTFIND($LB(""),""); returns 1
```

The following **$LISTFIND** examples return 0:

```
WRITE $LISTFIND("",""),!; returns 0
WRITE $LISTFIND($LB(),""),!; returns 0
```
The following examples list consists of an empty list concatenated to a list containing data. Prepending the empty list changes the list position of elements in the resulting concatenated list:

```objectscript
SET x=$LISTBUILD("A","B","C","D")
WRITE $LISTFIND(x,"B")    ; returns 2
WRITE $LISTFIND("_"_x,"B"),! ; returns 2
WRITE $LISTFIND($LB()_x,"B"),! ; returns 3
WRITE $LISTFIND($LB(,,,)_x,"B")  ; returns 6
```

However, concatenating a null string to value has no effect on $LISTFIND:

```objectscript
SET x=$LISTBUILD("A","B","C","D")
WRITE $LISTFIND(x,"B"),!    ; returns 2
WRITE $LISTFIND(x,"B"_""),! ; returns 2
WRITE $LISTFIND(x,"_"_B""),! ; returns 2
```

## Examples

The following example returns 2, the position of the first occurrence of the requested string:

```objectscript
SET x=$LISTBUILD("A","B","C","D")
WRITE $LISTFIND(x,"B")
```

The following example returns 0, indicating the requested string was not found:

```objectscript
SET x=$LISTBUILD("A","B","C","D")
WRITE $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 the string occurs at the `startafter` position:

```objectscript
SET x=$LISTBUILD("A","B","C","D")
WRITE $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:

```objectscript
SET y=$LISTBUILD("A","B","C","A")
WRITE $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”:

```objectscript
SET mylist = $LISTBUILD("ABC","BCD","BBB")
WRITE $LISTFIND(mylist,"B")
```

The following numeric examples all return 0, because numbers are converted to canonical form before matching. In these cases, the string numeric value and the canonical form number do not match:

```objectscript
SET y=$LISTBUILD("1.0","+2","003","2*2")
WRITE $LISTFIND(y,1.0),!
WRITE $LISTFIND(y,2),!
WRITE $LISTFIND(y,003),!
WRITE $LISTFIND(y,4)
```

The following numeric examples match because numeric values are compared in their canonical forms:

```objectscript
SET y=$LISTBUILD(7.0,+6,005,2*2)
WRITE $LISTFIND(y,++7.000),! ; returns 1
WRITE $LISTFIND(y,0006),!    ; returns 2
WRITE $LISTFIND(y,8-3),!     ; returns 3
WRITE $LISTFIND(y,--4.0)    ; returns 4
```

The following examples all return 0, because the specified `startafter` value results in no match:
The following example shows how $LISTFIND can be used to find a nested list. Note that Caché treats a multi-element nested list as a single list element with a list value:

```objectscript
SET y=$LISTBUILD("A",$LB("x","y"),"C","D")
WRITE $LISTFIND(y,$LB("x","y"))
```

### See Also

- $LIST function
- $LISTBUILD function
- $LISTDATA function
- $LISTFROMSTRING function
- $LISTGET function
- $LISTLENGTH function
- $LISTNEXT
- $LISTSAME function
- $LISTTOSTRING function
- $LISTUPDATE function
- $LISTVALID function
$LISTFROMSTRING

Creates a list from a string.

$LISTFROMSTRING(string,delimiter)  
$LFS(string,delimiter)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A string to be converted into a Caché list. This string contains one or more elements, separated by a <em>delimiter</em>. The <em>delimiter</em> does not become part of the resulting Caché list.</td>
</tr>
<tr>
<td>delimiter</td>
<td>Optional — The delimiter used to separate substrings (elements) in <em>string</em>. Specify <em>delimiter</em> as a quoted string. If no <em>delimiter</em> is specified, the default is the comma (,) character.</td>
</tr>
</tbody>
</table>

**Description**

$LISTFROMSTRING takes a quoted string containing delimited elements and returns a 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 ObjectScript $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:

```
SET namestring="Deborah Noah Martha Bowie"
SET namelist=$LISTFROMSTRING(namestring," ")
WRITE !,"1st element: ",${LIST(namelist,1)}
WRITE !,"2nd element: ",${LIST(namelist,2)}
WRITE !,"3rd element: ",${LIST(namelist,3)}
```

**See Also**

- $LISTTOSTRING function
- $LISTBUILD function
- $LIST function
ObjectScript Functions

- `$PIECE` function
- `$LISTDATA` function
- `$LISTFIND` function
- `$LISTGET` function
- `$LISTLENGTH` function
- `$LISTNEXT` function
- `$LISTSAME` function
- `$LISTUPDATE` function
- `$LISTVALID` function
$LISTGET

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**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>list</strong></td>
<td>An expression that evaluates to a valid list.</td>
</tr>
<tr>
<td><strong>position</strong></td>
<td>Optional — An integer code specifying the starting position in list. Permitted values are n (count from beginning of list), * (last element in list), and *-n (relative offset count backwards from end of list). Thus, the first element in the list is 1, the second element is 2, the last element in the list is *, and the next-to-last element is *-1. If position is a fractional number, it is truncated to its integer part. If position is omitted, it defaults to 1. -1 may be used in older code to specify the last element in the list. This deprecated use of -1 should not be combined with * or *-n relative offset syntax.</td>
</tr>
<tr>
<td><strong>default</strong></td>
<td>Optional — An expression that provides the value to return if the list element has an undefined value. If default is omitted, it defaults to the null string (&quot;&quot;&quot;). You must specify a position parameter value to specify a default value.</td>
</tr>
</tbody>
</table>

**Description**

$LISTGET returns the requested element in the specified list. If the value of position refers to a nonexistent element or identifies an element with an undefined value, the 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**

**list**

A list can be created using $LISTBUILD or $LISTFROMSTRING, or extracted from another list using $LIST. The null string (""") is also treated as a valid list. You can use $LISTVALID to determine if list is a valid list. An invalid list causes $LISTGET to generate a <LIST> error.

**position**

The position (element count) of the list element to return. An element is returned as a string. List elements are counted from 1. If position is omitted, $LISTGET returns the first element.

- If position is n (a positive integer), $LISTGET counts elements from the beginning of list. If position is greater than the number of elements in list, $LISTGET returns the default value.
- If position is *, $LIST returns the last element in list.
- If position is *-n (an asterisk followed by a negative integer), $LIST counts elements by relative offset from the end of list. Thus, *-0 is the last element in the list, *-1 is the next-to-last list element (an offset of 1 from the end). If the *-n offset specifies the position before the first element of list (for example, *-3 for a 3–element list), $LISTGET returns the default value. If the *-n offset specifies a position prior to that (for example, *-4 for a 3–element list), Caché issues a <RANGE> error.
• If position is 0 or -0, $LISTGET returns the default value.

The numeric portion of the position parameter evaluates to an integer. Caché truncates a fractional number to its integer portion. Specifying position as -1 (indicating last element of the list) is deprecated and should not be used in new code; a position negative number less than -1 generates a <RANGE> error.

When using a variable to specify *-n you must always specify the asterisk and a sign character in the parameter itself.

The following are valid specifications of *-n:

```objectscript
SET count=2
SET alph=$LISTBUILD("a","b","c","d")
WRITE $LISTGET(alph,*-count,"blank")

SET count=-2
SET alph=$LISTBUILD("a","b","c","d")
WRITE $LISTGET(alph,++count,"blank")
```

default

An expression that evaluates to a string or numeric value. $LISTGET returns default if the element specified by position does not exist. This may occur if position is beyond the end of the list, if position specifies an element that has no value, if position is 0, or if list contains no elements. However, if a position of *-n specifies a position before the 0th element of list, Caché issues a <RANGE> error.

If you omit the default parameter, the null string (""") is returned as the default value.

Examples

The $LISTGET functions in the following example return the value of the list element specified by position (the position default is 1):

```objectscript
SET list=$LISTBUILD("A","B","C")
WRITE !,$LISTGET(list)     ; returns "A"
WRITE !,$LISTGET(list,1)   ; returns "A"
WRITE !,$LISTGET(list,3)   ; returns "C"
WRITE !,$LISTGET(list,*-1) ; returns "B"
```

The $LISTGET functions in the following example return a value upon encountering the undefined 2nd element in the list. The first two returns a question mark (?), which the user has defined as the default value. The second two returns a null string because the user has not specified a default value:

```objectscript
WRITE "returns:",$LISTGET($LISTBUILD("A",,"C"),2,"?"),!
WRITE "returns:",$LISTGET($LISTBUILD("A",,"C"),*-1,"?"),!
```

The following example returns all of the element values in the list. It also lists the positions before and after the ends of the list. Where a value is non-existent, it returns the default value:

```objectscript
SET list=$LISTBUILD("a","b",,"d",,"g")
SET llen=$LISTLENGTH(list)
FOR x=0:1:llen+1 {
  WRITE "position ",x,"=",$LISTGET(list,x," no value"),!
}!
WRITE "end of the list"
```

The following example returns all of the element values in the list in reverse order. Where a value is omitted, it returns the default value:

```objectscript
SET list=$LISTBUILD("a","b",,"d",,"g")
SET llen=$LISTLENGTH(list)
FOR x=0:1:llen {
  WRITE "position ",x,"=",$LISTGET(list,*-x," no value"),!
}!
WRITE "beginning of the list"
```
The `$LISTGET` functions in the following example return the *list* element value of the null string; they do not return the *default* value:

```objectscript
WRITE "returns:","LISTGET("");,1,"no value"),!  
WRITE "returns:","LISTGET("");,"no value"),!  
WRITE "returns:","LISTGET("");,"-0,"no value"
```

The `$LISTGET` functions in the following example all return the *default* value:

```objectscript
WRITE $LISTGET("",1,"no value"),!  
WRITE $LISTGET($LB(),1,"no value"),!  
WRITE $LISTGET($LB(UndefinedVar),1,"no value"),!  
WRITE $LISTGET($LB(,),1,"no value"),!  
WRITE $LISTGET($LB(,),'"no value"'),!  
WRITE $LISTGET($LB(,),'-1,"no value"'),!  
WRITE $LISTGET($LB(""),2,"no value"),!  
WRITE $LISTGET($LB(""),'-1,"no value"')
```

**See Also**

- `$LIST` function
- `$LISTBUILD` function
- `$LISTDATA` function
- `$LISTFIND` function
- `$LISTFROMSTRING` function
- `$LISTLENGTH` function
- `$LISTNEXT`
- `$LISTSAME` function
- `$LISTTOSTRING` function
- `$LISTUPDATE` function
- `$LISTVALID` function
$LISTLENGTH

Returns the number of elements in a specified list.

$LISTLENGTH(list)
$LL(list)

Parameter

| list | Any expression that evaluates to a list. A list can be created using $LISTBUILD or $LISTFROMSTRING, or extracted from another list using $LIST. |

Description

$LISTLENGTH returns the number of elements in list. $LISTLENGTH counts as a list element every designated list position, whether or not that position contains data.

You can use the $LISTVALID function to determine if list is a valid list. If list is not a valid list, the system generates a <LIST> error.

An “empty” list created by $LISTBUILD defines an encoded list element, although that list element contains no data. Because $LISTLENGTH counts list elements (not elements containing data), an “empty” list has a $LISTLENGTH count of 1.

The null string (""") is used to represent a null list, a list containing no elements. Because it contains no list elements, it has a $LISTLENGTH count of 0.

Examples

The following example returns 3, because there are 3 elements in the list:

WRITE $LISTLENGTH($LISTBUILD("Red","Blue","Green"))

The following example also returns 3, even though the second element in the list contains no data:

WRITE $LISTLENGTH($LISTBUILD("Red","Green"))

The following examples all return 1. $LISTLENGTH makes no distinction between an empty list element and a list element containing data:

WRITE $LISTLENGTH($LB()),!
WRITE $LISTLENGTH($LB( UndefinedVar)),!
WRITE $LISTLENGTH($LB("")),!
WRITE $LISTLENGTH($LB($CHAR(0))),!
WRITE $LISTLENGTH($LB("John Smith"))

The following example returns 0. $LISTVALID considers the null string a valid list, but it contains no list elements:

WRITE $LISTLENGTH(""")

The following example returns 3, because the two placeholder commas represent 3 empty list elements:

WRITE $LISTLENGTH($LB(,,))

$LISTLENGTH and Concatenation

Concatenating two lists always results in a $LISTLENGTH equal to the sum of the lengths of the lists. This is true even when concatenating empty lists, or concatenating a null string to a list.

The following example all return a list length of 3:
WRITE $LISTLENGTH($LB()_$LB("a","b")),!
WRITE $LISTLENGTH($LB("a")_$LB(UndefinedVar)_$LB("c")),!
WRITE $LISTLENGTH($LB("")_$LB(_$LB(UndefinedVar))),!
WRITE $LISTLENGTH($LB("a","b","c")),!
WRITE $LISTLENGTH($LB("a","b")_$LB("c"))

$LISTLENGTH and Nested Lists

The following example returns 3, because $LISTLENGTH does not recognize the individual elements in a nested list, and treats it as a single list element:

WRITE $LISTLENGTH($LB("Apple","Pear","Walnut","Pecan")))

The following examples all return 1, because $LISTLENGTH counts only the outermost nested list:

WRITE $LISTLENGTH($LB($LB($LB()))),!
WRITE $LISTLENGTH($LB($LB($LB("Fred"))))),!
WRITE $LISTLENGTH($LB($LB("Barney"_$LB("Fred"))))),!
WRITE $LISTLENGTH($LB("Fred"_$LB("Barney"_$LB("Wilma"))))

See Also

• $LIST function
• $LISTBUILD function
• $LISTDATA function
• $LISTNEXT
• $LISTFROMSTRING function
• $LISTGET function
• $LISTSAME function
• $LISTTOSTRING function
• $LISTUPDATE function
• $LISTVALID function
**$LISTNEXT**

Retrieves elements sequentially from a list.

$LISTNEXT(list,ptr,value)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>Any expression that evaluates to a list.</td>
</tr>
<tr>
<td>ptr</td>
<td>A pointer to the next element in the list. You must specify <code>ptr</code> as a local variable initialized to 0. This value points to the beginning of <code>list</code>. Caché increments <code>ptr</code> using an internal address value algorithm (not a predictable integer counter). Therefore, the only value you can use to set <code>ptr</code> is 0. <code>ptr</code> cannot be a global variable or a subscripted variable.</td>
</tr>
<tr>
<td>value</td>
<td>A local variable used to hold the data value of a list element. <code>value</code> does not have to be initialized before invoking <code>$LISTNEXT</code>. <code>value</code> cannot be a global variable or a subscripted variable.</td>
</tr>
</tbody>
</table>

**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.

**Note:** Because `ptr` is an index into the internal structure of `list`, the list should not be modified while `$LISTNEXT` is being used on it. Modifying `list` may make the `ptr` value invalid and cause the next invocation of `$LISTNEXT` to issue a `<FUNCTION>` error.

You can use `$LISTVALID` to determine if `list` is a valid list. An invalid list causes `$LISTNEXT` to generate a `<LIST>` error.

When `$LISTNEXT` encounters an omitted list element (an element with a null value), it returns 1 indicating that a list element has been successfully retrieved, advances `ptr` to the next element, and resets `value` to be an undefined variable. This can happen when encountering an omitted list element, such as the second invocation of `$LISTNEXT` on `list=$LB("a","b")`, or with any of the following valid lists: `list=$LB()`, `list=$LB(UndefinedVar)`, or `list=$LB(,)`.

`$LISTNEXT("",ptr,value)` returns 0, and does not advance the pointer or set `value`.

`$LISTNEXT($LB(""),ptr,value)` returns 1, advances the pointer, and sets `value` to the null string ("").

**$LISTNEXT and Performance**

A Caché list is the most efficient way to process large numbers of data values. Because long strings are active by default, you can use lists to hold large numbers of values for processing, rather than using an array or other data structure. Using `$LISTNEXT` to return a large number of elements from a list is substantially more efficient than using `$LIST` to perform the same operation.

The following example rapidly accesses the elements in `mylist`:

```ObjectScript
SET ptr=0
WHILE $LISTNEXT(mylist,ptr,value) {
    /* perform some operation on value */
}
```

It is substantially faster than the following equivalent example:
FOR i=1:1:$LISTLENGTH(mylist) {
    SET value=$LIST(mylist,i)
    /* perform some operation on value */
}

$LISTNEXT and Nested Lists

The following example returns three elements, because $LISTNEXT does not recognize the individual elements in nested lists:

```
SET list=$LISTBUILD("Apple","Pear",$LISTBUILD("Walnut","Pecan"))
SET ptr=0,count=0
WHILE $LISTNEXT(list,ptr,value) {
    SET count=count+1
    WRITE !,value
}
WRITE !,"End of list: ",count," elements found"
QUIT
```

Example

The following example sequentially returns all the elements in the list. When it encounters an omitted element, the $SELECT returns the default value "omitted":

```
SET list=$LISTBUILD("Red","Blue",,"Green")
SET ptr=0,count=0
WHILE $LISTNEXT(list,ptr,value) {
    SET count=count+1
    WRITE !,count," : ",$SELECT($DATA(value):value,1:"omitted")
}
WRITE !,"End of list: ",count," elements found"
QUIT
```

See Also

- $LIST function
- $LISTBUILD function
- $LISTDATA function
- $LISTFIND function
- $LISTFROMSTRING function
- $LISTGET function
- $LISTLENGTH function
- $LISTSAME function
- $LISTTOSTRING function
- $LISTUPDATE function
- $LISTVALID function
$LISTSAME

Compares two lists and returns a boolean value.

$LISTSAME(list1, list2)
$LS(list1, list2)

**Parameters**

| list1 | Any expression that evaluates to a list. A list can be created using $LISTBUILD or $LISTFROMSTRING, or extracted from another list using $LIST. The null string (""") is also treated as a valid list. |
| list2 | Any expression that evaluates to a list. A list can be created using $LISTBUILD or $LISTFROMSTRING, or extracted from another list using $LIST. The null string (""") is also treated as a valid 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. $LISTSAME compares list elements using their string representations. $LISTSAME comparisons are case-sensitive.

$LISTSAME compares the two lists element-by-element in left-to-right order. Therefore $LISTSAME returns a value of 0 when it encounters the first non-identical pair of list elements; it does not check subsequent items to determine if they are valid list elements. If a $LISTSAME comparison encounters an invalid item, it issues a <LIST> error.

**Examples**

The following example returns 1, because the two lists are identical:

```ObjectScript
SET x = $LISTBUILD("Red","Blue","Green")
SET y = $LISTBUILD("Red","Blue","Green")
WRITE $LISTSAME(x,y)
```

The following example returns 0, because the two lists are not identical:

```ObjectScript
SET x = $LISTBUILD("Red","Blue","Yellow")
SET y = $LISTBUILD("Red","Yellow","Blue")
WRITE $LISTSAME(x,y)
```

**Identical Lists**

$LISTSAME considers two lists to be identical if the string representations of the two lists are identical.

When comparing a numeric list element and a string list element, the string list element must represent the numeric in canonical form; this is because Caché always reduces numerics to canonical form before performing a comparison. In the following example, $LISTSAME compares a string and a numeric. The first three $LISTSAME functions return 1 (identical); the fourth $LISTSAME function returns 0 (not identical) because the string representation is not in canonical form:

```ObjectScript
WRITE $LISTSAME($LISTBUILD("365"),$LISTBUILD(365)),!
WRITE $LISTSAME($LISTBUILD("365"),$LISTBUILD(365.0)),!
WRITE $LISTSAME($LISTBUILD("365.5"),$LISTBUILD(365.5)),!
WRITE $LISTSAME($LISTBUILD("365.0"),$LISTBUILD(365.0))
```

$LISTSAME comparison 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:
The equality (=) 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:

```objectscript
SET x = $LISTBUILD("360", "361", "362", "363", "364", "365", "366")
SET y = $LISTBUILD(00360.000, (19*19), +"362", 363, 364.0, +365,"3\_66")
WRITE !,$LISTSAME(x,y)," lists are identical"
```

### Numeric Maximum

A number larger than 2**63 (9223372036854775810) or smaller than -2**63 (–9223372036854775808) exceeds the maximum numeric range for \$LISTSAME list comparison. \$LISTSAME returns 0 when such extremely large numbers are compared, as shown in the following example:

```objectscript
SET bignum=$LISTBUILD(9223372036854775810)
SET bigstr=$LISTBUILD("9223372036854775810")
WRITE $LISTSAME(bignum,bigstr),!
SET bignum=$LISTBUILD(9223372036854775811)
SET bigstr=$LISTBUILD("9223372036854775811")
WRITE $LISTSAME(bignum,bigstr)
```

### 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:

```objectscript
WRITE !,$LISTSAME($LISTBUILD("")$LISTBUILD("")," null lists"
WRITE !,$LISTSAME("","")," null strings"
WRITE !,$LISTSAME($LISTBUILD("")$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 (values not identical). 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:

```objectscript
WRITE !,$LISTSAME("","abc")
WRITE !,$LISTSAME("abc",""
```

The following \$LISTSAME comparisons do issue a <LIST> error, because a list (even a null list) cannot be compared with a string:

```objectscript
SET x = $LISTBUILD(""
WRITE !,$LISTSAME("abc",x)
WRITE !,$LISTSAME(x,"abc"
```

### Comparing “Empty” Lists

\$LISTVALID considers all of the following as valid lists:

```objectscript
WRITE $LISTVALID(""!),
WRITE $LISTVALID($LB()),!
WRITE $LISTVALID($LB(UndefinedVar)),!
WRITE $LISTVALID($LB("")),!
WRITE $LISTVALID($LB($CHAR(0))),!
WRITE $LISTVALID($LB(),)
```

\$LISTSAME considers only the following pairs as identical:
Empty Elements

A $LISTBUILD can create empty elements by including extra commas between elements or appending one or more commas to either end of the element list. $LISTSAME is aware of empty elements, and does not treat them as equivalent to null string elements.

The following $LISTSAME examples all return 0 (not identical):

- WRITE $LISTSAME($LISTBUILD(365,,367),$LISTBUILD(365,367)),!
- WRITE $LISTSAME($LISTBUILD(365,366,),$LISTBUILD(365,366)),!
- WRITE $LISTSAME($LISTBUILD(365,,367),$LISTBUILD(365,"",367)),

$DOUBLE List Elements

$LISTSAME considers all forms of zero to be identical: 0, –0, $DOUBLE(0), and $DOUBLE(-0).

$LISTSAME considers a $DOUBLE("NAN") list element to be identical to another $DOUBLE("NAN") list element. However, because NAN (Not A Number) cannot be meaningfully compared using numerical operators, Caché operations (such as equal to, less than, or greater than) that attempt to compare $DOUBLE("NAN") to another $DOUBLE("NAN") fail, as shown in the following example:

```objectscript
SET x = $DOUBLE("NAN")
SET a = $LISTBUILD(1,2,x)
SET b = $LISTBUILD(1,2,x)
WRITE !,$LISTSAME(a,b)    /* 1 (NAN list elements same) */
WRITE !,x=x               /* 0 (NAN values not equal) */
```

Nested and Concatenated Lists

$LISTSAME does not support nested lists. It cannot compare two lists that contain lists, even if their contents are identical.

```objectscript
SET x = $LISTBUILD("365")
SET y = $LISTBUILD(365)
WRITE !,$LISTSAME(x,y)," lists identical"
WRITE !,$LISTSAME($LISTBUILD(x),$LISTBUILD(y))," nested lists not identical"
```

In the following example, both $LISTSAME comparisons returns 0, because these lists are not considered identical:

```objectscript
SET x=$LISTBUILD("Apple","Pear","Walnut","Pecan")
SET y=$LISTBUILD("Apple","Pear","Walnut","Pecan")
SET z=$LISTBUILD("Apple","Pear","Walnut","Pecan","")
WRITE !,$LISTSAME(x,y)," nested list"
WRITE !,$LISTSAME(x,z)," null string is list item"
```

$LISTSAME does support concatenated lists. The following example returns 1, because the lists are considered identical:

```objectscript
SET x=$LISTBUILD("Apple","Pear","Walnut","Pecan")
SET y=$LISTBUILD("Apple","Pear")_$LISTBUILD("Walnut","Pecan")
WRITE !,$LISTSAME(x,y)," concatenated list"
```

See Also

- $LIST function
- $LISTBUILD function
- $LISTDATA function
- $LISTFIND function
- $LISTFROMSTRING function
- $LISTGET function
• $LISTLENGTH function
• $LISTNEXT function
• $LISTTOSTRING function
• $LISTUPDATE function
• $LISTVALID function
• $DOUBLE function
$LISTTOSTRING

Creates a string from a list.

$LISTTOSTRING(list, delimiter, flag)
$LTS(list, delimiter, flag)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>A Caché list, created using $LISTBUILD or $LISTFROMSTRING, or extracted from another list using $LIST.</td>
</tr>
<tr>
<td>delimiter</td>
<td>Optional — A delimiter used to separate substrings. Specify delimiter as a quoted string. If no delimiter is specified, the default is the comma (,) character.</td>
</tr>
<tr>
<td>flag</td>
<td>Optional — A boolean value that specifies how to handle an omitted list element. 0 issues a &lt;NULL VALUE&gt; error. 1 inserts an empty string for the element. The default is 0.</td>
</tr>
</tbody>
</table>

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.

```objectscript
SET x=$CHAR(0,0,0,1,16,27,134,240)
SET a=$LISTTOSTRING(x,"","")  // generates a <LIST> error
```

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, specify the quote character twice ("";) or use $CHAR(34).
flag

A boolean flag used to specify how to handle omitted elements in list. In the following example, the $LISTBUILD creates a list with an omitted element, and the flag=1 option is specified to handle this list element:

```ObjectScript
SET colorlist=$LISTBUILD("Red","Blue")
WRITE $LISTTOSTRING(colorlist,,1)
```

If the flag option was omitted or set to 0, the $LISTTOSTRING would generate a <NULL VALUE> error.

Note that if flag=1, an element with an empty string value is indistinguishable from an omitted element. Thus $LISTBUILD("Red",","Blue") and $LISTBUILD ("Red","Blue") would return the same $LISTTOSTRING value. This flag=1 behavior is compatible with the implementation of $LISTTOSTRING in Caché SQL, as described in the Caché SQL Reference.

Examples

The following example creates a list of four elements, then converts it to a string with the elements delimited by the colon (:) character:

```ObjectScript
SET namelist=$LISTBUILD("Deborah","Noah","Martha","Bowie")
WRITE $LISTTOSTRING(namelist,"*")
```

returns "Deborah:Noah:Martha:Bowie"

The following example creates a list of four elements, then converts it to a string with the elements delimited by the *sp* string:

```ObjectScript
SET namelist=$LISTBUILD("Deborah","Noah","Martha","Bowie")
WRITE $LISTTOSTRING(namelist,"*sp*")
```

returns "Deborah*sp*Noah*sp*Martha*sp*Bowie"

The following example creates a list with one omitted element and one element with an empty string value. $LISTTOSTRING converts it to a string with the elements delimited by the colon (:) character. Because of the omitted element, flag=1 is required to avoid a <NULL VALUE> error. However, when flag=1, the omitted element and the empty string value are indistinguishable:

```ObjectScript
SET namelist=$LISTBUILD("Deborah",","","Bowie")
WRITE $LISTTOSTRING(namelist,":",1)
```

returns "Deborah:::Bowie"

$LISTVALID considers all of the following valid lists. With flag=1, $LISTTOSTRING returns the null string ("") for all of them:

- WRITE "1",$LISTTOSTRING("",1)!
- WRITE "2",$LISTTOSTRING($LB(),1)!
- WRITE "3",$LISTTOSTRING($LB(UndefinedVar),1)!
- WRITE "4",$LISTTOSTRING($LB(""),1)

With flag=0, $LISTTOSTRING returns the null string ("") for only the following:

- WRITE "1",$LISTTOSTRING("",0)!
- WRITE "4",$LISTTOSTRING($LB(""),0)

The others generate a <NULL VALUE> error.

See Also

- $LISTFROMSTRING function
- $LISTBUILD function
ObjectScript Functions

- $LIST function
- $PIECE function
- $LISTDATA function
- $LISTFIND function
- $LISTGET function
- $LISTLENGTH function
- $LISTNEXT function
- $LISTSAME function
- $LISTUPDATE function
- $LISTVALID function
$LISTUPDATE

Updates a list by optionally replacing a specified list element or sequence of elements.

$LISTUPDATE(list,position,bool:val...)
$LU(list,position,bool:val...)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>Any expression that evaluates to a list. A list can be created using $LISTBUILD or $LISTFROMSTRING, or extracted from another list using $LIST. The null string (&quot;&quot;) is also treated as a valid list.</td>
</tr>
<tr>
<td>position</td>
<td>A positive integer specifying the position in list to update, counting from 1. If position is larger than the number of elements in list, $LISTUPDATE appends the element, padding if necessary.</td>
</tr>
<tr>
<td>bool</td>
<td>Optional — A boolean variable specifying whether or not to update the specified list element. If omitted, bool defaults to 1, causing this element to be updated.</td>
</tr>
<tr>
<td>value</td>
<td>The value used to update the list at the specified position. You can specify a comma-separated list of value parameters or bool:value pair parameters in any combination.</td>
</tr>
</tbody>
</table>

Description

$LISTUPDATE returns a copy of a list updated by replacing or adding one or more list elements by position. The position specifies the position in the list at which to begin updating elements. Elements are updated sequentially, starting from this position. The position can be larger than the number of elements in the list. If so, additional null elements are added to the list (if necessary) to insert a new element at the specified position. Note that position must be a positive integer;

$LISTUPDATE cannot insert a new element at the beginning of the list. Setting position=0 performs no operation.

$LISTUPDATE also cannot specify end-of-list, except by position count from the beginning of the list.

$LISTUPDATE can specify or more than one bool:value pairs. Multiple bool:value pairs are separated by commas. These bool:value pairs update sequential elements, beginning with the position element. If bool=1, Caché updates that element with value; if bool=0, Caché does not update that element and proceeds to the next element.

Sequential elements that are not to be updated in a sequence of bool:value pairs do not have to be specified; they can be represented by placeholder commas. The bool: parameter is optional for each value; if omitted, it defaults to 1. If you omit bool also omit the colon (:) separator character. Thus the following are permitted ways to specify a bool:value pair:

- Element to update: either 1:"newval", or "newval" (bool defaults to 1).
- Element to not update: either 0:"newval", or a placeholder comma.

$LISTUPDATE is commonly used to return an updated version of an existing list. You can use $LISTUPDATE to create a list by specifying list="".

You can also use SET $LIST to update one or more elements in an existing list.

Examples

The following example replaces the list element at position 2 with the specified value:

```
SET caps=1
SET mylist = $LISTBUILD("Red","White","Blue")
SET newlist = $LISTUPDATE(mylist,2,caps:"WHITE")
ZWRITE newlist
```

The following example does not replace the list element at position 2 with the specified value:
SET caps=0
SET mylist = $LISTBUILD("Red","White","Blue")
SET newlist = $LISTUPDATE(mylist,2,caps:"WHITE")
ZWRITE newlist

The following example replaces the list element at position 2 with null:

SET caps=1
SET mylist = $LISTBUILD("Red","White","Blue")
SET newlist = $LISTUPDATE(mylist,2,caps:"")
ZWRITE newlist

The following example appends the list at position 7 with the specified value, padding null elements at positions 5 and 6:

SET bool=1
SET mylist = $LISTBUILD("Red","Orange","Yellow","Green")
SET newlist = $LISTUPDATE(mylist,7,bool:"Purple")
ZWRITE newlist

The following example does not append the list at position 7 with the specified value. The unchanged list is returned with no null element padding:

SET bool=0
SET mylist = $LISTBUILD("Red","Orange","Yellow","Green")
SET newlist = $LISTUPDATE(mylist,7,bool:"Purple")
ZWRITE newlist

The following three examples are all functionally identical. Each replaces the elements at positions 2 and 4, and appends a new element at position 7. It does not replace elements 3 and 5. Element 6 is created as a null element:

SET mylist = $LISTBUILD("Red","Orange","Yellow","Green","Blue")
SET newlist = $LISTUPDATE(mylist,2,1:"ORANGE",0:"YELLOW",
                           1:"GREEN",0:"BLUE",
                           0:"INDIGO",1:"VIOLET")
ZWRITE newlist

Here the bool parameter is only specified when it is 0:

SET mylist = $LISTBUILD("Red","Orange","Yellow","Green","Blue")
SET newlist = $LISTUPDATE(mylist,2,"ORANGE",0:"YELLOW",
                           0:"INDIGO","VIOLET")
ZWRITE newlist

Here placeholder commas are used to skip elements that are not updated:

SET mylist = $LISTBUILD("Red","Orange","Yellow","Green","Blue")
SET newlist = $LISTUPDATE(mylist,2,"ORANGE",",","GREEN",",","VIOLET")
ZWRITE newlist

See Also

- $LIST function
- $LISTBUILD function
- $LISTDATA function
- $LISTFIND function
- $LISTFROMSTRING function
- $LISTGET function
- $LISTLENGTH function
- $LISTNEXT function
- $LISTSAME function
- $LISTTOSTRING function
- $LISTVALID function
$LISTVALID

Determines if an expression is a list.

$LISTVALID(exp)
$LV(exp)

Parameter

| exp   | Any valid expression. |

Description

$LISTVALID determines whether exp is a 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. (Certain $CHAR non-printing character combinations, such as $CHAR(1), $CHAR(2,1), and $CHAR(3,1,asciicode) can also return a valid empty or one-element list.)

Examples

The following examples all return 1, indicating a valid list:

```
SET w = $LISTBUILD("Red","Blue","Green")
SET x = $LISTBUILD("Red")
SET y = $LISTBUILD(365)
SET z = $LISTBUILD(""")
WRITE 1,$LISTVALID(w)
WRITE 1,$LISTVALID(x)
WRITE 1,$LISTVALID(y)
WRITE 1,$LISTVALID(z)
```

The following examples all return 0. Numbers and strings (with the exception of the null string) are not valid lists:

```
SET x = "Red"
SET y = 44
WRITE 1,$LISTVALID(x)
WRITE 1,$LISTVALID(y)
```

The following examples all return 1. Concatenated, nested, and omitted value lists are all valid lists:

```
SET w=$LISTBUILD("Apple","Pear")
SET x=$LISTBUILD("Walnut","Pecan")
SET y=$LISTBUILD("Apple","Pear",$LISTBUILD("Walnut","Pecan"))
SET z=$LISTBUILD("Apple","Pear","Pecan")
WRITE 1,$LISTVALID(w,x) ; concatenated
WRITE 1,$LISTVALID(y)   ; nested
WRITE 1,$LISTVALID(z)   ; omitted element
```

The following examples all return 1. $LISTVALID considers all of the following “empty” lists as valid lists:

```
WRITE $LISTVALID(""),!
WRITE $LISTVALID($LB()),!
WRITE $LISTVALID($LB(UndefinedVar)),!
WRITE $LISTVALID($LB("")),!
WRITE $LISTVALID($LB($CHAR(0))),!
WRITE $LISTVALID($LB(,))
```

See Also

- $LIST function
- $LISTBUILD function
• $LISTDATA function
• $LISTFIND function
• $LISTFROMSTRING function
• $LISTGET function
• $LISTLENGTH function
• $LISTNEXT function
• $LISTSAME function
• $LISTTOSTRING function
• $LISTUPDATE function
$LOCATE

Locates the first match of a regular expression in a string.

$LOCATE(string,regexp,start,end,val)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>string</strong></td>
<td>The string to be matched.</td>
</tr>
<tr>
<td><strong>regexp</strong></td>
<td>A regular expression to match against <em>string</em>. A regular expression consists of one or more meta-characters, and may also contain literal characters.</td>
</tr>
<tr>
<td><strong>start</strong></td>
<td>Optional — An integer specifying the starting position within <em>string</em> from which to match the <em>regexp</em>. If you omit <em>start</em>, matching begins at the beginning of <em>string</em>. If you omit <em>start</em> and specify <em>end</em> and/or <em>val</em>, you must specify the place-holder comma.</td>
</tr>
<tr>
<td><strong>end</strong></td>
<td>Optional — $LOCATE assigns an integer value to this variable if the match is successful. This integer is the next character position after the matched string. Caché passes <em>end</em> by reference. This parameter must be a local variable. It cannot be an array, a global variable, or a reference to an object property.</td>
</tr>
<tr>
<td><strong>val</strong></td>
<td>Optional — $LOCATE assigns a string value to this variable if the match is successful. This string consists of the matched substring. Caché passes <em>val</em> by reference. This parameter must be a local variable. It cannot be an array, a global variable, or a reference to an object property.</td>
</tr>
</tbody>
</table>

**Description**

$LOCATE matches a regular expression against successive substrings of a specified string. It returns an integer specifying the starting location of the first *regexp* match within *string*. It counts locations from 1. It returns 0 if *regexp* does not match any subset of *string*.

Optionally, it can also assign the matching substring to a variable.

If you omit an optional parameter and specify a later parameter, you must specify the appropriate place-holder comma(s).

ObjectScript support for regular expressions consists of the $LOCATE and $MATCH functions:

- **$LOCATE** matches a regular expression to successive substrings of *string* and returns the location (and, optionally, the value) of the first match.

- **$MATCH** matches a regular expression to the full *string* and returns a boolean indicating whether a match occurred.

The Locate() method of the %Regex.Matcher class provides similar functionality as $LOCATE. The %Regex.Matcher class methods provide substantial additional functionality for using regular expressions.

Other ObjectScript matching operations use Caché pattern match operators.

**Parameters**

**string**

An expression that evaluates to a *string*. The expression can be specified as the name of a variable, a numeric value, a string literal, or any valid ObjectScript expression. A *string* can contain control characters.
If `string` is the empty string and `regexp` cannot match the empty string, `$LOCATE` returns 0; `end` and `val` are not set.

If `string` is the empty string and `regexp` can match the empty string, `$LOCATE` returns 1; `end` is set to 1, and `val` is set to the empty string.

**regexp**

A regular expression used to match against `string` to locate the desired substring. A regular expression is an expression that evaluates to a `string` consisting of some combination of meta-characters and literals. Meta-characters specify character types and match patterns. Literals specify one or more matching single characters, ranges of characters, or substrings. An extensive regular expression syntax is supported. For details, refer to the “Regular Expressions” chapter of Using Caché ObjectScript.

**start**

An integer specifying the starting position within `string` from which to match the `regexp`. 1 or 0 specify starting at the beginning of `string`. A `start` value equal to the length of `string` + 1 always returns 0. A `start` value greater than the length of `string` + 1 generates a `<REGULAR EXPRESSION>` error with ERROR #8351.

Regardless of the `start` position, `$LOCATE` returns the position of the first match as a count from the beginning of the string.

**end**

An output variable that `$LOCATE` assigns an integer value if the locate operation found a match. The assigned value is the location of the first character position after the matched substring, counting from the beginning of the string. If the match occurs at the end of the string, this character position can be one more that the total string length. If a match was not found, the `end` value is left unchanged. If `end` has not been previously set, the variable remains undefined.

The `end` variable cannot be a reference to an object property.

By using the same variable for `start` and `end`, you can invoke `$LOCATE` repeatedly to find all of the matches in the string. This is shown in the following example, which locates the positions of the vowels in the alphabet:

```objectscript
SET alphabet="abcdefghijklmnopqrstuvwxyz"
SET pos=1
SET val=""
FOR i=1:1:5 {
   WRITE $LOCATE(alphabet,"[aeiou]",pos,pos,val)
   WRITE " is the position of the ",i,"th vowel: ",val,!}
```

**val**

An output variable that `$LOCATE` assigns a string value if the locate operation found a match. This string value is the matching substring. If a match was not found, the `val` value is left unchanged. If `val` has not been previously set, the variable remains undefined.

The `val` variable cannot be a reference to an object property.

**Examples**

The following example returns 4, because the `regexp` literal “de” matches at the 4th character of the string:

```objectscript
WRITE $LOCATE("abcdef","de")
```

The following example returns 8, because `regexp` specifies a lowercase letter string of three characters, which is first found here as the substring “fga” starting a position 8:

```objectscript
WRITE $LOCATE("ABC-de-fgabc123ABC","[:lower:]{3}")
```
The following example returns 5, because the specified \textit{regexp} format of spaces (\textbackslash{s}) and non-space characters (\textbackslash{S}) is found beginning at the 5th character of the string. This example omits the \textit{start} parameter; it sets the \textit{end} variable to 11, which is the character after the matched substring.

\begin{verbatim}
WRITE $LOCATE("AAAAA\# 5 456789","\S\S\s\S\S\s\S",,end)
\end{verbatim}

The following example returns 9, because \textit{regexp} specifies a letter string of three characters, and the \textit{start} parameter states it must begin at or after position 6:

\begin{verbatim}
SET end="",val="
WRITE $LOCATE("abc-def-ghi-jkl","[[:alpha:]]{3}",6,end,val),!
WRITE "the position after the matched string is: ",end,!
WRITE "the matched value is: ",val
\end{verbatim}

The \textit{end} parameter is set to 12, and the \textit{val} parameter is set to “ghi”.

The following example shows that a numeric is resolved to canonical form before \textit{regexp} is matched to the resulting string. The \textit{end} parameter is set to 5, one character beyond the end of the 4-character string “1.23”:

\begin{verbatim}
WRITE $LOCATE(123E-2,\".\d*",1,end,val),!
WRITE "end is: ",end,!
WRITE "value is: ",val,!
\end{verbatim}

The following example sets the \textit{start} parameter to a value greater than the length of \textit{string}+1. This results in an error, as shown:

\begin{verbatim}
TRY {
  SET str="abcdef"
  SET strlen=$LENGTH(str)
  WRITE "start=string length, match=",$LOCATE(str,"\p{L}",strlen),!
  WRITE "start=string length+1, match=",$LOCATE(str,"\p{L}",strlen+1),!
  WRITE "start=string length+2, match=",$LOCATE(str,"\p{L}",strlen+2),!
}
CATCH exp {
  WRITE !,"CATCH block exception handler: ",!
  IF 1=exp.%IsA("%Exception.SystemException") {
    WRITE "System exception",!
    WRITE "Name: ",exp.Name,"O","HTML"),!
    WRITE "Location: ",exp.Location,!
    WRITE "Code: ",exp.Code,!
    WRITE "Regex.Matcher status:"
    SET err=$class(%Regex.Matcher).LastStatus()
    DO $SYSTEM.Status.DisplayError(err) 
  } ELSE {WRITE "Unexpected exception type",! }
  RETURN 
}
\end{verbatim}

\textbf{See Also}

- \texttt{SCHAR} function
- \texttt{$MATCH$} function
- \textit{Regular Expressions} in \textit{Using Caché ObjectScript}
- \textit{Pattern Matching} in \textit{Using Caché ObjectScript}
$MATCH

Matches a regular expression to a string.

$MATCH(string,regexp)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string to be matched.</td>
</tr>
<tr>
<td>regexp</td>
<td>A regular expression to match against <code>string</code>. A regular expression consists of one or more meta-characters, and may also contain literal characters.</td>
</tr>
</tbody>
</table>

**Description**

$MATCH is a boolean function that returns 1 if `string` and `regexp` match, and 0 if `string` and `regexp` do not match. By default, matching is case-sensitive.

ObjectScript support for regular expressions consists of the $LOCATE and $MATCH functions:

- $MATCH matches a regular expression to the full `string` and returns a boolean indicating whether a match occurred.
- $LOCATE matches a regular expression to successive substrings of `string` and returns the location (and, optionally, the value) of the first match.

The `Match()` method of the %Regex.Matcher class provides the same functionality. The %Regex.Matcher class provides additional functionality for using regular expressions.

Other ObjectScript matching operations use Caché pattern match operators.

**Parameters**

**string**

An expression that evaluates to a string. The expression can be specified as the name of a variable, a numeric value, a string literal, or any valid ObjectScript expression. A string can contain control characters.

**regexp**

A regular expression used to match against `string`. A regular expression is an expression that evaluates to a string consisting of some combination of meta-characters and literals. Meta-characters specify character types and match patterns. Literals specify one or more matching single characters, ranges of characters, or substrings. An extensive regular expression syntax is supported. For details, refer to the “Regular Expressions” chapter of Using Caché ObjectScript.

**Examples**

The following example matches a string with a regular expression that specifies that the first character must be an uppercase letter (\p{LU}), followed by at least one additional character (+ quantifier), and that this second character, and all subsequent characters, must be word characters (letters, numbers, or the underscore character) (\w):

```objectscript
SET strng(1)="Assembly_17"
SET strng(2)="Part5"
SET strng(3)="SheetMetalScrew"
SET n=1
WHILE $DATA(strng(n)) {
    IF $MATCH(strng(n),"\p{LU}\w+") {
        WRITE strng(n)," : successful match",! } 
    ELSE { WRITE strng(n)," : invalid string",! }
    SET n=n+1 }
```
The following example returns 1, because the hexadecimal regular expression (hex 41) matches the letter “A”:

```objectscript
WRITE $MATCH("A","\x41")
```

The following example returns 1, because the specified string matches the format of spaces (\s) and non-space characters (\S) specified in the regular expression:

```objectscript
WRITE $MATCH("A $ 4","\S\S\s\S\S\S")
```

The following example returns 1, because the specified date matches the format of digits and literals specified in the regular expression:

```objectscript
SET today=$ZDATE($HOROLOG)
WRITE $MATCH(today,"^\d\d/\d\d/\d\d\d\d")
```

Note that this format requires that the day and month each be specified as two digits, so a leading zero is required for values smaller than 10.

The following example returns 1, because each letter in the string is within the corresponding letter range in the regular expression:

```objectscript
WRITE $MATCH("HAL","[G-I][A-C][K-Z]")
```

The following example specifies an invalid `regexp` parameter. This results in an error, as shown:

```objectscript
TRY {
SET str="abcdef"
WRITE "match=",$MATCH(str,"\p{}"),!
}
CATCH exp {
WRITE !!,"CATCH block exception handler:",!
IF l=exp.%IsA("%Exception.SystemException") {
WRITE "Name: ",$ZCVT(exp.Name,"O","HTML"),!
WRITE "Location: ",exp.Location,!
WRITE "Code: ",exp.Code,!
WRITE "%Regex.Matcher status:"
SET err=##class(%Regex.Matcher).LastStatus()
DO $SYSTEM.Status.DisplayError(err) }
ELSE {WRITE "Unexpected exception type",! }
RETURN
}
```

**See Also**

- `$CHAR` function
- `$LOCATE` function
- `$ZSTRIP` function
- *Regular Expressions* in *Using Caché ObjectScript*
- *Pattern Matching* in *Using Caché ObjectScript*
$METHOD

Supports calls to an instance method.

$METHOD(instance, methodname, arg1, arg2, arg3, ... )

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance</td>
<td>An expression that evaluates to an object reference. The value of the expression must be that of an in-memory instance of a class.</td>
</tr>
<tr>
<td>methodname</td>
<td>An expression that evaluates to a string. The value of the string must exactly match the name of an existing method in the instance of the class given as the first argument.</td>
</tr>
<tr>
<td>arg1, arg2, arg3, ...</td>
<td>A series of expressions to be substituted for the arguments to the designated method. The values of the expressions can be of any type. It is the responsibility of the implementer to make sure that the supplied expressions both match in type and have values with the bounds that the method expects. (If the specified method expects no arguments then nothing beyond <code>classname</code> and <code>methodname</code> need be used in the function invocation. If the method requires arguments, the rules that govern what must be supplied are those of the target method.)</td>
</tr>
</tbody>
</table>

**Description**

$METHOD executes a named instance method for a specified instance of a designated class.

This function permits an ObjectScript program to call an arbitrary method in an existing instance of some class. Since the first argument must be a reference to an object, it is computed at execution time. The method name may be computed at runtime or supplied as a string literal. If the method takes arguments, they are supplied from the list of arguments that follow the method name. A maximum of 255 argument values may be passed to the method. If the method requires arguments, the rules that govern what must be supplied are those of the target method. To invoke a class method rather than an instance method, use the $CLASSMETHOD function.

The invocation of $METHOD as a function or a procedure determines the invocation of the target method. You can invoke $METHOD using the DO command, discarding the return value.

When used within one method of a class instance to refer to another method of that instance, the $METHOD may omit `instance`. The comma that would normally follow `Instance` is still required, however.

If there is an attempt to invoke a method that is nonexistent or that is declared to be a class method, this results in a `<METHOD DOES NOT EXIST>` error.

**Example**

The following example shows $METHOD used as a function:

```
SET ListOfStuff = ##class(%Library.ListOfDataTypes).%New()
FOR i = "First", "Second", "Third", "Fourth"
    DO ListOfStuff.Insert((i _ "-Element"))
[-------------------]
DO ListOfStuff.Insert((i _ "-Element"))
SET methodname = "Count"
SET elements = $METHOD(ListOfStuff,methodname)
WRITE "Elements: ",elements,!
SET i = $RANDOM(elements) + 1
WRITE "Element ", i , " = ", $METHOD(ListOfStuff,"GetAt", i), !
```
See Also

- `$CLASSMETHOD` function
- `$CLASSNAME` function
- `$PARAMETER` function
- `$PROPERTY` function
- `$THIS` special variable
$NAME

Returns the name value of a variable or a portion of a subscript reference.

$NAME(variable, integer)
$NA(variable, integer)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>The variable whose name value is to be returned. It can specify a local or global variable, which can be either subscripted or unsubscripted. It does not need to be a defined variable. However, it may not be a defined private variable. If variable is a subscripted global, you can specify a naked global reference.</td>
</tr>
<tr>
<td>integer</td>
<td>Optional — A numeric value that specifies which portion (level) of a subscript reference to return. It can be a positive integer, the name of a variable, or an expression. When used, variable must be a subscripted reference.</td>
</tr>
</tbody>
</table>

**Description**

$NAME returns a formatted form of the variable name reference supplied as variable. It does not check whether this variable is defined or has a data value. The value $NAME returns depends on the parameters used.

- $NAME(variable) returns the name value of the specified variable in canonical form; that is, as a fully expanded reference.
- $NAME(variable, integer) returns a portion of a subscript reference. Specifically, integer controls the number of subscripts returned by the function.

Execution of this function does not affect the naked indicator.

**Parameters**

**variable**

variable can be a local variable, a process-private global variable, or a global variable. It can be unsubscripted or subscripted. If variable is a global it can use extended global reference. $NAME returns the extended global reference as specified, without checking whether the specified namespace exists or whether the user has access privileges for the namespace. It does not capitalize the namespace name. If variable is a naked global reference, $NAME returns the full global reference. If variable is a private variable, a compile error occurs.

It can be a multidimensional object property; it cannot be a non-multidimensional object property. Attempting to use $NAME on a non-multidimensional object property results in an <OBJECT DISPATCH> error.

$NAME cannot return a special variable, even those that can be modified using SET. Attempting to return a special variable results in a <SYNTAX> error.

**integer**

The integer parameter is used when variable is a subscripted reference. If the value of integer is 0, $NAME returns only the name of the variable. If the value of integer is less than the number of subscripts in variable, $NAME returns the number of subscripts indicated by the value of integer. If integer is greater than the number of subscripts in variable, $NAME returns the full subscripted reference.

If variable is an unsubscripted variable, the value of integer is ignored; $NAME returns the variable name. If integer is the null string (""") or a nonnumeric string, $NAME returns the variable name with no subscripts.
The value of integer receives standard integer parsing. For example, leading zeros and the plus sign are ignored. Fractional digits are truncated and ignored. A negative integer value results in a <FUNCTION> error.

**Examples**

In this example, the integer parameter specifies the level to return. If the specified number of subscripts in integer matches or exceeds the number of subscript levels (in this case, 3), then $NAME returns all defined levels, behaving as if you specified the one-parameter form. If you specify an integer level of zero (0), the null string (""), or any nonnumeric string (such as "A"), $NAME returns the name of the array (in this case "^client")

```objectscript
SET ^client(4)="Vermont"
SET ^client(4,1)="Brattleboro"
SET ^client(4,1,1)="Yankee Ingenuity"
WRITE !,$NAME(^client(4,1,1),1) ; returns 1 level
WRITE !,$NAME(^client(4,1,1),2) ; returns 2 levels
WRITE !,$NAME(^client(4,1,1),3) ; returns 3 levels
WRITE !,$NAME(^client(4,1,1),4) ; returns all (3) levels
WRITE !,$NAME(^client(4,1,1),0) ; returns array name
WRITE !,$NAME(^client(4,1,1),"") ; returns array name
WRITE !,$NAME(^client(4,1,1)) ; returns all (3) levels
```

In the following example, $NAME is used with a naked reference in a loop to output the values for all elements in the current (user-supplied) array level.

```objectscript
READ !,"Array element: ",ary
SET x=ary ; dummy operation to set current array and level
SET y=$ORDER(^"\") ; null string to find beginning of level
FOR i=0:0 {
    WRITE !,$NAME(^y)
    SET y=$ORDER(^y)
    QUIT:y=""
}
```

The first SET command performs a dummy assignment to establish the user-supplied array and level as the basis for the subsequent naked references. The $ORDER function is used with a naked reference to return the number of the first subscript (whether negative or positive) at the current level.

The WRITE command in the FOR loop uses $NAME with a naked global reference and argument indirection to output the value of the current element. The SET command uses $ORDER with a naked global reference to return the subscript of the next existing element that contains data. Finally, the postconditional QUIT checks the value returned by $ORDER to detect the end of the current level and terminate the loop processing.

You can use the returned $NAME string value for name or subscript indirection or pass it as a parameter to a routine or user-defined function. For more information, refer to Indirection in Using Caché ObjectScript. Consider the routine ^DESCEND that lists descendant nodes of the specified node.

```objectscript
DESCEND (ROOT) ;List descendant nodes
NEW REF
SET REF=ROOT
IF ROOT'("{" { FOR { SET REF=$QUERY(@REF)
QUIT:REF=""
WRITE REF, !}
}
ELSE {
SET $EXTRACT(ROOT,$LENGTH(ROOT))=","
FOR { SET REF=$QUERY(@REF)
QUIT:REF'[ROOT
WRITE REF, !]
}
```

The following example demonstrates how you can use $NAME to pass a parameter to the ^DESCEND routine defined in the previous example.

```objectscript
FOR var1="ONE","TWO","THREE" { DO ^DESCEND($NAME(X(var1))) }
```
$NAME

Notes

Uses for $NAME

You typically use $NAME to return the name values of array elements for use with the $DATA and $ORDER functions. If a reference to an array element contains expressions, $NAME evaluates the expressions before returning the canonical form of the name. For example:

```objectscript
SET x=1+2
SET y=$NAME(^client(4,1,x))
WRITE y
```

$NAME evaluates the variable x and returns the value ^client(4,1,3).

Naked Global References

$NAME also accepts a naked global reference and returns the name value in its canonical form (that is, a full (non-naked) reference). A naked reference is specified without the array name and designates the most recently executed global reference. In the following example, the first SET command establishes the global reference and the second SET command uses the $NAME function with a naked global reference.

```objectscript
SET ^client(5,1,2)="Savings/27564/3270.00"
SET y=$NAME(^3)
WRITE y
```

In this case, $NAME returns the value ^client(5,1,3). The supplied subscript value (3) replaces the existing subscript value (2), at the current level.

For more details, see Naked Global Reference in Using Caché Globals.

Extended Global References

You can control whether $NAME returns name values in extended global reference form on a per-process basis using the RefInKind() method of the %SYSTEM.Process class. The system-wide default behavior can be established by setting the RefInKind property of the Config.Miscellaneous class.

With extended reference mode in effect, the following example returns the defined namespace and name

```objectscript
^["PAYROLL"]MyRoutine (as shown in the first example), and not just ^MyRoutine (as shown in the second example):
```

```objectscript
DO #class(%SYSTEM.Process).RefInKind(0)
WRITE $NAME(["PAYROLL"]MyRoutine)
```

```objectscript
DO #class(%SYSTEM.Process).RefInKind(1)
WRITE $NAME(["PAYROLL"]MyRoutine)
```

For a description of extended global reference syntax, see the Global Structure chapter in Using Caché Globals.

See Also

- $DATA function
- $ORDER function
- $GET function
### $NCONVERT

Converts a number to a binary value encoded in a string of 8-bit characters.

#### $NCONVERT(n, format, endian)

#### $NC(n, format, endian)

#### Parameters

- **n**: Any number, which can be specified as a value, a variable, or an expression. Additional limitations on valid values are imposed by the *format* selected.
- **format**: One of the following format codes, specified as a quoted string: S1, S2, S4, S8, U1, U2, U4, F4, or F8.
- **endian**: *Optional* — A boolean value, where 0 = little-endian and 1 = big-endian. The default is 0.

#### Description

$NCONVERT uses the specified *format* to convert the number *n* to an encoded string of 8-bit characters. The values of these characters are in the range $CHAR(0) through $CHAR(255).

The following are the supported *format* codes:

- **S1**: Signed integer encoded into a string of one 8-bit byte. The value must be in the range -128 through 127, inclusive.
- **S2**: Signed integer encoded into a string of two 8-bit bytes. The value must be in the range -32768 through 32767, inclusive.
- **S4**: Signed integer encoded into a string of four 8-bit bytes. The value must be in the range -2147483648 through 2147483647, inclusive.
- **S8**: Signed integer encoded into a string of eight 8-bit bytes. The value must be in the range -9223372036854775808 through 9223372036854775807, inclusive.
- **U1**: Unsigned integer encoded into a string of one 8-bit byte. The maximum value is 255.
- **U2**: Unsigned integer encoded into a string of two 8-bit bytes. The maximum value is 65535.
- **U4**: Unsigned integer encoded into a string of four 8-bit bytes. The maximum value is 4294967295.
- **F4**: IEEE floating point number encoded into a string of four 8-bit bytes.
- **F8**: IEEE floating point number encoded into a string of eight 8-bit bytes.

Values beyond the range of *format* limits result in a <VALUE OUT OF RANGE> error. Specifying a negative number for an *Unsigned* *format* results in a <VALUE OUT OF RANGE> error. If *n* is a non-numeric value (contains any non-numeric characters) Caché performs conversion of a string to a numeric value. A string beginning with a non-numeric character is converted to 0.

Caché rounds a fractional number to an integer value for all formats except F4 and F8.

You can use the `IsBigEndian()` class method to determine which bit ordering is used on your operating system platform: 1=big-endian bit order; 0=little-endian bit order.

```ojs
WRITE $SYSTEM.Version.IsBigEndian()
```
$SCONVERT provides the inverse of the $NCONVERT operation.

**Examples**

The following example converts a series of unsigned numbers to two-byte encoded values:

```objectscript
FOR x=250:1:260 {
    ZZDUMP $NCONVERT(x,"U2")
}
QUIT
```

The following example performs the same operation in big-endian order:

```objectscript
FOR x=250:1:260 {
    ZZDUMP $NCONVERT(x,"U2",1)
}
QUIT
```

**See Also**

* $SCONVERT function
$NEXT

Returns the value of the next subscript in a subscripted variable.

$NEXT(variable)  
$N(variable)

Parameter

| variable | The array subscript to be used to determine the next existing subscript value. |

Description

$NEXT returns the value of the next subscript in the subscripted variable. It can be either a local or global subscripted array element. The value of the rightmost subscript must be a number whose value is -1 or greater. If you are using a naked global reference, the naked indicator must be defined.

$NEXT is an obsolete function that has been superseded by the $ORDER function. It is documented here only for completeness.

Note: Do not use $NEXT in any new code. Use $ORDER in place of $NEXT to retrieve subscripts of an array.

Notes

$NEXT and $ORDER

$NEXT is similar to $ORDER. Both return the subscripts of the next sibling in collating order to the specified node. However, $NEXT and $ORDER have different start and failure codes, as follows:

<table>
<thead>
<tr>
<th></th>
<th>$NEXT</th>
<th>$ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Starting point</td>
<td>-1</td>
<td>Null string</td>
</tr>
<tr>
<td>Failure code</td>
<td>-1</td>
<td>Null String</td>
</tr>
</tbody>
</table>

See Also

$ORDER function
$NORMALIZE

Validates and returns a numeric value; rounds to a specified precision.

$NORMALIZE(num, scale)

Parameters

<table>
<thead>
<tr>
<th>num</th>
<th>The numeric value to be validated. It can be a numeric or string value, a variable name, or any valid ObjectScript expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale</td>
<td>The number of significant digits to round num to as the returned value. This number can be larger or smaller than the actual number of fractional digits in num. Permitted values are 0 (round to integer), –1 (truncate to integer), and positive integers (round to specified number of fractional digits). There is no maximum scale value. However, the functional maximum cannot exceed the numeric precision. For standard Caché fractional numbers the functional scale maximum is 18 (minus the number of integer digits – 1).</td>
</tr>
</tbody>
</table>

Description

The $NORMALIZE function validates num and returns the normalized form of num. It performs rounding (or truncation) of fractional digits using the scale parameter. You can use the scale parameter to round a real number to a specified number of fractional digits, to round a real number to an integer, or to truncate a real number to an integer.

After rounding, $NORMALIZE removes trailing zeros from the return value. For this reason, the number of fractional digits returned may be less than the number specified in scale, as shown in the following example:

```objectscript
WRITE $NORMALIZE($ZPI,11),!
WRITE $NORMALIZE($ZPI,12),!    /* trailing zero removed */
WRITE $NORMALIZE($ZPI,13),!
WRITE $NORMALIZE($ZPI,14)
```

Parameters

num

The number to be validated may be an integer, a real number, or a scientific notation number (with the letter “E” or “e”). It may be a string, expression, or variable that resolves to a number. It may be signed or unsigned, and may contain leading or trailing zeros. $NORMALIZE validates character-by-character. It stops validation and returns the validated portion of the string if:

- num contains any characters other than the digits 0–9, + or – signs, a decimal point (.), and a letter “E” or “e”.
- num contains more than one decimal point, or letter “E” or “e”.
- If a + or – sign is found after a numeric in num it is considered a trailing sign, and no further numerics are parsed.
- The letter “E” or “e” is not followed by an integer.

The scale parameter value causes the returned value to be a rounded or truncated version of the num value. The actual value of the num variable is not changed by $NORMALIZE processing.

scale

The mandatory scale parameter is used to specify how many fractional digits to round to. Depending on the value specified, scale can have no effect on fractional digits, round to a specified number of fractional digits, round to an integer, or truncate to an integer.
A nonnegative \( scale \) value causes \( num \) to be rounded to that number of fractional digits. When rounding, a value of 5 or greater is always rounded up. To avoid rounding a number, make \( scale \) larger than the number of possible fractional digits in \( num \). A \( scale \) value of 0 causes \( num \) to be rounded to an integer value (3.9 = 4). A \( scale \) value of \(-1\) causes \( num \) to be truncated to an integer value (3.9 = 3). A \( scale \) value which is nonnumeric or the null string is equivalent to a \( scale \) value of 0.

Specify an integer value for \( scale \); decimal digits in the \( scale \) value are ignored. You can specify a \( scale \) value larger than the number of decimal digits specified in \( num \). You can specify a \( scale \) value of \(-1\); all other negative \( scale \) values result in a <FUNCTION> error.

**Examples**

In the following example, each invocation of \$NORMALIZE \ returns the normalized version of \( num \) with the specified rounding (or integer truncation):

```
WRITE !,$NORMALIZE(0,0)        ; All integers OK
WRITE !,$NORMALIZE('',0)       ; Null string is parsed as 0
WRITE !,$NORMALIZE(4.567,2)    ; Real numbers OK
WRITE !,$NORMALIZE('+4.567',2) ; Numeric strings OK
WRITE !,$NORMALIZE(-+.0,99)    ; Leading/trailing signs OK
WRITE !,$NORMALIZE('+004.500',1); Leading/trailing 0's OK
WRITE !,$NORMALIZE(4E2,-1)     ; Scientific notation OK
```

In the following example, each invocation of \$NORMALIZE \ returns a numeric subset of \( num \):

```
WRITE !,$NORMALIZE("4,567",0)  ; NumericGroupSeparators (commas) are not recognized
                                  ; here validation halts at the comma, and 4 is returned.
WRITE !,$NORMALIZE("4A",0)     ; Invalid (nonnumeric) character halts validation
                                  ; here 4 is returned.
```

The following example shows the use of the \( scale \) parameter to round (or truncate) the return value:

```
WRITE !,$NORMALIZE(4.55,2)      ; When scale is equal to the fractional digits of num,
                                 ; all digits of num are returned without rounding.
WRITE !,$NORMALIZE(3.85,1)      ; num is rounded to 1 fractional digit,
                                 ; (with values of 5 or greater rounded up)
                                 ; here 3.9 is returned.
WRITE !,$NORMALIZE(4.01,17)     ; scale can be larger than number of fractional digits,
                                 ; and no rounding is performed; here 4.01 is returned.
WRITE !,$NORMALIZE(3.85,0)      ; When scale=0, num is rounded to an integer value.
                                 ; here 4 is returned.
WRITE !,$NORMALIZE(3.85,-1)     ; When scale=-1, num is truncated to an integer value.
                                 ; here 3 is returned.
```

**Notes**

**$DOUBLE Numbers**

$DOUBLE IEEE floating point numbers are encoded using binary notation. Most decimal fractions cannot be exactly represented in this binary notation. When a $DOUBLE value is input to $NORMALIZE with a \( scale \) value, the return value frequently contains more fractional digits than specified in \( scale \) because the fractional decimal result is not representable in binary, so the return value must be rounded to the nearest representable $DOUBLE value, as shown in the following example:

```
SET x=1234.1234
SET y=$DOUBLE(1234.1234)
WRITE "Double: ",$NORMALIZE(y,2),!
WRITE "Dec/Dub: ",$NORMALIZE($DECIMAL(y),2)
```

384 Caché ObjectScript Reference
If you are normalizing a $DOUBLE value for decimal formatting, you should convert the $DOUBLE value to decimal representation before normalizing the result, as shown in the above example.

$NORMALIZE handles $DOUBLE ("INF") and $DOUBLE ("NAN") values, and returns INF and NAN.

DecimalSeparator Value Ignored

$NORMALIZE is intended to operate on numbers. A num string is interpreted as a number. Caché converts a number to a canonical form before supplying it to $NORMALIZE. This conversion to a canonical number does not use the DecimalSeparator property value for the current locale.

For example, if you specify for num the string "00123.4500", $NORMALIZE treats this as the canonical number 123.45, regardless of the current DecimalSeparator value. If you specify the string "00123,4500", $NORMALIZE treats this as the canonical number 123 (truncating at the first non-numeric character), regardless of the current DecimalSeparator value.

If you want a function that takes a string as input, use $INUMBER. If you want a function that produces a string result use $FNUMBER.

$NORMALIZE, and $NUMBER Compared

The $NORMALIZE, and $NUMBER functions both validate numbers and return a validated version of the specified number.

These two functions offer different validation criteria. Select the one that best meets your needs.

• Both functions parse signed and unsigned integers (including –0), scientific notation numbers (with “E” or “e”), and real numbers. However, $NUMBER can be set (using the “I” format) to reject numbers with a fractional part (including scientific notation with a negative base-10 exponent). Both functions parse both numbers (123.45) and numeric strings (“123.45”).

• Both functions strip out leading and trailing zeroes. The decimal character is stripped out unless followed by a nonzero value.

• Numeric strings containing a NumericGroupSeparator: $NUMBER parses NumericGroupSeparator characters (American format: comma (,); European format: period (.) or apostrophe (')) and the decimal character (American format: period (.) or European format: comma (,)) based on its format parameter (or the default for the current locale). It accepts and strips out any number of NumericGroupSeparator characters. For example, in American format it validates “123,4,56.99” as the number 123456.99. $NORMALIZE does not recognize NumericGroupSeparator characters. It validates character-by-character until it encounters a nonnumeric character; for example, it validates “123,456,99” as the number 123.

• Multiple leading signs (+ and –) are interpreted by both functions for numbers. Only $NORMALIZE accepts multiple leading signs in a quoted numeric string.

• Trailing + and – signs: Both functions reject trailing signs in numbers. In a quoted numeric string $NUMBER parses one (and only one) trailing sign. $NORMALIZE parses multiple trailing signs.

• Parentheses: $NUMBER parses parentheses surrounding an unsigned number in a quoted string as indicating a negative number. $NORMALIZE treats parentheses as nonnumeric characters.

• Numeric strings containing multiple decimal characters: $NORMALIZE validates character-by-character until it encounters the second decimal character. For example, in American format it validates “123.4,56” as the number 123.4. $NUMBER rejects any string containing more than one decimal character as an invalid number.

Numeric strings containing other nonnumeric characters: $NORMALIZE validates character-by-character until it encounters an alphabetic character. It validates “123A456” as the number 123. $NUMBER validates or rejects the entire string; it reject “123A456” as an invalid number.

• The null string: $NORMALIZE parses the null string as zero (0). $NUMBER rejects the null string.
The $NUMBER function provides optional min/max range checking. This is also available using the $ISVALIDNUM function.

$NORMALIZE, $NUMBER, and $ISVALIDNUM all provide rounding of numbers to a specified number of fractional digits. $NORMALIZE can round fractional digits, and round or truncate a real number to return an integer. For example, $NORMALIZE can round 488.65 to 488.7 or 489, or truncate it to 488. $NUMBER can round real numbers or integers. For example, $NUMBER can round 488.65 to 488.7, 489, 490 or 500.

**See Also**

- $DOUBLE function
- $FNUMBER function
- $INUMBER function
- $ISVALIDNUM function
- $NUMBER function
- More information on locales in the section on “System Classes for National Language Support” in *Caché Specialized System Tools and Utilities*. 
$NOW

Returns the local date and time with fractional seconds for the current process.

$NOW(tzmins)

Parameter

tzmins  
Optional — A positive or negative integer value that specifies the desired time zone offset from the Greenwich meridian, in minutes. A value of 0 corresponds to the Greenwich meridian. Positive integers correspond to time zones west of Greenwich; negative integers correspond to time zones east of Greenwich. For example, a value of 300 corresponds to United States Eastern Standard Time, 5 hours (300 minutes) west of Greenwich. The range of permitted values is -1440 through 1440; values beyond this range result in an <ILLEGAL VALUE> error.

If you omit tzmins, the $NOW function returns the local date and time based on the $ZTIMEZONE special variable value. The range of $ZTIMEZONE values that the $NOW function supports is -1440 through 1440; values beyond this range result in an <ILLEGAL VALUE> error.

Description

$NOW can return the following:

• The current local date and time with fractional seconds for the current process.
• The local date and time for a specified time zone, with fractional seconds, for the current process.

The $NOW function returns a character string that consists of two numeric values, separated by a comma. The first number is an integer that represents the current local date. The second is a fractional number that represents the current local time. These values are counters, not user-readable dates and times.

$NOW returns the date and time in Caché storage ($HOROLOG) format, with the additional feature of fractional seconds. $NOW returns the current local date and time in the following format: dddd,ssss.sfffff

The first integer (ddddd) is the number of days since December 31, 1840, where day 1 is January 1, 1841. The maximum value for this date integer is 2980013, which corresponds to December 31, 9999.

The second number (sssss.sfffff) is the number of seconds (and fractional seconds) since midnight of the current day. Caché increments the sssss field from 0 to 86399 seconds. When it reaches 86399 at midnight, Caché resets the sssss field to 0 and increments the date field by 1. Note that within the first second after midnight, seconds are represented as 0.fffff (for example, 0.123); this number is not in ObjectScript canonical form (for example, .123), which affects the string sorting order of these values. You can prepend a plus sign (+) to force conversion of a number to canonical form before performing a sort operation. The number of fffff fractional digits is the maximum precision supported by the current operating system. For more on Windows fractional seconds, see below.

The $NOW function can be invoked with or without a parameter value. The parentheses are mandatory.
$NOW with no parameter value returns the current local date and time for the current process. It determines the local time zone from the value set in the $ZTIMEZONE special variable. Setting $ZTIMEZONE changes the time portion of $NOW, and this change of time can also change the date portion of $NOW.

CAUTION: The $NOW local time value may not correspond to local clock time. $NOW determines local time using the $ZTIMEZONE value. $ZTIMEZONE is continuous throughout the year; it does not adjust for Daylight Saving Time (DST) or other local time variants.

Offset from UTC time is calculated using a count of time zones from the Greenwich meridian. It is not a comparison of your local time with local Greenwich time. The term Greenwich Mean Time (GMT) may be confusing; local time at Greenwich is the same as UTC during the winter; during the summer it differs from UTC by one hour. This is because a local time variant, known as British Summer Time, is applied. $NOW ignores all local time variants.

Also, because $NOW resynchronizes its time value with the system clock, comparisons of time values between $NOW and other Caché time functions and special variables may show slight variations. This variation is limited to 0.05 seconds; however, within this range of variation, comparisons may yield misleading results. For example, WRITE $NOW(),!, $HOROLOG may yield results such as the following:

61438,38794.002085
61438,38793

This anomaly is caused both by the 0.05 second resynchronization variation and by $HOROLOG truncation of fractional seconds.

$NOW with a parameter value returns the time and date that correspond to the time zone specified in tzmins. The value of $ZTIMEZONE is ignored.

**Time Functions Compared**

The various ways to return the current date and time are compared, as follows:

- **$NOW** returns the local date and time for the current process. $NOW returns the date and time in Caché storage format. It includes fractional seconds; the number of fractional digits is the maximum precision supported by the current operating system.
  - $NOW() determines the local time zone from the value of the $ZTIMEZONE special variable. The local time is not adjusted for local time variants, such as Daylight Saving Time. It therefore may not correspond to local clock time.
  - $NOW(tzmins) returns the time and date that correspond to the specified tzmins time zone parameter. The value of $ZTIMEZONE is ignored.

- **$HOROLOG** contains the local, variant-adjusted date and time in Caché storage format. The local time zone is determined from the current value of the $ZTIMEZONE special variable, and then adjusted for local time variants, such as Daylight Saving Time. It returns whole seconds only; fractions of a second are truncated.

- **$ZTIMESTAMP** contains the UTC (Coordinated Universal Time) date and time, with fractional seconds, in Caché storage format. Fractional seconds are expressed in three digits of precision (on Windows systems), or six digits of precision (on UNIX® systems). For this reason, $NOW(0) may return UTC time with greater fractional second precision than $ZTIMESTAMP.

None of these ways to return the current date and time are an exact replacement for the obsolete $ZUTIL(188) function, which provided local date and time with both adjustment for local time variants and fractional seconds.

**Microseconds on Windows**

The precision of microseconds on Windows systems is generally less accurate than other platforms. This is due to the limitations of the QueryPerformanceCounter() Windows library routine. Caché determines microseconds by comparing the
result computed by QueryPerformanceCounter() with the system clock used for other date/time functions, which has at best 1 millisecond precision (and usually between 1 and 20 milliseconds of accuracy depending on the hardware running Windows.) When the $NOW microsecond precision clock differs from the millisecond precision clock by more than 50 milliseconds, Caché adjusts the microsecond precision clock forward or backwards to agree with the more accurate (but less precise) millisecond precision clock.

Because of this, on Windows, the $NOW function (and legacy SZUTIL(188) function) is suitable for measuring microsecond precision time intervals only over a very short duration, lasting no longer than a small number of seconds. On systems with multiple CPU cores, the Caché Windows process for each core will make microsecond adjustments that are local to the individual process. This means that $NOW values returned on different Caché CPU processes can disagree with each other. This discrepancy among CPU processes is more pronounced when Windows is running on a virtual machine that is hosted on a multi-core CPU chip.

**Separating Date and Time**

To get just the date portion or just the time portion of $NOW, you can use the $PIECE function, specifying the comma as the delimiter character:

```objectscript
SET dateval=$PIECE($NOW(),",",1)
SET timeval=$PIECE($NOW(),",",2)
WRITE !,"Date only: ",dateval
WRITE !,"Time only: ",timeval
```

**Setting the Date**

The value returned by $NOW and $ZTIMESTAMP cannot be set using the FixedDate() method of the %SYSTEM.Process class.

The value contained in $HOROLOG can be set to a user-specified date for the current process using the FixedDate() method of the %SYSTEM.Process class.

**Examples**

The following example shows two ways to return the current local date and time:

```objectscript
WRITE $ZDATETIME($NOW(),1,1,3)," $NOW() date & time",!
WRITE $ZDATETIME($HOROLOG,1,1,3)," $HOROLOG date & time"
```

Note that $HOROLOG adjusts for local time variants, such as Daylight Saving Time. $NOW does not adjust for local time variants.

The following example uses $ZDATE to convert the date field in $NOW to a date format.

```objectscript
WRITE $ZDATE($PIECE($NOW(),",",1))
```

returns a value formatted like this: 04/29/2009

The following example converts the time portion of $NOW to a time in the form of hours:minutes:seconds.ffff on a 12-hour (a.m. or p.m.) clock.

```objectscript
CLOCKTIME
NEW
SET Time=$PIECE($NOW(),",",2)
SET Sec=Time#60
SET Totmin=Time\60
SET Min=Totmin#60
SET Milhour=Totmin\60
IF Milhour=12 { SET Hour=12,Meridian=" pm" }
ELSEIF Milhour>12 { SET Hour=Milhour-12,Meridian=" pm" }
ELSE { SET Hour=Milhour,Meridian=" am" }
WRITE !,"Hour,"#,Min,"",Sec,Meridian
QUIT
```
See Also

- $ZDATE function
- $ZDATEH function
- $ZDATETIME function
- $ZDATETIMEH function
- $ZTIME function
- $ZTIMEH function
- $HOROLOG special variable
- $ZTIMESTAMP special variable
- $ZTIMEZONE special variable
$NUMBER

Validates and returns a numeric value; optionally provides rounding and range checking.

$NUMBER(num, format, min, max)
$NUM(num, format, min, max)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>The numeric value to be validated and then converted to Caché canonical form. It can be a numeric or string value, a variable name, or any valid ObjectScript expression.</td>
</tr>
<tr>
<td>format</td>
<td>Optional — Specifies which processing options to apply to num. These processing options dictate primarily how to recognize and handle numbers containing decimal points.</td>
</tr>
<tr>
<td>min</td>
<td>Optional — The minimum acceptable numeric value.</td>
</tr>
<tr>
<td>max</td>
<td>Optional — The maximum acceptable numeric value.</td>
</tr>
</tbody>
</table>

**Description**

The $NUMBER function converts and validates the num numeric value using the specified format. It accepts numbers supplied with a variety of punctuation formats and returns numbers in Caché canonical form. You can use format to test whether a number is an integer. If min or max are specified, the number must fall within that range of values.

$NUMBER can be used for American format numbers, European format numbers, and Russian/Czech format numbers.

Using $NUMBER on the $DOUBLE values INF, –INF, or NAN always returns the empty string.

**Parameters**

**format**

The possible format codes are as follows. These format codes may be specified in any order. A nonnumeric format must be specified as a quoted string. Any or all of the following format codes may be omitted. If format is invalid, $NUMBER generates a <SYNTAX> error.

- Decimal character: either “.” or “,” indicating whether to use the American (“.”) or European (“,”) convention for validating the decimal point. You can specify either of these characters, or no decimal character. If you omit the decimal character, the number takes the DecimalSeparator of the current locale.

- Rounding factor: an integer indicating how many digits to round to. This integer can be preceded by an optional + or - sign. If the rounding factor is positive (or unsigned) the number is rounded to the specified number of fractional digits. If the rounding factor is 0, the number is rounded to an integer. If the rounding factor is a negative integer, the number is rounded the indicated number of places to the left of the decimal separator. For example, a rounding factor of -2 rounds 234.45 to 200. The number “5” is always rounded up; thus a rounding factor of 1 rounds 123.45 to 123.5.

- Integer indicator: the letter “I” (uppercase or lowercase) which specifies that the number must resolve to an integer. For example, –07.00 resolves to an integer, but –07.01 does not. If the number does not resolve to an integer, $NUMBER returns the null string. In the following example, only the first three $NUMBER functions returns an integer. The other three return the null string:

```ObjectScript
WRITE $NUMBER(-07.00,"I")," non-canonical integer numeric",
WRITE $NUMBER(+"-07.00","I")," string forced as integer numeric",
WRITE $NUMBER("-75.00","I")," canonical integer string numeric",
WRITE $NUMBER("-07.00","I")," non-canonical integer string numeric",
WRITE $NUMBER(-07.01,"I")," fractional numeric",
WRITE $NUMBER("-07.01","I")," fractional string numeric"
```
**min and max**

You can specify a minimum allowed value, a maximum allowed value, neither, or both. If specified, the `num` value (after rounding) must be greater than or equal to the `min` value, and less than or equal to the `max` value. A null string as a `min` or `max` value is equal to zero. If a value does not meet these criteria, `$NUMBER` returns the null string.

Thus in the following examples, the first is valid because `num` (4.0) equals `max` (4). The second is valid because `num` (4.003) still equals `max` (4) within the format range (two fractional digits). However, the third is not valid because `$NUMBER` rounds `num` up to a value (4.01) greater than `max` within the format range. It returns a null string.

```
WRITE !,$NUMBER(4.0,2,0,4)
WRITE !,$NUMBER(4.003,2,0,4)
WRITE !,$NUMBER(4.006,2,0,4)
```

You can omit parameters, retaining the commas as place holders. The first line of the following example sets a `max` value, but no `format` or `min` value. The second line sets no `format` value, but sets a `min` value of the null string, which is equivalent to zero. Thus the first line returns −7, and the second line fails the `min` criteria and returns the null string.

```
SET max=10
WRITE !,$NUMBER(-7,,,,max)
WRITE !,$NUMBER(-7,,"",max)
```

You cannot specify trailing commas. The following results in a `<SYNTAX>` error:

```
WRITE $NUMBER(mynum,,min,)
```

**Notes**

**Order of Operations**

$NUMBER performs the following series of conversions and validations. If the number fails any validation step, $NUMBER returns a null string (""). If the number passes all validation steps, $NUMBER returns the resulting converted Caché canonical form number.

1. $NUMBER uses the decimal character format to determine which character is the group separator and strips out all group separator characters (regardless of their location in the number). It uses the following rule: If the decimal character specified in `format` is a period (.), then the group separator is a comma (,) or blank space. If the decimal character specified in `format` is a comma (,), then the group separator is a period (.) or blank space. If no decimal character is specified in `format`, the group separator is the NumericGroupSeparator property of the current locale. (The Russian (rusw), Ukrainian (ukrw), and Czech (csyw) locales use a blank space as the numeric group separator.)

2. $NUMBER validates that the number is well-formed. A well-formed number can contain any of the following:
   - Numbers
   - An optional decimal indicator character, as defined above (one or none, but not more than one).
   - An optional plus (+) or minus (-) sign character (leading or trailing, but not more than one).
   - Optional parentheses enclosing the number to indicate a negative value (debit). The number within the parentheses cannot have a sign character.
   - An optional base-10 exponent, indicated by an “E” (uppercase or lowercase) followed by an integer. If “E” is specified, an exponent integer must be present. The exponent integer may be preceded by a sign character.

3. If the integer indicator is present in `format`, $NUMBER checks for integers. An integer cannot contain a decimal indicator character. Numeric strings (“123.45”) and numbers (123.45) are parsed differently. Numeric strings fail this integer test even if there are no digits following the decimal indicator character, or if expansion of scientific notation or rounding would eliminate the fractional digits. Numbers pass these validation tests. If a number fails the integer indicator check, $NUMBER returns the null string ("").
4. **$NUMBER** converts the number to a Caché canonical form number. It expands scientific notation, replaces enclosing parentheses with a negative sign character, strips off leading and trailing zeros, and deletes a decimal indicator character if it is not followed by any nonzero digits.

5. **$NUMBER** uses the rounding factor (if present) to round the number the specified number of digits. It then strips off any leading or trailing zeros and the decimal indicator character if it is not followed by any digits.

6. **$NUMBER** validates the number against the minimum value, if specified.

7. **$NUMBER** validates the number against the maximum value, if specified.

8. **$NUMBER** returns the resulting number.

---

**European and American Decimal Separators**

$NUMBER returns a number in canonical form, removing all numeric group separators and includes at most one decimal separator character. You can use the *format* values “,“ or “.” to identify the decimal separator used in *num*; by specifying the decimal separator, you are also implicitly specifying the numeric group separator.

To determine the DecimalSeparator character for your locale, invoke the following method:

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DecimalSeparator")
```

To determine the NumericGroupSeparator character and NumericGroupSize number for your locale, invoke the following methods:

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("NumericGroupSeparator"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("NumericGroupSize")
```

In following examples, a comma is specified as the decimal separator:

```
SET num="123,456"
WRITE !,$NUMBER(num,",")
// converts to the fractional number "123.456"
// (comma is identified as decimal separator)
SET num="123,45,6"
WRITE !,$NUMBER(num,",")
// returns the null string
// (invalid number, too many decimal separators)
SET num="123.456"
WRITE !,$NUMBER(num,",")
// converts to the integer "123456"
// removing group separator
// (if comma is decimal, then period is group separator)
SET num="123.4.56"
WRITE !,$NUMBER(num,",")
// converts to the integer "123456"
// removing group separators
// (number and placement of group separators ignored)
```

---

**Rounding and Precision**

When rounding numbers, be aware that IEEE floating point numbers and standard Caché fractional numbers differ in precision. **$DOUBLE** IEEE floating point numbers are encoded using binary notation. They have a precision of 53 binary bits, which corresponds to 15.95 decimal digits of precision. (Note that the binary representation does not correspond exactly to a decimal fraction.) Because most decimal fractions cannot be exactly represented in this binary notation, an IEEE floating point number may differ slightly from the corresponding standard Caché floating point number. Standard Caché fractional numbers have a precision of 18 decimal digits on all supported Caché system platforms. When an IEEE floating point number is displayed as a fractional number, the binary bits are often converted to a fractional number with far more than 18 decimal digits. This *does not* mean that IEEE floating point numbers are more precise than standard Caché fractional numbers.

**See Also**

- **$DOUBLE** function
ObjectScript Functions

- $FNUMBER function
- $INUMBER function
- $ISVALIDNUM function
- $NORMALIZE function
- More information on locales in the section on “System Classes for National Language Support” in *Caché Specialized System Tools and Utilities*. 
### $ORDER

Returns the next local variable or the subscript of a local or global variable.

$ORDER(variable,direction,target)
$O(variable,direction,target)

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>A subscripted local, process-private global, or global variable. If an array, the subscript is required. You cannot specify just the array name. You can specify an unsubscripted local variable using indirection (see example below). You cannot specify a simple object property reference as <code>variable</code>; you can specify a multidimensional property reference as <code>variable</code> with the syntax <code>obj.property</code>.</td>
</tr>
<tr>
<td>direction</td>
<td>Optional — The subscript order in which to traverse the target array. Values for subscripted variables can be: 1 = ascending subscript order (the default) or –1 = descending subscript order. For unsubscripted local variables, 1 (the default) is the only permitted value.</td>
</tr>
<tr>
<td>target</td>
<td>Optional — Returns the current data value of the next or previous node of <code>variable</code>. Whether it is the next or previous depends on the setting of <code>direction</code>. You must specify a <code>direction</code> value to specify a <code>target</code>. For unsubscripted local variables, <code>direction</code> must be set to 1. If <code>variable</code> is undefined, the <code>target</code> value remains unchanged. The <code>target</code> parameter cannot be used with structured system variables (SSVNs) such as <code>^$ROUTINE</code>.</td>
</tr>
</tbody>
</table>

#### Description

$ORDER is primarily used to loop through subscripted variables at a specified subscript level from a specified starting point. It returns sequential variables in collation sequence. It allows for gaps in the subscript sequence.

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(4,1,6)) returns the next third-level subscript. That would be 7, if the variable ^client(4,1,7) 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. (See example below).

- **$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); attempting to do so results in a <FUNCTION> error.

- **$ORDER(variable,direction,target)** returns the subscript for the variable, and sets `target` to its current data value. This can be either the next or the previous subscript for a subscripted variable, depending on the `direction` setting. For an unsubscripted local variable, `direction` must be set to 1 to return the current data value to `target`. The `target` parameter cannot be used with structured system variables (SSVNs) such as `^$ROUTINE`. The ZBREAK command cannot specify the `target` parameter as a watchpoint.
First Subscript Returned

You can either start a $ORDER loop with the variable following a specified variable, or with the first variable:

- Start at specified point: SET key=$ORDER(^mydata(99)) returns the next higher subscript after 99 — subscript 100, if it exists. Note that the node you specify in the argument (here subscript 99) need not exist. To return all positive subscripts you can specify SET key=$ORDER(^mydata(-1)). To return all negative subscripts you can specify SET key=$ORDER(^mydata(0),-1).

- Start at beginning: SET key=$ORDER(^mydata(“”)) returns the first subscripted variable in collation sequence. This technique is required if the level may contain negative as well as positive subscripts.

The following example returns both negative and positive first-level subscripts in ascending numeric sequence.

```
SET mydata(1)="a",mydata(-3)="C",mydata(5)="e",mydata(-5)="E"
// Get first subscript
SET key=$ORDER(mydata(“”))
WHILE (key’=“”) {
    WRITE key,"!
    // Get next subscript
    SET key = $ORDER(mydata(key))
}
```

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.

You can use $ORDER to return a limited subset of the defined local variables. You can use argumentless WRITE to display all defined local variables.

Examples

The examples shown here return local variables. $ORDER can also return subscripted global variables and subscripted process-private global variables.

The following example uses $ORDER in a WHILE loop to return all of the first-level subscripts in the mydata(n) global:

```
SET mydata(1)="a",mydata(3)="c",mydata(7)="g"
// Get first subscript
SET key=$ORDER(mydata(""))
WHILE (key’="") {
    WRITE key,"!
    // Get next subscript
    SET key = $ORDER(mydata(key))
}
```

The following example uses $ORDER in a WHILE loop to return all of the second-level subscripts in the mydata(1,n) global. Note that the first-level and third-level subscripts are ignored. This example returns both the subscript numbers and the corresponding variable values:

```
SET mydata(1,1)="a",mydata(1,3)="c",mydata(1,7)="lcase",mydata(1)="A",mydata(1,7)="g"
SET key=$ORDER(mydata(1,""),1,target)
WHILE (key’="") {
    WRITE key," = ",target,"!
    // Get next subscript
    SET key = $ORDER(mydata(1,key),1,target)
}
```

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:
The following example uses $ORDER in a WHILE loop to return unsubscripted local variables. Local variables are returned in collation sequence. This example returns both the local variable names and their values. Note that the @ indirection operator must be used when looping through unsubscripted local variables. This example starts with the next local variable in collation sequence after $b$ (in this case, $bminus$). It then loops through all defined local variables that follow it in collation sequence. To avoid listing the variables $foo$ and $target$, these variables are defined as process-private globals, rather than local variables:

```objectscript
SET a="great",b="good",bminus="pretty good",c="fair",d="poor",f="failure"
SET ^||foo="b"
SET ^||foo=$ORDER(@^||foo,1,^||target)
WHILE ^||foo '= "" {
  WRITE ^||foo," = ",^||target,!
  SET ^||foo=$ORDER(@^||foo,1,^||target)
}
```

### Notes

#### Uses for $ORDER

$ORDER is typically used with loop processing to traverse the nodes in an array that doesn’t use consecutive integer subscripts. $ORDER simply returns the subscript of the next existing node. For example:

```objectscript
SET struct=""
FOR {
  SET struct=$ORDER(^client(struct))
  QUIT:struct=""
  WRITE ","^client(struct)
}
```

The above routine writes the values for all the top-level nodes in the ^client global array.

$ORDER returns subscripts of existing nodes, but not all nodes contain a value. If you use $ORDER in a loop to feed a command (such as WRITE) that expects data, you must include a $DATA check for valueless nodes. For example, you could specify the WRITE command in the previous example with a postconditional test, as follows:

```objectscript
WRITE:($DATA(^client(struct))#10) !,^client(struct)
```

This test covers the case of both a valueless pointer node and a valueless terminal node. If your code becomes too cumbersome for a simple FOR loop, you can relegate part of it to a block-structured DO.

#### Global References

If a variable is global variable, it can include an extended global reference, specifying a global in a different namespace. If you specify a nonexistent namespace, Caché issues a <NAMESPACE> error. If you specify a namespace for which you do not have privileges, InterSystems IRIS issues a <PROTECT> error, followed by the global name and database path, such as the following: <PROTECT> ^myglobal(1),c:\intersystems\iris\mgr\. If the global variable has subscript mapping to a namespace for which the user does not have Read permission, the <PROTECT> error information shows the original global reference because you cannot see a subscript in a namespace for which you do not have privileges. However, the <PROTECT> error database path shows the protected database, not the original database.

If variable is a subscripted global variable, it can be a naked global reference. A naked global reference is specified without the array name and designates the most recently executed global reference. For example:
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 would return the value 1, indicating ^client(4,1), if that subscripted global is defined. If ^client(4,1) is not defined, it would return the value 2, indicating ^client(4,2) if that subscripted global is defined, and so forth.

All three parameters of $ORDER can take a naked global reference, or specify the global reference. However, if direction specifies an explicit global reference, subsequent naked global references do not use the direction global reference. They continue to use the prior established global reference, as shown in the following example:

```
SET ^client(4,3)="Jones"
SET ^client(4,5)="Smith"
SET ^dir(1)=-1
SET rtn=$ORDER(^client(4,5),-1)
WRITE $ZREFERENCE,!
    /* naked global ref is ^client(4,5) */
SET rtn=$ORDER(^client(4,5),^dir(1))
WRITE $ZREFERENCE,!
    /* NOTE: naked global ref is ^client(4,5) */
SET rtn=$ORDER(^client(4,5),^dir(1),^targ(1))
WRITE $ZREFERENCE
    /* NOTE: naked global ref is ^targ(1) */
WRITE ^targ(1),!
SET ^rtn(1)=$ORDER(^client(4,5),^dir(1),^targ(2))
WRITE $ZREFERENCE
    /* naked global ref is ^rtn(1) */
WRITE ^targ(2)
```

For more details, see Naked Global Reference in Using Caché Globals.

**$ORDER and $DOUBLE Subscripts**

$DOUBLE floating point numbers can be used as subscript identifiers. However, when used as a subscript identifier, the $DOUBLE number is converted to a string. When $ORDER returns a subscript of this type, it returns it as a numeric string, not a $DOUBLE floating point number.

**$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:

<table>
<thead>
<tr>
<th></th>
<th>$NEXT</th>
<th>$ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Starting point</td>
<td>-1</td>
<td>Null string</td>
</tr>
<tr>
<td>Failure code</td>
<td>-1</td>
<td>Null String</td>
</tr>
</tbody>
</table>

Because $ORDER starts and finishes on the null string, it correctly returns nodes having both negative and positive subscripts.

**See Also**

- $DATA function
- $GET function
- $NEXT function
- $QUERY function
- $ZPREVIOUS function
- Global Structure chapter in Using Caché Globals
### $PARAMETER

Returns the value of the specified class parameter.

$PARAMETER(class, parameter)

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>Optional — Either a class name or an object reference (OREF) to a class instance. If omitted, uses the object reference of the current class instance. When omitted, you must specify the placeholder comma.</td>
</tr>
<tr>
<td>parameter</td>
<td>The name of a parameter. An expression which evaluates to a string. The value of the string must match the name of an existing parameter of the class identified by class.</td>
</tr>
</tbody>
</table>

#### Description

$PARAMETER returns the value of a specified class parameter. $PARAMETER can look up this parameter in the current class context or in a specified class context. You can specify a class name as a quoted string, specify an OREF, or omit the class parameter and take as default the current instance (see $THIS). Specifying class is optional; specifying the comma separator is mandatory.

For example:

```
WRITE $PARAMETER("%Library.Boolean","XSDTYPE")
```

There are several ways to return the value of a parameter using object syntax, as shown in the following example:

```ObjectScript
ZNSPACE "Samples"
WRITE "ObjectScript function:",!
WRITE $PARAMETER("Sample.Person","EXTENTQUERYSPEC")
WRITE ","class parameter:",!
WRITE ##class(Sample.Person).#EXTENTQUERYSPEC
WRITE ","instance parameter:",!
SET myinst=##class(Sample.Person).%New()
WRITE myinst.%GetParameter("EXTENTQUERYSPEC")
WRITE ","instance parameter:",!
WRITE myinst.#EXTENTQUERYSPEC
```

#### Invalid Values

- $PARAMETER("","XMLTYPE") : attempting to invoke an invalid OREF (such as the empty string, an integer, or a fractional number) results in an <INVALID OREF> error.
- $PARAMETER("bogus","XMLTYPE") : attempting to invoke a nonexistent class results in a <CLASS DOES NOT EXIST> error, followed by the specified class name. If a package name is not specified, Caché assumes the default. For example, attempting to invoke the nonexistent class “bogus” results in the error <CLASS DOES NOT EXIST> *User.bogus.*
- $PARAMETER(,"XMLTYPE") : attempting to default to the current object instance when none has been established results in a <NO CURRENT OBJECT> error.
- $PARAMETER("%SYSTEM.Task","") : attempting to reference an invalid parameter name (for example, an empty string) or to reference a parameter by number generates an <ILLEGAL VALUE> error.
- $PARAMETER("%SYSTEM.Task","MakeCoffee") : attempting to reference a nonexistent parameter name returns the empty string ("").
Examples

The following example specifies class names and returns the class default values for the XMLTYPE and XSDTYPE parameters:

```
WRITE $PARAMETER("%SYSTEM.Task","XMLTYPE"),!
WRITE $PARAMETER("%Date","XSDTYPE")
```

The following example specifies an OREF and returns the value of the XMLTYPE parameter for this instance:

```
SET oref=##class(%SYSTEM.Task).%New()
WRITE $PARAMETER(oref,"XMLTYPE")
```

The following example returns a system parameter using $PARAMETER syntax and using class syntax:

```
WRITE $PARAMETER("%SYSTEM.SQL","%RandomSig"),!
WRITE ##class(%SYSTEM.SQL).#%RandomSig
```

See Also

- $CLASSMETHOD function
- $CLASSNAME function
- $METHOD function
- $PROPERTY function
- $THIS special variable
$PIECE

Returns or replaces a substring, using a delimiter.

$PIECE(string,delimiter,from,to)
$P(string,delimiter,from,to)
SET $PIECE(string,delimiter,from,to)=value
SET $P(string,delimiter,from,to)=value

Parameters

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The target string in which delimited substrings are identified. Specify string as an expression that evaluates to a quoted string or a numeric value. In SET $PIECE syntax, string must be a variable or a multi-dimensional property.</td>
</tr>
<tr>
<td>delimiter</td>
<td>A delimiter used to identify substrings within string. Specify delimiter as an expression that evaluates to a quoted string containing one or more characters.</td>
</tr>
<tr>
<td>from</td>
<td>Optional — An expression that evaluates to a code specifying the location of a substring, or the beginning of a range of substrings, within string. Substrings are separated by a delimiter, and counted from 1. Permitted values are n (a positive integer specifying the substring count from the beginning of string), * (specifying the last substring in string), and -*n (offset integer count of substrings counting backwards from end of string). SET $PIECE syntax also supports *+n (offset integer count of substrings to append beyond the end of string). Thus, the first delimited substring is 1, the second delimited substring is 2, the last delimited substring is *, and the next-to-last delimited substring is *-1. If from is omitted, it defaults to the first delimited substring.</td>
</tr>
<tr>
<td>to</td>
<td>Optional — An expression that evaluates to a code specifying the ending substring for a range of substrings within string. Must be used with from. Permitted values are n (a positive integer specifying the substring count from the beginning of string), * (specifying the last substring in string), and -*n (offset integer count of substrings from end of string). SET $PIECE syntax also supports *+n (offset integer for a range of substrings to append beyond the end of string). If to is prior to from in string, no operation is performed and no error is generated.</td>
</tr>
</tbody>
</table>

Description

$PIECE identifies substrings within string by the presence of a delimiter. If the delimiter does not occur in string, the entire string is treated as a single substring.

$PIECE can be used in two ways:

- To return a substring from string. This uses the $PIECE(string,delimiter,from,to) syntax.
- To replace a substring within string. It identifiers a substring and replaces it with another substring. The replacement substring may be the same length, longer, or shorter than the original substring. This uses the SET $PIECE(string,delimiter,from,to)=value syntax.

Note: $PIECE is a general-purpose function for handling a string containing delimited substrings. For handling a MultiValue dynamic array string containing the specific MultiValue delimiters, use the $MV function.

Returning a Substring

When returning a specified substring (piece) from string, the substring returned depends on the parameters used:
**$PIECE**

- **$PIECE(string, delimiter)** returns the first substring in string. If delimiter occurs in string, this is the substring that precedes the first occurrence of delimiter. If delimiter does not occur in string, the returned substring is string.

- **$PIECE(string, delimiter, from)** returns a substring whose location is specified by the from parameter. Substrings are delimited by delimiters and the beginning and end of string. The delimiter itself is not returned.

- **$PIECE(string, delimiter, from, to)** returns a range of substrings including the substring specified in from through the substring specified in to (inclusive). This four-argument form of $PIECE returns a substring that includes any intermediate occurrences of delimiter that occur between the from and to substrings. If to is greater than the number of substrings, the returned substring includes all substrings to the end of string.

**Parameters**

**string**

When $PIECE is used to return a substring, string can be a string literal enclosed in quotation marks, a canonical numeric, a variable, an object property, or any valid ObjectScript expression that evaluates to a string or a numeric. If you specify a null string ("") as the target string, $PIECE always returns the null string, regardless of the other parameter values.

A target string usually contains instances of a character (or character string) which are used as delimiters. This character or character string cannot also be used as a data value within string.

When $PIECE is used with SET on the left hand side of the equals sign to replace a substring, string can be a variable or a multidimensional property reference; it cannot be a non-multidimensional object property.

**delimiter**

The search string to be used to delimit substrings within string. It can be a string literal enclosed in quotation marks, a canonical numeric, a variable or any valid ObjectScript expression that evaluates to a string or a numeric.

Commonly, a delimiter is a designated character which is never used as data within string, but is set aside solely for use as a delimiter separating substrings. For example, if delimiter is "^", the string "Red^Orange^Yellow" contains three delimited substrings.

A delimiter can be a multi-character string, the individual characters of which can be used within string data. For example, if delimiter is "^#", the string "Red^Orange^#Yellow#Green#Blue" contains two delimited substrings: "Red^Orange" and "^Yellow#Green^#Blue".

Commonly, string does not begin or end with a delimiter. If string begins or ends with a delimiter, $PIECE treats this delimiter as demarcating a substring with a null string ("") value. For example, if delimiter is "^", the string "^Red^Orange^Yellow" contains five delimited substrings; substrings 1 and 5 have null string values.

If the specified delimiter is not in string, $PIECE returns the entire string. If the specified delimiter is the null string (""), $PIECE returns the null string.

**from**

The location of a substring within string. Use n (a positive integer) to count delimited substrings from the beginning of string. Use * to specify the last delimited substring in string. Use *-n to count delimited substrings by offset from the last delimited substring in string.

- 1 specifies the first substring of string (the substring that precedes the first occurrence of delimiter). If string does not contain the specified delimiter, a from value of 1 returns string. If from is omitted, it defaults to 1.

- 2 specifies the second substring of string (the substring that appears between the first and second occurrences of delimiter, or between the first occurrence of delimiter and the end of string).

- * specifies the last substring of string (the substring that follows the last occurrence of delimiter). If string does not contain the specified delimiter, a from value of * returns string.
- 1 specifies the next-to-last substring of string. *-n counts by offset from the last substring of string. *-0 is the last substring of string; * and *-0 are functionally identical.

- For SET SPIECE syntax only — *+n (an asterisk followed by a positive number) appends delimited substrings by offset beyond the end of string. Thus, *+1 appends a delimited substring beyond the end of string, *+2 appends a delimited substring two positions beyond the end of string, padding with delimiters.

- If from is the null string (""), zero, a negative number, or specifies a count or offset beyond the number of substrings in string, SPIECE returns a null string.

SPIECE converts a from numeric to canonical form (resolving leading plus and minus signs and removing leading zeros), then truncates it to an integer.

If the from parameter is used with the to parameter, it identifies the start of a range of substrings to be returned as a string, and should be less than the value of to.

**to**

The number of the substring within string that ends the range initiated by the from parameter. The returned string includes both the from and to substrings, as well as any intermediate substrings and the delimiters separating them. The to parameter must be used with from and should be greater than the value of from.

Use n (a positive integer) to count delimited substrings from the beginning of string. Use * to specify the last delimited substring in string. Use *-n to count delimited substrings by offset backwards from the last delimited substring in string.

For SET SPIECE syntax only — *+n (an asterisk followed by a positive number) specifies the end of a range of substrings to append beyond the end of string.

- If from is less than to, SPIECE returns a string consisting of all of the delimited substrings within this range, including the from and to substrings. This returned string contains the substrings and the delimiters within this range. If to is greater than the number of delimited substrings, the returned string contains all the string data (substrings and delimiters) beginning with the from substring and continuing to the end of string.

- If from is equal to to, SPIECE returns the from substring. This can occur if from and to are the same value, or are different values that reference the same substring.

- If from is greater than to, is zero (0), or is the null string (""), SPIECE returns a null string.

SPIECE converts a to numeric to canonical form (resolving leading plus and minus signs and removing leading zeros), then truncates it to an integer.

**Specifying *-n and *+n Parameter Values**

When using a variable to specify *-n or *+n, you must always specify the asterisk and a sign character in the parameter itself.

The following are valid specifications of *-n:

```
SET count=2
SET alph="a\"b\"c\"d"
WRITE SPIECE(alph,"\"",*-count)
```

```
SET count=-2
SET alph="a\"b\"c\"d"
WRITE SPIECE(alph,"\"",*+count)
```

The following is a valid specification of *+n:

```
SET count=2
SET alph="a\"b\"c\"d"
SET SPIECE(alph,"\"",*+count)="F"
WRITE alph
```
Whitespace is permitted within these parameter values.

**Examples: Returning a Delimited Substring**

In the following example, each $PIECE returns the specified substring as identified by the "," delimiter:

```ObjectScript
SET colorlist="Red, Green, Blue, Yellow, Orange, Black"
WRITE $PIECE(colorlist,",",1) ; returns "Red" (substring 1) by default
WRITE $PIECE(colorlist,",",3) ; returns "Blue" the third substring
WRITE $PIECE(colorlist,",",-1) ; returns "Orange" the next-to-last substring
```

In the following example, $PIECE returns the integer and fractional parts of a number:

```ObjectScript
SET int=$PIECE(123.999,".")
SET frac=$PIECE(123.999,".",*)
WRITE "integer=",int," fraction =.",frac
```

The following example returns "Blue,Yellow,Orange", the third through fifth substrings in colorlist, as delimited by ",":

```ObjectScript
SET colorlist="Red, Green, Blue, Yellow, Orange, Black"
SET extract=$PIECE(colorlist,",",3,5)
WRITE extract
```

The following WRITE statements all return the first substring “123”, showing that these formats are equivalent when from and to have a value of 1:

```ObjectScript
SET numlist="123#456#789"
WRITE ",1-arg=",$PIECE(numlist,"#")
WRITE ",3-arg=",$PIECE(numlist,"#",1)
WRITE ",4-arg=",$PIECE(numlist,"#",1,1)
```

In the following example, both $PIECE functions returns the entire string string, because there are no occurrences of delimiter in string:

```ObjectScript
SET colorlist="Red, Green, Blue, Yellow, Orange, Black"
SET extract1=$PIECE(colorlist,""")
SET extract2=$PIECE(colorlist,"",1,4)
WRITE ",extract1=",extract1,"",extract2
```

The following example $PIECE returns the second substring from an object property:

```ObjectScript
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SchemaPath="MyTests,Sample,Cinema"
WRITE "whole schema path: ",tStatement.%SchemaPath,!
WRITE "2nd piece of schema path: ",$PIECE(tStatement.%SchemaPath,"",2),!
```

The following two examples use more complex delimiters.

This example uses a delimiter string “#-#” to return three substrings of the string numlist. Here, the component characters of the delimiter string, “#” and “-”, can be used as data values; only the specified sequence of characters ( #- # ) is set aside:

```ObjectScript
SET numlist="1#2-3#-45#6-789"
WRITE ",$PIECE(numlist,"-#",1)
WRITE ",$PIECE(numlist,"#-",2)
WRITE ",$PIECE(numlist,"#-#",3)
```

The following example uses a non-ASCII delimiter character (in this case, the Unicode character for pi), specified using the $CHAR function, and inserted into string by using the concatenate operator (_):

```ObjectScript
IF $SYSTEM.Version.IsUnicode()  {
SET a = $CHAR(960)
SET colorlist="Red_a_Green_a_Blue"
SET extract1=$PIECE(colorlist,a)
SET extract2=$PIECE(colorlist,a,2)
SET extract3=$PIECE(colorlist,a,2,3)
WRITE extract1,!,extract2,!,extract3
}
ELSE {WRITE "This example requires a Unicode installation of Caché"}
```
Replacing a Substring Using SET $PIECE

When making assignments with the SET command, you can use $PIECE to the left, as well as to the right, of the equals sign. When used to the left of the equals sign, $PIECE designates a substring to be replaced by the assigned value.

When $PIECE is used with SET on the left hand side of the equals sign, string can be a valid variable name. If the variable does not exist, SET $PIECE defines it. The string parameter can also be a multidimensional property reference; it cannot be a non-multidimensional object property. Attempting to use SET $PIECE on a non-multidimensional object property results in an <OBJECT DISPATCH> error.

The use of $PIECE (and $LIST and $EXTRACT) in this context differs from other standard functions because it modifies an existing value, instead of just returning a value. You cannot use SET (a,b,c,...)=value syntax with $PIECE (or $LIST or $EXTRACT) on the left of the equals sign, if the function uses relative offset syntax: * representing the end of a string and *-n or +-n representing relative offset from the end of the string. You must instead use SET a=value,b=value,c=value,... syntax.

Examples: Replacing a Delimited Substring

The following example changes the value of colorlist to "Magenta,Green,Cyan,Yellow,Orange,Black":

```ObjectScript
SET colorlist="Red,Green,Blue,Yellow,Orange,Black"
WRITE colorlist,!
SET $PIECE(colorlist,"," ,1)="Magenta"
WRITE colorlist,!
SET $PIECE(colorlist,"," ,*-3)="Cyan"
WRITE colorlist,!
```

The replacement substring may, of course, be longer or shorter than the original, and may include delimiters:

```ObjectScript
SET colorlist="Red,Green,Blue,Yellow,Orange,Black"
WRITE colorlist,!
SET $PIECE(colorlist,"," ,3)="Turquoise,Aqua,Teal"
WRITE colorlist,!
```

If you specify a from and to argument, the included substrings are replaced by the specified value, in this case the 4th through 6th delimited substrings:

```ObjectScript
SET colorlist="Red,Blue,Yellow,Green,Orange,Black"
WRITE !,colorlist
SET $PIECE(colorlist,"," ,4,6)="Yellow+Blue,Yellow+Red"
WRITE !,colorlist
```

You can append one or more delimited substrings either by delimited substring count (using n), or by offset from the end of string (using *+n). SET $PIECE appends additional delimiters as needed to append the delimited substring(s) at the specified location. The following examples both change the value of colorlist to "Green^Blue^^Red", padding with an extra empty string delimited substring:

```ObjectScript
SET colorlist="Green^Blue"
SET $PIECE(colorlist,"^",4)="Red"
WRITE colorlist

SET colorlist="Green^Blue"
SET $PIECE(colorlist,"^",*+2)="Red"
WRITE colorlist
```

If delimiter doesn't appear in string, $PIECE treats string as a single piece and performs the same substitutions described above. If there is no from argument specified, the new value replaces the original string:

```ObjectScript
SET colorlist="Red,Green,Blue"
WRITE colorlist,!
SET $PIECE(colorlist,"^" )="Purple^Orange"
WRITE colorlist
```

If delimiter doesn't appear in string, and from is specified as an integer greater than 1, $PIECE appends from-1 delimiters and the supplied value to the end of string:
SET colorlist="Red,Green,Blue"
WRITE colorlist,!
SET $PIECE(colorlist,"\",3)="Purple"
WRITE colorlist

If *from* represents a position prior to the beginning of the string, Caché performs no operation:

SET colorlist="Red,Green,Blue"
WRITE colorlist,!
SET $PIECE(colorlist,"\",*-7)="Purple"
WRITE colorlist

If *from* represents a position prior to the beginning of the string and *to* is provided, Caché treats *from* as position 1:

SET colorlist="Red,Green,Blue"
WRITE colorlist,!
SET $PIECE(colorlist,"\",*-7,1)="Purple"
WRITE colorlist

**Initializing a String Variable**

The *string* variable does not need to be defined before being assigned a value. The following example initializes *newvar* to the character pattern ">>>>>>TOTAL":

SET $PIECE(newvar,">",7)="TOTAL"
WRITE newvar

See the "SET with $PIECE and $EXTRACT" section of the SET command documentation for more information.

**Delimiter is Null String**

If the delimiter is the null string, the new value replaces the original *string*, regardless of the values of the *from* and *to* arguments.

The following two examples both set *colorlist* to “Purple”:

SET colorlist="Red,Green,Blue"
WRITE !,colorlist
SET $PIECE(colorlist,"\",3)="Purple"
WRITE !,colorlist

SET colorlist="Red,Blue,Yellow,Green,Orange,Black"
WRITE !,colorlist
SET $PIECE(colorlist,"",3,5)="Purple"
WRITE !,colorlist

**$PIECE with Parameters over 32,768 Characters**

The following example creates a string of 5 periods and a null:

SET x=""
SET $PIECE(x,"\",6)=""
WRITE x

Now consider the following example that creates a string of 32767 periods and a null:

SET x=""
SET $PIECE(x,"\",32768)=""

Although technically within the maximum length of a string, this example generates a <MAXSTRING> error if long strings are not enabled. Long strings are enabled system-wide by default. If you wish to use $PIECE with a parameter greater than 32,767 characters, you can check or set the system-wide long strings setting using the EnableLongStrings property of the Config.Miscellaneous class.
Notes

Using $PIECE to Unpack Data Values

$PIECE is typically used to "unpack" data values that contain multiple fields delimited by a separator character. Typical delimiter characters include the slash (/), the comma (,), the space ( ), and the semicolon (;). The following sample values are good candidates for use with $PIECE:

"John Jones/29 River St./Boston MA, 02095"
"Mumps;Measles;Chicken Pox;Diptheria"
"45.23,52.76,89.05,48.27"

$PIECE and $LENGTH

The two-argument form of $LENGTH returns the number of substrings in a string, based on a delimiter. Use $LENGTH to determine the number of substrings in a string, and then use $PIECE to extract individual substrings, as shown in the following example:

```
SET sentence="The quick brown fox jumped over the lazy dog's back."
SET delim=" ">
SET countdown=$LENGTH(sentence,delim)
SET countup=1
FOR reps=countdown:-1:1 {
  SET extract=$PIECE(sentence,delim,countup)
  WRITE !,countup," ",extract
  SET countup=countup+1
}
WRITE !,"All done!"
```

Null Values

$PIECE does not distinguish between a delimited substring with a null string value, and a nonexistent substring. Both return a null string value. For example, the following examples both return the null string for a from value of 7:

```
SET colorlist="Red,Green,Blue,Yellow,Orange,Black"
SET extract1=$PIECE(colorlist,",",6)
WRITE "6=",extract1,"7=",extract2
```

```
SET colorlist="Red,Green,Blue,Yellow,Orange,Black,"
SET extract1=$PIECE(colorlist,",",6)
SET extract2=$PIECE(colorlist,",",7)
WRITE "6=",extract1,"7=",extract2
```

In the first case, there is no seventh substring; a null string is returned. In the second case there is a seventh substring, as indicated by the delimiter at the end of the string; the value of this seventh substring is the null string.

The following example shows null values within a string. It extracts substrings 1 and 3. These substrings exist, but both contain a null string. (Substring 1 is defined as the string preceding the first delimiter character):

```
SET colorlist="",Red,,Blue,"
SET extract1=$PIECE(colorlist,"",6)
SET extract2=$PIECE(colorlist,"",3)
WRITE !,"sub1=",extract1,"sub3=",extract3
```

The following examples also returns a null string, because the specified substrings do not exist:

```
SET colorlist="Red,Green,Blue,Yellow,Orange,Black"
SET extract=$PIECE(colorlist,"",0)
WRITE !,"Length=",$LENGTH(extract),!,"Value=",extract
```

```
SET colorlist="Red,Green,Blue,Yellow,Orange,Black"
SET extract=$PIECE(colorlist,"",8,20)
WRITE !,"Length=",$LENGTH(extract),!,"Value=",extract
```
Forcing Numeric Evaluation

Prefacing $PIECE (or any ObjectScript function) with a unary + sign forces numeric evaluation of the return value. It returns a numeric substring in canonical form. It returns a non-numeric substring as 0. It returns the leading numeric part of a mixed numeric substring. It returns 0 for a null string value or a nonexistent substring.

This forced numeric evaluation is shown in the following example:

```
SET str="snow white,7dwarves,007.00,99.90,-0,"
WRITE "Substrings:";!
FOR i=1:1:7 {WRITE i,"=",$PIECE(str,"",i)," "}
WRITE !,"Forced Numerics:";!
FOR i=1:1:7 {WRITE i,"=",$+PIECE(str,"",i)," "}
```

Nested $PIECE Operations

To perform complex extractions, you can nest $PIECE references within each other. The inner $PIECE returns a substring that is operated on by the outer $PIECE. Each $PIECE uses its own delimiter. For example, the following returns the state abbreviation “MA”:

```
SET patient="John Jones/29 River St./Boston MA 02095"
SET patientstateaddress=$PIECE($PIECE(patient,"/",3)," ",2)
WRITE patientstateaddress
```

The following is another example of nested $PIECE operations, using a hierarchy of delimiters. First, the inner $PIECE uses the caret (^) delimiter to find the second piece of nestlist: "A,B,C". Then the outer $PIECE uses the comma (,) delimiter to return the first and second pieces ("A,B") of the substring "A,B,C":

```
SET nestlist="1,2,3^A,B,C^@#!"
WRITE $PIECE($PIECE(nestlist,"^",2),",",1,2)
```

$PIECE Compared with $EXTRACT and $LIST

$PIECE determines a substring by counting user-defined delimiter characters within the string. $PIECE takes as input an ordinary character string containing multiple instances of a character (or string) intended for use as a delimiter.

$EXTRACT determines a substring by counting characters from the beginning of a string. $EXTRACT takes as input an ordinary character string.

$LIST determines an element from an encoded list by counting elements (not characters) from the beginning of the list. The $LIST functions specify substrings without using a designated delimiter. If setting aside a delimiter character or character sequence is not appropriate to the type of data (for example, bitstring data), you should use the $LISTBUILD and $LIST functions to store and retrieve substrings. You can convert a delimited string into a list using the $LISTFROMSTRING function. You can convert a list to a delimited string using the $LISTTOSTRING function.

The data storage strategies used by $PIECE and the $LIST functions are incompatible, and their use should not be combined. For example, attempted to use $PIECE on a list created using $LISTBUILD yields unpredictable results and should be avoided.

See Also

- SET command
- $EXTRACT function
- $LENGTH function
- $LIST function
- $LISTBUILD function
- $LISTFROMSTRING function
- $LISTTOSTRING function
• **SREVERSE** function
$PREFETCHOFF

Ends pre-fetching of globals.

$PREFETCHOFF(gref,gref2)

Parameters

<table>
<thead>
<tr>
<th>gref</th>
<th>Optional — A global reference.</th>
</tr>
</thead>
<tbody>
<tr>
<td>gref2</td>
<td>Optional — A global reference used to establish a range.</td>
</tr>
</tbody>
</table>

Description

$PREFETCHOFF turns off the pre-fetching of global nodes established by $PREFETCHON.

There are three forms of $PREFETCHOFF:

- $PREFETCHOFF() turns off all pre-fetching established for the current process.
- $PREFETCHOFF(gref) turns off pre-fetching of the gref node and all of its descendents. The gref value must correspond exactly to the $PREFETCHON value.
- $PREFETCHOFF(gref,gref2) turns off pre-fetching of the nodes in the range gref through gref2. gref and gref2 must be nodes of the same global. The gref and gref2 values must correspond exactly to the $PREFETCHON values. You cannot turn off part of a range of values.

Upon successful completion, $PREFETCHOFF() returns 0. It returns 0 even if there were no pre-fetches to turn off.

Upon successful completion, $PREFETCHOFF(gref) and $PREFETCHOFF(gref, gref2) return a string of six integers separated by commas. These six values are: number of blocks prefetched, number of I/Os performed, number of prefetch operations, milliseconds of prefetch disk time, background job: number of blocks prefetched, and background job: number of I/Os performed.

Upon failure, all forms of $PREFETCHOFF return -1. $PREFETCHOFF(gref) and $PREFETCHOFF(gref, gref2) return -1 if there is no corresponding $PREFETCHON that exactly matches the specified global or range of globals, or if the specified prefetch global or range of globals has already been turned off.

Parameters

gref

A global reference, either a global or a process-private global. The global does not need to be defined at the time that the pre-fetch is turned off.

You can specify this global using @ indirection. Refer to Indirection in Using Caché ObjectScript.

You cannot specify a structured system variable name (SSVN) for this parameter.

gref2

A global reference used to establish a range with gref. Therefore, gref2 must be a global node lower in the same global tree as gref.

You can specify this global using @ indirection. Refer to Indirection in Using Caché ObjectScript.

Examples

The following example establishes two pre-fetches, then turns them off individually:
SET ret=$PREFETCHON(^a)
IF ret=1 { WRITE !,"prefetch established" }
ELSE { WRITE !,"prefetch not established" }
SET ret2=$PREFETCHON(^b)
IF ret2=1 { WRITE !,"prefetch established" }
ELSE { WRITE !,"prefetch not established" }
SET retoff=$PREFETCHOFF(^a)
IF retoff=-1 { WRITE !,"prefetch turned off. Values:" ,retoff }
ELSE { WRITE !,"prefetch not turned off" }
SET retoff2=$PREFETCHOFF(^b)
IF retoff2=-1 { WRITE !,"prefetch turned off. Values:" ,retoff2 }
ELSE { WRITE !,"prefetch not turned off" }

The following example establishes two pre-fetches, then turns off all pre-fetches for the current process:

SET ret=$PREFETCHON(^a)
IF ret=1 { WRITE !,"prefetch established" }
ELSE { WRITE !,"prefetch not established" }
SET ret2=$PREFETCHON(^b)
IF ret2=1 { WRITE !,"prefetch established" }
ELSE { WRITE !,"prefetch not established" }
SET retoff=$PREFETCHOFF()
IF retoff=0 { WRITE !,"all prefetches turned off" }
ELSE { WRITE !,"prefetch not turned off" }

See Also

• $PREFETCHON function
• Globals in Using Caché ObjectScript
$PREFETCHON

Establishes pre-fetch for specified globals.

$PREFETCHON(gref,gref2)

Parameters

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gref</td>
<td>A global reference.</td>
</tr>
<tr>
<td>gref2</td>
<td>Optional — A global reference used to establish a range.</td>
</tr>
</tbody>
</table>

Description

$PREFETCHON improves performance by turning on pre-fetching for a global or a range of globals. $PREFETCHON returns 1 indicating successful completion (pre-fetching is enabled). $PREFETCHON returns 0 indicating the desired pre-fetch could not be established. A 0 might be returned if the specified range includes two different global names, or if there is some other problem that prevents pre-fetching. A returned 0 is not an error; it does not interrupt program execution, and processing of global references in the specified range is not impaired. It simply means that these global operations do not have the performance boost of pre-fetching.

Note: Pre-fetching of globals is not supported on a remote database.

$PREFETCHOFF turns off pre-fetching.

There are two forms of $PREFETCHON:

- $PREFETCHON(gref) pre-fetches the gref node and all of its descendents. For example, $PREFETCHON(^abc(4)) pre-fetches all of the descendents of ^abc(4), such as ^abc(4,1), ^abc(4,2,2), and so forth. It does not pre-fetch ^abc(5).
- $PREFETCHON(gref,gref2) pre-fetches the nodes in the range gref through gref2. This does not include the descendents of gref2. gref and gref2 must be nodes of the same global. For example, $PREFETCHON(^abc(4),^abc(7,5)) pre-fetches all of the global nodes in the range of ^abc(4) through ^abc(7,5), including ^abc(4,2,2), ^abc(5), and ^abc(7,1,2). However, it does not pre-fetch ^abc(7,5,1).

Pre-fetching is not restricted to read access; it also works well when a large number of SET operations are being performed.

Pre-fetching and Performance

When you invoke $PREFETCHON, one or more pre-fetch background processes (daemons) are started as required. These pre-fetch daemons are shared systemwide by all processes. Because each pre-fetch daemon processes only one pre-fetch request at a time, it is usually advantageous to have several pre-fetch daemons running on your system. However, large numbers of concurrent pre-fetch daemons can have a performance impact on interactive system access.

Pre-fetching can improve performance when running an application that reads a large number of disk blocks containing nodes from the same global tree. Pre-fetching is most efficient when:

- Data is accessed in generally ascending order, meaning that data blocks of the global tree are generally accessed in left-to-right order. However, there is no requirement for adhering strictly to ascending order. Pre-fetching works best when either the data blocks of the global tree are generally accessed in left-to-right order, or when at least one data block within the range is likely to be accessed prior to most of its neighbors to the right in the logical tree.
- Most of the data blocks in the specified range are accessed. However, the initial pre-fetch does not fetch as many blocks as subsequent fetches, in case the user decides to cancel the operation after accessing only a small portion of the data.
- Less than 100 pre-fetches are active at any given time.
Parameters

gref
A global reference, either a global or a process-private global. The global does not need to be defined at the time that the pre-fetch is established.

You can specify this global using @ indirection. Refer to Indirection in Using Caché ObjectScript.

You cannot specify a structured system variable name (SSVN) for this parameter.

gref2
A global reference used to establish a range with gref. Therefore, gref2 must be a global node lower in the same global tree as gref.

You can specify this global using @ indirection. Refer to Indirection in Using Caché ObjectScript.

Examples

The following example establishes pre-fetch for the global ^a.

```
SET ^a="myglobal"
SET x=^a
SET ret=$PREFETCHON(^a)
IF ret=1 { WRITE !,"prefetch established" }
ELSE { WRITE !,"prefetch not established" }
SET ret=$PREFETCHOFF()
```

The following example establishes pre-fetch for the range of globals ^a(1) through ^a(50).

```
SET ret=$PREFETCHON(^a(1),^a(50))
IF ret=1 { WRITE !,"prefetch established" }
ELSE { WRITE !,"prefetch not established" }
```

See Also

- $PREFETCHOFF function
- Globals in Using Caché ObjectScript
$PROPERTY

Supports reference to a particular property of an instance.

$PROPERTY(instance, propertyname, index1, index2, index3... )

Parameters

- **instance**
  - *Optional* — An expression that evaluates to an object instance reference (OREF). The value of the expression must be that of an in-memory instance of the desired class. If omitted, defaults to the current object.

- **propertyname**
  - An expression that evaluates to a string. The value of the string must match the name of an existing property defined in the class identified by *instance*.

- **index1, index2, index3, ...**
  - *Optional* — If *propertyname* is a multidimensional value, then this series of expressions is treated as indices into the array represented by the property. (If the specified property is not multidimensional, the presence of extra arguments causes an error at runtime.)

Description

$PROPERTY gets or sets the value of a property in an instance of the designated class. This function permits an ObjectScript program to select the value of an arbitrary property in an existing instance of some class. Since the first argument must be an instance of a class, it is computed at execution time. The property name may be computed at runtime or supplied as a string literal. The contents of the string must match exactly the name of a property declared in the class. Property names are case-sensitive.

If the property is declared to be multidimensional, then the arguments after the property name are treated as indices into a multidimensional array. A maximum of 255 argument values may be used for the index.

$PROPERTY may also appear on the left side of an assignment. When $PROPERTY appears to the left of an assignment operator, it provides the location to which a value is assigned. When it appears to the right, it is the value being used in the calculation.

If *instance* is not a valid in-memory OREF, an <INVALID OREF> error occurs. If *propertyname* is not a valid property, a <PROPERTY DOES NOT EXIST> error occurs. If you specify an index1 and *propertyname* is not multidimensional, an <OBJECT DISPATCH> error occurs.

Remarks

The $PROPERTY function calls the Get() or Set() methods of the property passed to it. It is functionally the same as using the “Instance.PropertyName” syntax, where “Instance” and “PropertyName” are equivalent to the arguments as listed in the function’s signature. Because of this, $PROPERTY should not be called within a property’s Get() or Set() method, if one exists. For more information on Get() and Set() methods, see the chapter “Using and Overriding Property Methods” in Using Caché Objects.

When used within a method to refer to a property of the current instance, $PROPERTY may omit instance. The comma that would normally follow *instance* is still required, however.

An attempt to get a multidimensional value from a property which is not declared to be multidimensional results in a <FUNCTION> error; likewise for attempting to set a multidimensional value into a property that is not multidimensional.

Examples

The following example returns the current NLS Language property value:
The following example shows $PROPERTY used as a function:

```objectscript
SET TestName = "%Library.File"
SET ClassDef = ##class(%Library.ClassDefinition).%OpenId(TestName)
FOR i = "Name", "Super", "Persistent", "Final"
    { WRITE i, " ": $PROPERTY(ClassDef, i), !
    }
```

The following example shows $PROPERTY used on both sides of an assignment operator:

```objectscript
SET TestFile = ##class(%Library.File).%New("AFile")
WRITE "Initial file name: ", $PROPERTY(TestFile, "Name")!
SET $PROPERTY(TestFile, "Name") = $PROPERTY(TestFile, "Name") _ "Renamed"
WRITE "File name afterward: ", $PROPERTY(TestFile, "Name")!
```

The following example returns a property value from the current object, in this case the SQL Shell. $PROPERTY is specified with its first parameter omitted:

```objectscript
USER>DO $SYSTEM.SQL.Shell()
SQL Command Line Shell
---------------------------------------
The command prefix is currently set to: <<nothing>>.
Enter q to quit, ? for help.
USER>>set path="a,b,c"
path="a,b,c"
USER>>COS WRITE "The schema search path is ", $PROPERTY("Path")
The schema search path is "a,b,c"
USER>>
```

See Also

- $CLASSMETHOD function
- $CLASSNAME function
- $METHOD function
- $PARAMETER function
- $THIS special variable
$QLENGTH

Returns the number of subscript levels in a variable.

$QLENGTH(var)

Parameter

| var       | A string, or expression that evaluates to a string, that contains the name of a variable. The variable name can specify no subscripts or one or more subscripts. |

Description

$QLENGTH returns the number of subscript levels in var. $QLENGTH simply counts the number of subscript levels specified in var. The var variable does not have to be defined for $QLENGTH to return the number of subscript levels.

Parameter

var

A quoted string, or expression that evaluates to a string, which specifies a variable. It can be a local variable, a process-private variable, or a global variable.

If the string is a global reference, var can specify an extended global reference by including a namespace name. Because var is a quoted string, the quotes around a namespace reference must be doubled to be parsed correctly as literal quotation marks. For example, "^|""SAMPLES"|myglobal(1,4,6)". The same applies to the quotes in a process-private global with "^" syntax. For example, "^|""^"|ppgname(3,6)". $QLENGTH does not check whether the specified namespace exists or whether the user has access privileges for the namespace.

A var must specify a variable name in canonical form (a fully expanded reference). To use $QLENGTH with a naked global reference, or with indirection, you can use the $NAME function to return the corresponding fully expanded reference.

Examples

The following example show the results of $QLENGTH when used with subscripted and unsubscripted globals. The first $QLENGTH takes a global with no subscripts and returns 0. The second $QLENGTH takes a global with two subscript levels and returns 2. Note that quotes found in the variable name are doubled because var is specified as a quoted string.

```
WRITE 1,$QLENGTH("^|""USER"|test")
; returns 0
SET name="^|""USER"|test(1,""customer"")"
WRITE 1,$QLENGTH(name)
; returns 2
```

The following example returns the $QLENGTH value for a process-private global with three subscript levels. The $ZREFERENCE special variable contains the name of the most recently referenced variable.

```
SET ^||myppg("food","fruit",1)="apples"
WRITE 1,$QLENGTH($ZREFERENCE) ; returns 3
```

The following example returns the $QLENGTH value for a process-private global specified as a naked global reference. The $NAME function is used to expand the naked global reference to canonical form:

```
SET "grocerylist("food","fruit",1)="apples"
SET (^2)="bananas"
WRITE 1,$QLENGTH($NAME(^2))) ; returns 3
```
See Also

- $QUERY function
- $QSUBSCRIPT function
- $NAME function
- $ZREFERENCE special variable
- Global Structure in Using Caché Globals
$QSUBSCRIPT

Returns a variable name or a subscript name.

$QSUBSCRIPT(namevalue,intexpr)  
$QS(namevalue,intexpr)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>namevalue</code></td>
<td>A string, or an expression that evaluates to a string, which is the name of a local variable, process-private global, or global variable, with or without subscripts.</td>
</tr>
<tr>
<td><code>intexpr</code></td>
<td>An integer code that specifies which name to return: variable name, subscript name, or namespace name.</td>
</tr>
</tbody>
</table>

**Description**

$QSUBSCRIPT returns the variable name, or the name of a specified subscript of `namevalue`, depending on the value of `intexpr`. If `namevalue` is a global variable, you can also return the namespace name, if it was explicitly specified. $QSUBSCRIPT does not return a default namespace name.

**Parameters**

*namevalue*

A quoted string, or expression that evaluates to a string, which is a local or global reference. It can have the form: Name($s_1, s_2, ..., s_n$).

If the string is a global reference, it can contain a namespace reference. Because `namevalue` is a quoted string, the quotes around a namespace reference must be doubled to be parsed correctly as literal quotation marks.

A `namevalue` must reference a variable name in canonical form (a fully expanded reference). To use $QSUBSCRIPT with a naked global reference, or with indirection, you can use the $NAME function to return the corresponding fully expanded reference.

*intexpr*

An integer expression code that indicates which value to return. Assume that the `namevalue` parameter has the form NAME($s_1, s_2, ..., s_n$), where $n$ is the ordinal number of the last subscript. The `intexpr` parameter can have any of the following values:

<table>
<thead>
<tr>
<th>Code</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; -1</td>
<td>Generates a &lt;FUNCTION&gt; error; these numbers are reserved for future extensions.</td>
</tr>
<tr>
<td>-1</td>
<td>Returns the namespace name if a global variable <code>namevalue</code> includes one; otherwise, returns the null string (&quot;&quot;).</td>
</tr>
<tr>
<td>0</td>
<td>Returns the variable name. Returns ^NAME for a global variable, and ^</td>
</tr>
<tr>
<td>&lt;=n</td>
<td>Returns the subscript name for the level of subscription specified by the integer $n$, with 1 being the first subscript level and $n$ being the highest defined subscript level.</td>
</tr>
<tr>
<td>&gt;n</td>
<td>An integer &gt; $n$ returns the null string (&quot;&quot;), where $n$ is the highest defined subscript level.</td>
</tr>
</tbody>
</table>
Examples

The following example returns $QSUBSCRIPT values when namevalue is a subscripted global with one subscript level and a specified namespace:

```objectscript
SET global="^"|"account"|\$test(""customer"")"
WRITE !,$QSUBSCRIPT(global,-1) ; account
WRITE !,$QSUBSCRIPT(global,0) ; ^%test
WRITE !,$QSUBSCRIPT(global,1) ; customer
WRITE !,$QSUBSCRIPT(global,2) ; null string
```

The following example returns $QSUBSCRIPT values when namevalue is a process-private global with two subscript levels. The $ZREFERENCE special variable contains the name of the most recently referenced variable.

```objectscript
SET ^||myppg(1,3)="apples"
WRITE !,$QSUBSCRIPT($ZREFERENCE,-1) ; null string
WRITE !,$QSUBSCRIPT($ZREFERENCE,0) ; ^||myppg
WRITE !,$QSUBSCRIPT($ZREFERENCE,1) ; 1
WRITE !,$QSUBSCRIPT($ZREFERENCE,2) ; 3
```

The following example returns the $QSUBSCRIPT value for a global variable specified as a naked global reference. The $NAME function is used to expand the naked global reference to canonical form:

```objectscript
SET ^grocerylist("food","fruit",1)="apples"
SET ^ cosmestic""|\$test(""customer"")"
SET ^(2)="bananas"
WRITE !,$QSUBSCRIPT($NAME(^(2)),2) ; returns "fruit"
```

See Also

- $QUERY function
- $QLENGTH function
- $NAME function
- $ZREFERENCE special variable
- Global Structure in Using Caché Globals
$QUERY

Performs a physical scan of a local or global array.

$QUERY(reference,direction,target)
$Q(reference,direction,target)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference</td>
<td>A reference that evaluates to the name (and optionally subscripts) of a public local or global variable.</td>
</tr>
<tr>
<td>direction</td>
<td>Optional — The direction (forwards or backwards) to traverse the array.</td>
</tr>
<tr>
<td>target</td>
<td>Optional — Returns the current data value of the local or global variable specified in reference.</td>
</tr>
</tbody>
</table>

Description

$QUERY performs a physical scan of a public local or global array; it returns the full reference, name and subscripts, of the defined node next in sequence to the specified array node. If no such node exists, $QUERY returns the null string.

Parameters

reference

This parameter must evaluate to a public variable or a global. $QUERY cannot scan a private variable.

The returned global reference can be at the same level, a lower level, or a higher level as the level specified in the reference parameter. If you specify reference without specifying subscripts, $QUERY returns the first defined node in the array.

direction

If no direction is specified, the default direction is forward. If you wish to specify a direction, a parameter value of 1 will traverse the array forward, a value of -1 will traverse the array backward.

target

If you wish to specify a target, you must specify a direction. If the variable identified in the reference parameter is undefined, the target value remains unchanged.

The ZBREAK command cannot specify the target parameter as a watchpoint.

Example

This example presents a generalized routine for outputting the data values for all the nodes in a user-specified array. It can accommodate arrays with any number of levels. The code performs the same operation as the code shown in the example under the $ORDER function. Instead of requiring 23 lines, however, it requires only six and is not restricted as to the number of levels it can handle.

Start
READ !,"Array name: ",ary QUIT:ary=""
SET queryary=$QUERY(@ary@(""))
WRITE !,@queryary
FOR   {
    SET queryary=$QUERY(@queryary)
    QUIT:queryary=""
    WRITE !,@queryary
}
WRITE !,"Finished."
QUIT
The first `SET` command uses `$QUERY` with subscript indirection to initialize the global reference to the first existing node that contains data. For more information, refer to Indirection in Using Caché ObjectScript. (Like `$ORDER`, `$QUERY` accepts a null string to designate the first subscript in an array.) The first `WRITE` command outputs the value of the first node found. If it were omitted, the first node would be skipped.

In the `FOR` loop, `$QUERY` is used to retrieve the next node and update the global reference, whose contents are then output by the `WRITE` command. The postconditional `QUIT` terminates the loop when it finds a null string (""), indicating that `$QUERY` has reached the end of the array.

No `$DATA` tests are required, unless you wish to distinguish between pointer nodes ($DATA=11) and terminal nodes ($DATA=1).

**Notes**

**Using `$QUERY` to Traverse an Array**

Used repetitively, `$QUERY` can traverse an entire array in left-to-right, top-to-bottom fashion, returning the entire sequence of defined nodes. `$QUERY` can start at the point determined by the subscript specified for `reference`. It proceeds along both the horizontal and vertical axes. For example:

```
SET exam=`QUERY(^client(4,1,2))
```

Based on this example, `$QUERY` might return any of the following values, assuming a three-level array:

<table>
<thead>
<tr>
<th>Value</th>
<th>Returned by the <code>$QUERY</code> Function If...</th>
</tr>
</thead>
<tbody>
<tr>
<td>^client(4,1,3)</td>
<td>If ^client(4,1,3) exists and contains data.</td>
</tr>
<tr>
<td>^client(4,2)</td>
<td>If ^client(4,1,3) does not exist or does not contain data and if ^client(4,2) does exist and contains data.</td>
</tr>
<tr>
<td>^client(5)</td>
<td>If ^client(4,1,3) and ^client(4,2) do not exist or do not contain data and if ^client(5) does exist and contains data.</td>
</tr>
<tr>
<td>null string (&quot;&quot;)</td>
<td>If none of the previous global references exist or contain data; <code>$QUERY</code> has reached the end of the array.</td>
</tr>
</tbody>
</table>

With a direction value of -1, `$QUERY` can traverse an entire array in reverse order in right-to-left, bottom-to-top fashion.

**`$QUERY` Compared to `$ORDER`**

`$QUERY` differs from the `$ORDER` function in that `$QUERY` returns a full global reference, while `$ORDER` returns only the subscript of the next node. `$ORDER` proceeds along only the horizontal axis, across nodes at one level.

`$QUERY` also differs from `$ORDER` in that it selects only those existing nodes that contain data. `$ORDER` selects existing nodes, regardless of whether or not they contain data. Where `$ORDER` performs an implicit test for existence ($DATA'=0), `$QUERY` performs an implicit test for both existence and data ($DATA'=0 and $DATA'=10). Note, however, that `$QUERY` does not distinguish between pointer nodes ($DATA=11) and terminal nodes ($DATA=1) that contain data. To make this distinction, you must include appropriate `$DATA` tests in your code.

Like `$ORDER`, `$QUERY` is typically used with loop processing to traverse the nodes in an array that doesn’t use consecutive integer subscripts. `$QUERY` simply returns the global reference of the next node with a value. `$QUERY` provides very compact code for accessing global arrays.

Like the `$NAME` and `$ORDER` functions, `$QUERY` 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:

```
SET a=`^client(1)
SET x=2
SET z=$QUERY(`(x))
```
The first `SET` command establishes the current global reference, including the level for the reference. The second `SET` command sets up a variable for use with subscripts. The `$QUERY` function uses a **naked global reference** to return the full global reference for the next node following `^client(2)`. For example, the returned value might be `^client(2,1)` or `^client(3)`.

### $QUERY and Extended Global References

You can control whether `$QUERY` returns global references in Extended Global Reference form on a per-process basis using the `RefInKind()` method of the `%SYSTEM.Process` class. The system-wide default behavior can be established by setting the `RefInKind` property of the `Config.Miscellaneous` class.

For further details on extended global references, see the **Global Structure** chapter in *Using Caché Globals*.

### See Also

- `$DATA` function
- `$NAME` function
- `$ORDER` function
- `$QLENGTH` function
- `$QSUBSCRIPT` function
$RANDOM

Returns a pseudo-random integer value in the specified range.

$RANDOM(range)
$R(range)

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>range</td>
<td>A nonzero positive integer used to specify the upper bound of the range of possible random numbers.</td>
</tr>
</tbody>
</table>

Description

$RANDOM returns a pseudo-random integer value between 0 and range-1 (inclusive). Thus $RANDOM(3) returns 0, 1, 2, but not 3. Returned numbers are uniformly distributed across the specified range.

$RANDOM is sufficiently random for most purposes. Applications that require strictly random values should use the GenCryptRand() method of the %SYSTEM.Encryption class.

Parameters

range

This value specifies the upper bound of the range of possible random numbers; the highest random number being range minus 1. The range value can be a nonzero positive integer value, the name of an integer variable, or any valid ObjectScript expression that evaluates to a nonzero positive integer. The maximum range value is 1E17 (100000000000000000); specifying a value beyond this maximum results in a <FUNCTION> error. $RANDOM(1) is valid, but always returns 0. $RANDOM(0) results in a <FUNCTION> error.

Examples

The following example returns a random number from 0 through 24 (inclusive).

WRITE $RANDOM(25)

To return a random number with a fractional portion, you can use the concatenation operator (_) or the addition operator (+), as shown in the following example:

SET x=$RANDOM(10)_$RANDOM(10)/10
WRITE ",x
SET y=$RANDOM(10)+($RANDOM(10)/10)
WRITE ",y

This program returns numbers with one fractional digit, ranging between .0 and 9.9 (inclusive). Using either operator, Caché deletes any leading and trailing zeros (and the decimal point, if the fractional portion is zero). However, if both $RANDOM functions return zero (0 and .0), Caché returns a zero (0).

The following example simulates the roll of two dice:

Dice
FOR { 
    READ "Roll dice? ",reply#1
    IF "Yy"[reply,reply'="" { 
        WRITE ","Pair of dice: 
        WRITE $RANDOM(6)+1,"+",$RANDOM(6)+1,!
    }
    ELSE { QUIT } 
}


$REPLACE

Returns a new string that consists of a string-for-string substring replacement from an input string.

$REPLACE(string,oldsub,newsub,start,count,case)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The source string. It can be a numeric value, a string literal, the name of a variable, or any valid ObjectScript expression. If string is an empty string (&quot;&quot;), $REPLACE returns an empty string.</td>
</tr>
<tr>
<td>oldsub</td>
<td>The substring to search for in string. It can be a numeric value, a string literal, the name of a variable, or any valid ObjectScript expression. If oldsub is an empty string (&quot;&quot;), $REPLACE returns string.</td>
</tr>
<tr>
<td>newsub</td>
<td>The replacement substring substituted for instances of oldsub in string. It can be a numeric value, a string literal, the name of a variable, or any valid ObjectScript expression. If newsub is an empty string (&quot;&quot;), $REPLACE returns string with the occurrences of oldsub removed.</td>
</tr>
<tr>
<td>start</td>
<td>Optional — Character count position within string where substring search is to begin. String characters are counted from 1. A value of 0, a negative number, a nonnumeric string or an empty string are equivalent to 1. If omitted, 1 is assumed. If start &gt; 1, the substring of string beginning with that character is returned, with substring substitutions (if any) performed. If start &gt; $LENGTH(string), $REPLACE returns the empty string (&quot;&quot;).</td>
</tr>
<tr>
<td>count</td>
<td>Optional — Number of substring substitutions to perform. If omitted, the default value is -1, which means perform all possible substitutions. A value of 0, a negative number other than -1, a nonnumeric string or an empty string are equivalent to 0 which means perform no substitutions. count must be used in conjunction with start.</td>
</tr>
<tr>
<td>case</td>
<td>Optional — Boolean flag indicating whether matching of oldsub in string is to be case-sensitive. 0 = case-sensitive (the default). 1 = not case-sensitive. Any nonzero number is equivalent to 1. Any nonnumeric value is equivalent to 0. Placeholder commas can be supplied when start or count are not specified.</td>
</tr>
</tbody>
</table>

Description

The $REPLACE function returns a new string that consists of a string-for-string replacement of the input string. It searches string for the oldsub substring. If $REPLACE finds a match, it replaces the oldsub substring with newsub and returns the resulting string. The newsub parameter value may be long or shorter than oldsub; newsub may be an empty string.

By default, $REPLACE begins at the start of string and replaces every instance of oldsub. You can use the optional start parameter to begin comparisons at a specified character count location within the string. The returned string is a substring of string that begins at the start location and replaces every instance of oldsub from that point. You can use the optional count parameter to replace only a specified number of matching substrings.

By default, $REPLACE substring matching is case-sensitive. You can use the optional case parameter to specify not case-sensitive matching.

$REPLACE supports long strings (strings of > 32,767 characters) on Caché instances for which long strings are enabled.

Note: Because $REPLACE can change the length of a string, you should not use $REPLACE on encoded string values, such as Caché %List elements.
$REPLACE and $TRANSLATE

$REPLACE performs string-for-string matching and replacement. $TRANSLATE performs character-for-character matching and replacement. $REPLACE can replace a single specified substring of one or more characters with another substring. $TRANSLATE can replace multiple specified characters with corresponding specified new characters. By default, both functions replace all matching instances in the string.

$REPLACE matching is case-sensitive by default, but can be invoked as not case-sensitive; $TRANSLATE matching is always case-sensitive. $REPLACE can specify the starting point for matching and/or the number of replacements to perform; $TRANSLATE replaces all matches in the source string.

Examples

The following example shows two ways of using $REPLACE. The first $REPLACE does not change the input string value. The second $REPLACE changes the input string value by setting it equal to the function’s return value:

```objectscript
SET str="The quick brown fox"
SET newstr=$REPLACE(str,"brown","red")
WRITE "source string: ",str,"new string: ",newstr,!!
// creates a new string, does not change str value
SET str=$REPLACE(str,"brown","silver")
WRITE "revised string: ",str
// creates a new string and replaces str with new string value
```

In the following example, invocations of $REPLACE match and substitute for the all instances of a substring, and the first two instances of a substring:

```objectscript
SET str="1110/1110/1100/1110"
WRITE ",before conversion ",str
SET newall=$REPLACE(str,"111","AAA")
WRITE ",after replacement ",newall
SET newsome=$REPLACE(str,"111","AAA",1,2)
WRITE ",after replacement ",newsome
```

In the following example, invocations of $REPLACE perform case-sensitive and not case-sensitive matching and replacement of all occurrences in the string:

```objectscript
SET str="Yes/yes/Y/YES/Yes"
WRITE ",before conversion ",str
SET case=$REPLACE(str,"Yes","NO")
WRITE ",after replacement ",case
SET nocase=$REPLACE(str,"Yes","NO",1,-1,1)
WRITE ",after replacement ",nocase
```

The following example compares the $REPLACE and $TRANSLATE functions:

```objectscript
SET str="A mom, o plom, o comal, Pomama"
WRITE ",before conversion ",str
SET s4s=$REPLACE(str,"om","an")
WRITE ",after replacement ",s4s
SET c4c=$TRANSLATE(str,"om","an")
WRITE ",after translation ",c4c
```

$REPLACE returns "A man, o plan, o canal, Panama"

$TRANSLATE returns "A nan, a plan, a canal, Panana"

In the following example, the four-parameter form of $REPLACE returns only the part of the string beginning with the start point, with the string-for-string replacements performed:

```objectscript
SET str="A mon, a plon, a conal, Ponama"
WRITE ",before start replacement ",str
SET newstr=$REPLACE(str,"on","an",8)
WRITE ",after start replacement ",newstr
```

$REPLACE returns "a plan, a canal, Panama"
See Also

- $TRANSLATE function
- $EXTRACT function
- $PIECE function
- $REVERSE function
- $ZCONVERT function
$REVERSE

Returns the characters in a string in reverse order.

$REVERSE(string)
$RE(string)

Parameter

| string | A string or expression that evaluates to a string. |

Description

$REVERSE returns the characters in string in reverse order. The string can contain 8-bit characters or 16-bit Unicode characters. For further details on Caché Unicode support, refer to Unicode in Using Caché ObjectScript.

Surrogate Pairs

$REVERSE does not recognize surrogate pairs. Surrogate pairs are used to represent some Chinese characters and to support the Japanese JIS2004 standard. You can use the $WISWIDE function to determine if a string contains a surrogate pair. The $WREVERSE function recognizes and correctly parses surrogate pairs. $REVERSE and $WREVERSE are otherwise identical. However, because $REVERSE is generally faster than $WREVERSE, $REVERSE is preferable for all cases where a surrogate pair is not likely to be encountered.

Examples

The following WRITE commands shows the return value from $REVERSE. The first returns “CBA”, the second returns 321.

    WRITE !,$REVERSE("ABC")
    WRITE !,$REVERSE(123)

You can use the $REVERSE function with other functions to perform search operations from the end of the string. The following example demonstrates how you can use $REVERSE with the $FIND and $LENGTH functions to locate the last example of a string within a line of text. It returns the position of that string as 33:

    SET line="THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG."
    SET position=$LENGTH(line)+2-$FIND($REVERSE(line),$REVERSE("THE"))
    WRITE "The last THE in the line begins at ",position

See Also

- $FIND function
- $SEXTRACT function
- $LENGTH function
- $PIECE function
- $WISWIDE function
- $WLENGTH function
- $WREVERSE function
$SCONVERT

Converts a binary encoded value to a number.

$SCONVERT(s,format,endian,position)
$SCL(s,format,endian,position)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>A string of 8-bit bytes which encode for a number. Limitations on valid values are imposed by the \textit{format} selected.</td>
</tr>
<tr>
<td>format</td>
<td>One of the following format codes, specified as a quoted string: S1, S2, S4, S8, U1, U2, U4, F4, or F8.</td>
</tr>
<tr>
<td>endian</td>
<td>Optional — A boolean value, where 0 = little-endian and 1 = big-endian. The default is 0.</td>
</tr>
<tr>
<td>position</td>
<td>Optional — The character position in the string of 8-bit bytes at which to begin conversion. Character positions are counted from 1. The default value is 1. If you specify \textit{position}, you must either specify \textit{endian} or a placeholder comma.</td>
</tr>
</tbody>
</table>

Description

$SCONVERT$ converts \( s \) from an encoded string of 8-bit bytes to a numeric value, using the specified \textit{format}. The following are the supported \textit{format} codes:

- \textbf{S1} Signed integer encoded into a string of one 8-bit bytes. The value must be in the range \(-128 \) through \( 127 \), inclusive.
- \textbf{S2} Signed integer encoded into a string of two 8-bit bytes. The value must be in the range \(-32768 \) through \( 32767 \), inclusive.
- \textbf{S4} Signed integer encoded into a string of four 8-bit bytes. The value must be in the range \(-2147483648 \) through \( 2147483647 \), inclusive.
- \textbf{S8} Signed integer encoded into a string of eight 8-bit bytes. The value must be in the range \(-9223372036854775808 \) through \( 9223372036854775807 \), inclusive.
- \textbf{U1} Unsigned integer encoded into a string of one 8-bit bytes. The maximum value is 256.
- \textbf{U2} Unsigned integer encoded into a string of two 8-bit bytes. The maximum value is 65535.
- \textbf{U4} Unsigned integer encoded into a string of four 8-bit bytes. The maximum value is 4294967295.
- \textbf{F4} IEEE floating point number encoded into a string of four 8-bit bytes.
- \textbf{F8} IEEE floating point number encoded into a string of eight 8-bit bytes.

String \( s \) must contain sufficient characters starting at and following the specified character \textit{position} to satisfy the number of 8-bit bytes required by the \textit{format} code. For example, $\$SCONVERT\left(s, "S4", 0, 9\right)$ requires that the length of \( s \) be at least 12 characters because the decoded result comes from the character positions 9, 10, 11 and 12. Values beyond this range result in a \texttt{<VALUE OUT OF RANGE>} error.

$\$SCONVERT$ is intended only for use on 8-bit byte strings. If $\$SCONVERT$ is on a Unicode instance of Caché and if any of the characters in the encoded string are in the range $\$CHAR(256)$ through $\$CHAR(65536)$ the return value is unpredictable.
If argument s is a numeric value, it is converted to a string containing its canonical numeric form before it is decoded.

You can use the IsBigEndian() class method to determine which bit ordering is used on your operating system platform: 1=big-endian bit order; 0=little-endian bit order.

WRITE $SYSTEM.Version.IsBigEndian()

$SCONVERT provides the inverse of the $NCONVERT operation.

Examples

In the following example, $SCONVERT converts a two-byte binary encoded value to a number:

```
SET x=$NCONVERT(258,"U2")
ZZDUMP x
SET y=$SCONVERT(x,"U2")
WRITE !,y
```

The following example, $SCONVERT converts a two-byte binary encoded value in big-endian order to a number:

```
SET x=$NCONVERT(258,"U2",1)
ZZDUMP x
SET y=$SCONVERT(x,"U2",1)
WRITE !,y
```

See Also

- $NCONVERT function
$SELECT

Returns the value associated with the first true expression.

$SELECT(expression:value,...)
$S(expression:value,...)

Parameters

<table>
<thead>
<tr>
<th>expression</th>
<th>The select test for the associated value parameter.</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>The value to be returned if the associated expression evaluates to true.</td>
</tr>
</tbody>
</table>

Description

The $SELECT function returns the value associated with the first expression that evaluates to true (1). Each $SELECT argument is a pair of expressions separated by a colon. The left half is a truth-valued expression. The right half can be any expression. In the following example, the truth values of the first three expressions are tested; if none of them evaluate to true, the final expression (which always evaluates to true) returns its value:

WRITE $SELECT(x=1:"1st is True",x=2:"2nd is True",x=3:"3rd is True",1:"The Default")

The specified list of expression:value pairs can be of any length. $SELECT evaluates the parameters from left to right. When $SELECT discovers a truth-valued expression with the value of true (1), it returns the matching expression to the right of the colon. $SELECT stops evaluation after it discovers the left-most true truth-valued expression. It never evaluates later pairs on the parameter list.

You can construct complex logic by nesting $SELECT functions. Like all evaluated truth conditions, a NOT logical operator (') can be applied to a nested $SELECT.

Parameters

expression

The select test for the associated value parameter. It can be any valid Caché relational or logical expression. If no expression evaluates to true, the system generates a $SELECT error. To prevent an error from disrupting an executing routine, the final expression can be the value 1, which always evaluates to true.

When expression is a string or numeric, any non-zero numeric value evaluates to true. A zero numeric value or a non-numeric string evaluates to false.

value

The value to be returned if the associated expression evaluates to true. It can be a numeric value, a string literal, a variable name, or any valid ObjectScript expression. If you specify an expression for value, it is evaluated only after the associated expression evaluates to true. If value contains a subscripted global reference, it changes the naked indicator when it is evaluated. For this reason, be careful when using naked global references either within or immediately after a $SELECT function. For more details on the naked indicator, see Naked Global Reference in Using Caché Globals.

Examples

To ensure that a $SELECT error never results, you should always include the value 1 as the last expression with an appropriate default value. This is shown in the following example:
Start
  READ !,"Which level?: ",a
  QUIT:a=""
  SET x=$SELECT(a=1:"Level1",a=2:"Level2",a=3:"Level3",1:"Start")
  DO @x
  Level1()
  WRITE !,"This is Level 1"
  Level2()
  WRITE !,"This is Level 2"
  Level3()
  WRITE !,"This is Level 3"

If the user enters a value other than 1, 2, 3, or the null string, control is passed back to the top of the routine.

You can use $SELECT to replace multiple IF clauses. The following example uses IF, ELSEIF, and ELSE clauses to determine whether a number is odd or even:

OddEven()
  READ !,"Enter an Integer: ",x
  QUIT:x=""
  WRITE !,"The input value is 
  IF 0=$ISVALIDNUM(x) { WRITE "not a number" }
  ELSEIF x=0 { WRITE "zero" }
  ELSEIF ""=$NUMBER(x,"I") { WRITE "not an integer" }
  ELSEIF x#2=1 { WRITE "odd" }
  ELSE { WRITE "even" }
  DO OddEven

The following example also accepts a number and determines if the number is odd or even. It uses $SELECT to replace the IF command in the previous example:

OddEven()
  READ !,"Enter an Integer: ",x
  QUIT:x=""
  WRITE !,"The input value is 
  WRITE $SELECT(0=$ISVALIDNUM(x):"not a number",x=0:"zero",
  ""=$NUMBER(x,"I"):"not an integer",x#2=1:"odd",1:"even")
  DO OddEven

See Also

- $CASE function
$SEQUENCE

Increments a global variable shared by multiple processes.

$SEQUENCE(gvar)  
SET $SEQUENCE(gvar)=value

Parameter

| gvar | The variable whose value is to be incremented. Commonly, gvar is a global variable ("gvar"), either subscripted or unsubscripted. The variable need not be defined. If gvar is not defined, or is set to the null string (""), $SEQUENCE treats it as having an initial value of zero and increments accordingly, returning a value of 1.
|      | You cannot specify a literal value for gvar. You cannot specify a simple object property reference as gvar; you can specify a multidimensional property reference as gvar with the syntax obj.property.

Description

$SEQUENCE provides a fast way for multiple processes to obtain unique (non-duplicate) integer indices for the same global variable. For each process, $SEQUENCE allocates a sequence (range) of integer values. Subsequent calls to $SEQUENCE increment to the next value in the allocated sequence for that process. When a process consumes all of the integer values in its allocated sequence, it is automatically assigned a new sequence of integer values. $SEQUENCE automatically determines the size of the sequence of integer values to allocate. It determines the size of the allocated sequence separately for each sequence allocation. In some cases, this sequence may be a single integer.

$SEQUENCE always increments an integer value by 1. By default, $SEQUENCE assigns positive integers, beginning with 1. However, $SEQUENCE can be set to a negative integer; negative integers are incremented towards zero. If $SEQUENCE was set to a negative integer, a subsequent call can assign zero as an increment. Setting gvar to a non-integer numeric value generates an <ILLEGAL VALUE> error.

$SEQUENCE is intended to be used when multiple processes concurrently increment the same global. Both $SEQUENCE and $INCREMENT can perform this operation, but $SEQUENCE commonly provides better performance. The order in which $SEQUENCE allocates indices is different from the $INCREMENT order. $SEQUENCE can allocate a sequential range of increments to a process, rather than the $INCREMENT behavior of allocating single integer increments to each process. This can substantially improve performance by reducing process collision and synchronization. It can also improve data block performance when inserting records, because sequential record IDs are grouped by process.

When a process calls $SEQUENCE, one of the following occurs:

- The ‘gvar global variable is undefined because no process has defined this variable. The integers returned by $SEQUENCE will start with 1.
- The ‘gvar global variable was last modified by a $SEQUENCE call from another process. The integers returned by $SEQUENCE will start with the first integer after the sequence allocated to the other process.
- The ‘gvar global variable was last modified by a SET $SEQUENCE called from any process. The integers returned by $SEQUENCE will start with the first integer after the value to which $SEQUENCE was set.

The size of the sequence that $SEQUENCE allocates to a process depends on an internal timestamp. When a process invokes $SEQUENCE for the second time, Caché compares the prior timestamp with the current time. Depending on the
duration between these $SEQUENCE calls, Caché allocates either a single increment or a calculated sequence of increments to the process:

- Allocated sequence is 1: $SEQUENCE behaves like $INCREMENT.
- Allocated sequence is > 1: $SEQUENCE uses this per-process sequence of increments. Each process uses its allocated sequence, then is assigned a new sequence.

For example, Process A and Process B are both incrementing the same global. The first time each process increments the global it is a single increment. The next time each process increments the global, Caché compares the two $SEQUENCE operations and calculates a sequence of increments (this sequence may be one integer). Subsequent $SEQUENCE operations use up these per-process sequences before re-allocating increments. This might result in increments such as the following: A1, B2 (single increments setting the clock), A3 (Cache compares A1 & A3, allocates 4, 5, 6, 7 to Process A), B8 (Cache compare B2 and B8, allocates 9, 10, 11 to Process B). The full increment sequence might be as follows: A1, B2, A3, A4, B8, A5, A6, B9, A7, B10, B11.

If a process does not use all of its allocated sequence, the remaining numbers are unused, gaps in the increment sequence.

The following example shows the difference between the increment integer returned by $SEQUENCE (the current sequence number) and the value of $gvar (the highest allocated sequence number):

```ObjectScript
SET $SEQUENCE("myseq")=""
FOR i=1:1:15 {WRITE "increment:"$,SEQ("myseq")," allocated:"$myseq,"!
```

For further details on using $SEQUENCE with global variables, see Using Multidimensional Storage (Globals) in Using CachéGlobals.

**Dedicated Global Variable**

Once a sequence has been started by the first call on $SEQUENCE (^gvar), all future changes to the value of ^gvar during the life of that sequence can only be made by calling $SEQUENCE (^gvar). Using any other function or statement to change the value of ^gvar causes the sequence to become undefined.

Restrictions on the use of $SEQUENCE and $INCREMENT on the same global are described below.

**SET $SEQUENCE**

You can use SET $SEQUENCE to kill or reset a $SEQUENCE global. SET $SEQUENCE resets the global variable and deallocates sequences of integers allocated to other processes.

- SET $SEQUENCE (^gvar)="" kills the specified global node and notifies all jobs with cached $SEQUENCE numbers to purge their current increment value. The first call to $SEQUENCE increments to 1. SET $SEQUENCE (^gvar)="" only kills the specified global node; it does not kill that node’s descendants (if any).
- SET $SEQUENCE (^gvar)=n (where n is an integer) resets the specified global node to n, and notifies all jobs with cached $SEQUENCE numbers to purge their current increment value. Subsequent invocations of $SEQUENCE on all jobs will use the new n increment starting value.

If SET $SEQUENCE attempts to set ^gvar to a fractional number, an <ILLEGAL VALUE> error occurs. If SET $SEQUENCE attempts to set ^gvar to a non-numeric string, ^gvar is set to 0.

You cannot use KILL ^gvar or SET ^gvar to kill or reset a $SEQUENCE global because these commands do not deallocate sequences of integers allocated to processes.
Parameter

**gvar**

A variable containing an integer value to be incremented. The variable does not need to be defined; the first call to $SEQUENCE defines an undefined variable as 0 then increments its value to 1. The *gvar* value must be a positive or negative integer.

Commonly, the *gvar* parameter is a global variable, either subscripted or unsubscripted: *^gvar*. It can contain an extended global reference. If a subscripted global variable, it can be specified using a naked global reference.

The *gvar* parameter can be a local variable or process-private global. However, because $SEQUENCE is intended for use across processes, this usage is not meaningful, in most cases. Using $SEQUENCE on a local variable or process-private global is the same as using $INCREMENT with a numeric increment of 1. The $SEQUENCE restrictions described below concerning locking, journaling, and transaction rollback do not apply to local variables or process-private globals. Using $SEQUENCE on a local variable or process-private global has the same error behavior as $INCREMENT; this is different from the error behavior for $SEQUENCE on a global variable, as described in the next section.

The *gvar* parameter can be a multidimensional property reference. For example, $SEQUENCE(..Count). It cannot be a non-multidimensional object property. Attempting to increment a non-multidimensional object property results in an <OBJECT DISPATCH> error.

$SEQUENCE cannot increment special variables, even those that can be modified using SET. Attempting to increment a special variable results in a <SYNTAX> error.

### Incrementing Very Large Numbers

The integers returned by $SEQUENCE are in the range -9223372036854775807 to 9223372036854775806 (-2**63+1 to 2**63-2). Attempting a SET $SEQUENCE on a global variable with an integer beyond this range generates an <ILLEGAL VALUE> error.

In the following example, $SEQUENCE on a global variable can be set to 9.223372036854775800E18, but incrementing this number past the range limit generates a <MAXINCREMENT> error. You can run this example repeatedly to perform “slow increments” and “fast increments”. Note that “fast increments” in this example may result in <MAXINCREMENT> before actually incrementing to the range limit, because $SEQUENCE is attempting to allocate a sequence of numbers beyond the range limit:

```objectscript
TRY {
  SET rand=$RANDOM(2)
  SET $SEQUENCE(^bignum)=9.223372036854775800E18
  IF rand=0 { WRITE "slow increments: ";
    FOR x=1:1:10 {WRITE $SEQUENCE(^bignum), " increment #",x,!
    HANG .5 }
  }
  IF rand=1 {WRITE "fast increments: ";
    FOR i=1:1:10 {WRITE $SEQUENCE(^bignum), " increment #",i,!
    }
  }
  CATCH exp { WRITE ",in the CATCH block!
    IF 1-exp.%IsA("%Exception.SystemException") { 
      WRITE "System exception!",!
      WRITE "Name: ",%ZCVT(exp.Name, "O", "HTML"),!
      WRITE "Location: ",exp.Location,!
    }
    ELSE { WRITE "unknown error!
    }
  }

These types of errors only occur when incrementing a global variable.

$SEQUENCE or $INCREMENT

$SEQUENCE is intended specifically for integer increment operations involving multiple simultaneous processes. $INCREMENT is a more general increment/decrement function:
$SEQUENCE increments an integer by 1. $INCREMENT increments or decrements any numeric value by any specified numeric value.

$SEQUENCE can allocate a sequence of increments to a process. $INCREMENT allocates only a single increment.

SET $SEQUENCE can be used to change or undefine (kill) a global. $INCREMENT cannot be used on the left side of the SET command.

Note: $SEQUENCE and $INCREMENT may be used on the same global variable only when performing an operation that simply increments a numeric value, such as Id allocation. Any other use of $SEQUENCE and $INCREMENT on the same global may produce unpredictable results and is not recommended.

Locking and Simultaneous Global Increments

$SEQUENCE uses special, efficient locking techniques that only synchronize $SEQUENCE calls with other $SEQUENCE calls. Attempting to use the LOCK command on a global used by $SEQUENCE will have no effect on $SEQUENCE. For example, suppose Process 1 executes a lock on ^COUNTER:

LOCK ^COUNTER

Then suppose, Process 2 increments ^COUNTER:

SET x=$SEQUENCE(^COUNTER)

Process 2 is not prevented from incrementing ^COUNTER by the lock held by Process 1.

$SEQUENCE and Transaction Processing

- Locking: The common usage for $SEQUENCE is to increment a counter before adding a new entry to a database. $SEQUENCE provides a way to do this very quickly, avoiding the use of the LOCK command. The trade-off for this is that gvar is not locked. The gvar may be incremented by one process within a transaction and, while that transaction is still processing, be incremented by another process in a parallel transaction.

- Rollback: Calls to $SEQUENCE are not journaled. Therefore, rolling back a transaction will not change the value of gvar. Any integer values allocated by $SEQUENCE during a rolled back transaction will not be available for allocation by any future $SEQUENCE call.

For further details on using $SEQUENCE in a distributed database environment, refer to The $INCREMENT Function and Application Counters in the “Developing Distributed Applications” chapter of the Caché Distributed Data Management Guide.

See Also

- $INCREMENT function
- TROLLBACK command
- $GET function
- $ZINCREMENT function
- Using ObjectScript for Transaction Processing in Using Caché ObjectScript
$SORTBEGIN

Initiates a sorting mode to improve performance of multiple sets to a global.

$SORTBEGIN(set_global)

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_global</td>
<td>A global variable name.</td>
</tr>
</tbody>
</table>

Description

$SORTBEGIN initiates a special sorting mode during which SET operations to the specified target global are redirected to a process-private temporary area and sorted into subsets. This mode is ended with a call to $SORTEND which copies the data into the target global reference. When the special sorting mode is in effect, all sets to the target global reference and any of its descendants are affected.

$SORTBEGIN is designed to help the performance of operations, such as index building, where a large amount of unordered data needs to be written to a global. When the amount of data written approaches or exceeds the amount of available buffer pool memory, performance can suffer drastically. $SORTBEGIN solves this problem by guaranteeing that data is written to the target global in sequential order, thus minimizing the number of physical disk accesses needed. It does this by writing and sorting data into one or more temporary buffers (using space in the ^CacheTemp global if needed) and then, when $SORTEND is called, copying the data sequentially into the target global.

While $SORTBEGIN is in effect, data read from the target global will not reflect any SET operations. You cannot use $SORTBEGIN in cases where you need to read global values from the same global in which you are inserting values.

Caché object and Caché SQL applications automatically make use of $SORTBEGIN for index and temporary index creation. The $SORTBEGIN sorting mode can be terminated without writing data to the target global by calling $SORTEND with its optional second parameter set to 0.

If successful, $SORTBEGIN returns a nonzero integer value. If unsuccessful, $SORTBEGIN returns zero.

Sorting Mode Errors

Invoking some operations between the $SORTBEGIN and $SORTEND result in Caché issuing an error code:

- If the mapping of the namespace of set_global is changed between $SORTBEGIN and $SORTEND, a <NAMESPACE> error occurs when you invoke $SORTEND. However, if $SORTBEGIN specifies set_global with an implied namespaces, subsequent namespace mapping changes have no effect on $SORTEND. Global references with implied namespace and global references with explicit namespaces should not be mixed in the same sort operation. For information on modifying namespaces, see Configuring Namespaces in the Caché System Administration Guide.

- If you establish a $SORTBEGIN global, and then issue a $SORTBEGIN for an ancestor or descendent of that global, Caché issues a <DUPLICATEARG> error. For example, if you invoke $SORTBEGIN(^test(1,2,3)), the following function calls result in a <DUPLICATEARG> error: $SORTBEGIN(^test(1,2)) or $SORTBEGIN(^test(1,2,3,4)).

See Also

- $SORTEND function
$SORTEND

Concludes the sorting mode initiated by $SORTBEGIN.

$SORTEND(set_global,dosort)

Parameters

<table>
<thead>
<tr>
<th>parameter</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_global</td>
<td>Optional — A global variable that was specified in a corresponding $SORTBEGIN. If omitted, $SORTEND concludes all $SORTBEGIN operations for the current process.</td>
</tr>
<tr>
<td>dosort</td>
<td>Optional — A flag parameter. If 1, Caché performs the sort operation initiated by $SORTBEGIN and copies the sorted data into set_global. If 0, Caché terminates the sort operation without copying any data. The default is 1.</td>
</tr>
</tbody>
</table>

Description

$SORTEND specifies the end of a special sorting mode initiated by $SORTBEGIN on a specific target global. The value of the $SORTEND set_global must match the corresponding $SORTBEGIN set_global.

If you omit set_global, $SORTEND ends all current sorting modes initiated by all active $SORTBEGIN functions for the current process. Therefore, $SORTEND() or $SORTEND(),1 end and commit all current sorting modes for the process; $SORTEND(),0 aborts all current sorting modes for the process.

- If successful, $SORTEND returns a positive integer count of the total number of global nodes set. When set_global is specified, this is the number of sets applied to the specified set_global variable. When set_global is omitted, this is the number of sets applied to all current $SORTBEGIN set_global variables. This integer count is returned regardless of the dosort flag setting.
- If unsuccessful, $SORTEND returns -1. For example, if $SORTEND specifies a set_global that does not have a corresponding active $SORTBEGIN.
- If no-op, $SORTEND returns 0. This can occur if you there are no sets applied to the specified set_global variable, or if there is no active $SORTBEGIN when you issue a $SORTEND that does not specify a set_global.

If the mapping of the namespace of set_global is changed between $SORTBEGIN and $SORTEND, a <NAMESPACE> error occurs when you invoke $SORTEND. However, if $SORTBEGIN specifies set_global with an implied namespaces, subsequent namespace mapping changes have no effect on $SORTEND. Global references with implied namespace and global references with explicit namespaces should not be mixed in the same sort operation. For information on modifying namespaces, see Configuring Namespaces in the Caché System Administration Guide.

Examples

The following example applies three sets to the global ^myyestest. $SORTEND returns 3. Because dosort is 1, these sets are applied, as shown by the $DATA function return values:

```
WRITE $SORTBEGIN(^myyestest),!
SET ^myyestest(1)="apple"
SET ^myyestest(2)="orange"
SET ^myyestest(3)="banana"
WRITE $SORTEND(^myyestest,1),!
WRITE $DATA(^myyestest(1)),!
WRITE $DATA(^myyestest(2)),!
WRITE $DATA(^myyestest(3))
KILL ^myyestest
```

The following example applies three sets to the global ^mynotest. $SORTEND returns 3. Because dosort is 0, these sets are not applied, as shown by the $DATA function return values:
WRITE $SORTBEGIN(^mynotest),!
  SET ^mynotest(1)="apple"
  SET ^mynotest(2)="orange"
  SET ^mynotest(3)="banana"
WRITE $SORTEND(^mynotest,0),!
WRITE $DATA(^mynotest(1)),!
WRITE $DATA(^mynotest(2)),!
WRITE $DATA(^mynotest(3))!
KILL ^mynotest

The following two examples specify two $SORTBEGIN operations, and within them apply three sets to the global ^mytesta and two sets to the global ^mytestb. $SORTEND does not specify a set_global, and therefore ends all current $SORTBEGIN operations and returns 5. Note that in both examples $SORTEND returns 5, though the first example commits these sets and the second example aborts these sets.

WRITE $SORTBEGIN(^mytesta),!
  SET ^mytesta(1)="apple"
  SET ^mytesta(2)="orange"
  SET ^mytesta(3)="banana"
WRITE $SORTBEGIN(^mytestb),!
  SET ^mytestb(1)="corn"
  SET ^mytestb(2)="carrot"
WRITE "$SORTEND returns: $, $SORTEND(,0),!
  WRITE "global sets committed?: $, $DATA(^mytesta(2))
KILL ^mytesta,^mytestb

WRITE $SORTBEGIN(^mytesta),!
  SET ^mytesta(1)="apple"
  SET ^mytesta(2)="orange"
  SET ^mytesta(3)="banana"
WRITE $SORTBEGIN(^mytestb),!
  SET ^mytestb(1)="corn"
  SET ^mytestb(2)="carrot"
WRITE "$SORTEND returns: $, $SORTEND(,0),!
  WRITE "global sets committed?: $, $DATA(^mytesta(2))
KILL ^mytesta,^mytestb

See Also

• $SORTBEGIN function
$STACK

Returns information about active contexts saved on the process call stack.

$STACK(context_level,code_string)
$ST(context_level,code_string)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>context_level</td>
<td>An integer specifying the zero-based context level number of the context for which information is requested. Supported values include 0, positive integers, and -1.</td>
</tr>
<tr>
<td>code_string</td>
<td>Optional — A keyword string that specifies the kind of context information that is requested. Supported values are “PLACE”, “MCODE”, and “ECODE”</td>
</tr>
</tbody>
</table>

Description

The $STACK function returns information on either the current execution stack or the current error stack, depending on the value of the $ECODE special variable. $STACK is most commonly used to return information on the current execution stack (also known as the process call stack).

Each time a routine invokes a **DO** command, an **XECUTE** command, or a user-defined function (but not a **GOTO** command), the context of the currently executing routine is saved on the *call stack* and execution starts in the newly created context of the called routine. The called routine, in turn, can call another routine, and so on, causing more saved contexts to be placed on the call stack.

The $STACK function returns information about these active contexts saved on your process call stack. $STACK also can return information about the currently executing context. However, during error processing, $STACK returns a snapshot of all the context information that is available when an error occurs in your application.

You can use the $STACK special variable to determine the current context level.

$ECODE and $STACK

The values returned by $STACK are dependent on the $ECODE special variable. If $ECODE is clear (set to the null string), $STACK returns the current execution stack. If $ECODE contains a non-null value, $STACK returns the current error stack.

Error stack context information is only available when the $ECODE special variable contains a non-null value. This can occur either when an error has occurred or when $ECODE is explicitly set to a non-null value. In this case, $STACK returns information about the error stack context rather than an active stack context at the specified context level.

When error stack context information is not available ($ECODE="") and you specify the current context level with the two-argument form of $STACK, Caché returns information about the currently executing command. To ensure consistent behavior when accessing the current execution stack, specify SET $ECODE="" before calling $STACK.

See Error Handling in Using Caché ObjectScript for more detailed information about error processing and your error process stack.

The One-Argument Form of $STACK

$STACK(context_level) returns a string that indicates how the specified context level was established. The following table describes the string values that can be returned:
<table>
<thead>
<tr>
<th>Context</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO</td>
<td>Returned when the specified context was established by a DO command.</td>
</tr>
<tr>
<td>XECUTE</td>
<td>Returned when the specified context was established by an XECUTE command or a BREAK command.</td>
</tr>
<tr>
<td>$$</td>
<td>Returned when the specified context was established by a user-defined function reference.</td>
</tr>
<tr>
<td>An ECODE string</td>
<td>The error code value of the error that caused the specified context to be added to the error stack. For example, $M26, When an error occurs at a context level where an error has already occurred, the context information is placed at the next higher error stack level; it is only returned when context information at the specified error stack context level is relocated information.</td>
</tr>
</tbody>
</table>

When the specified context level is zero (0) or is undefined, **$STACK** returns the null string.

You can also specify a -1 for the context level in the one-argument form of the **$STACK** function. In this case, **$STACK** returns the maximum context level for the information that is available that, during normal processing, is the context level number of the currently executing context. However, during error processing, **$STACK(-1)** returns whichever is greater:

- The maximum context level of your process error stack
- The context level number of the currently executing context

**The Two-Argument Form of $STACK**

**$STACK**(context_level, code_string) returns information about the specified context level according to the code_string you specify. A code_string must be specified as a quoted string; code_string values are not case-sensitive. For example, **$STACK(1,"PLACE")** or **$STACK(1,"place")**.

The following describes the code strings and the information returned when you specify each.

- **PLACE** — Returns the entry reference and command number of the last command executed at a specified context level. The value is returned in the following format for DO and user-defined function contexts: "label[+offset][\^routine name] +command". For XECUTE contexts, the following format is used: "@ +command".

- **MCODE** — Returns the source routine line, XECUTE string, or $ETRAP string that contains the last command executed at the specified context level. (Routine lines are returned in the same manner as those returned by the **$TEXT** function.)

  **Note:** During error processing, if memory is low while the error stack is being built or updated, you may not have enough memory to store the source line. In this case, the return value for the MCODE code string is the null string. However, the return value for the PLACE code string indicates the location.

- **ECODE** — The error code of any error that occurred at the specified context level (available only in error stack contexts).

When the requested information is not available at the specified context level, the two argument form of **$STACK** returns the null string.

**Example**

The following example demonstrates some of the information that **$STACK** can return:
ObjectScript Functions

```
STAC ;
SET $ECODE=""
XECUTE "DO First"
QUIT
First SET varSecond=$$Second()
QUIT
Second() FOR loop=0:1:$STACK(-1) {
  WRITE !,"Context level:",loop,?25,"Context type: ",$STACK(loop)
  WRITE !,?5,"Current place: ",$STACK(loop,"PLACE")
  WRITE !,?5,"Current source: ",$STACK(loop,"MCODE")
}
QUIT 1

>DO ^STAC
Context level: 0      Context type:
  Current place: @ +1
  Current source: DO ^STAC
Context level: 1      Context type: DO
  Current place: STAC+2^STAC +1
  Current source: XECUTE "DO First"
Context level: 2      Context type: XECUTE
  Current place: @ +1
  Current source: DO First
Context level: 3      Context type: DO
  Current place: First^STAC +1
  Current source: First SET Second=$$Second
Context level: 4      Context type: $$
  Current place: Second+2^STAC +4
  Current source: WRITE !,?5,"Current source: ",$STACK(loop,"MCODE")
```

Notes

**Cross-Namespace Routine Calls**

If a routine calls a routine in a different namespace, `$STACK` returns the namespace name as part of the routine name. For example, if a routine in the USER namespace calls a routine in the SAMPLES namespace, `$STACK` returns `^|"SAMPLES"|routinename`.

`$STACK` uses the caret (^) character as a delimiter. Therefore, if an implied namespace name includes the caret (^) character, Caché displays this namespace name character as the @ character.

**$STACK Counts Multiple-Argument Commands**

When you specify a multiple-argument command, the command count includes command keywords and all command arguments beyond the first. Consider the following multiple-argument command:

```
TEST
  SET X=1,Y=Z
```

In Caché, the `$STACK` statement, `$STACK(1,"PLACE")` returns "TEST^TEST +2" because the Y=Z argument counts as a separate command.

**$STACK with <STORE> Errors or Low Memory Conditions**

After a <STORE> error or under low-memory conditions, the information available normally through the application of the two-argument form of `$STACK` may not be available.

See Also

- DO command
- XECUTE command
- $ECODE special variable
- $ESTACK special variable
- $STACK special variable
- Error Handling in Using Caché ObjectScript
$TEXT

Returns a line of source code found at the specified location.

$TEXT(label+offset^routine)
$TEXT(@expr_atom)
$T(label+offset^routine)
$T(@expr_atom)

Parameters

<table>
<thead>
<tr>
<th>label</th>
<th>Optional — A line label in a routine. Must be a literal value; a variable cannot be used to specify label. Line labels are case-sensitive. If omitted, +offset is counted from the beginning of the routine.</th>
</tr>
</thead>
<tbody>
<tr>
<td>+offset</td>
<td>Optional — An expression that resolves to a positive integer that identifies the line to be returned as an offset number of lines. If omitted, the line identified by label is returned.</td>
</tr>
<tr>
<td>^routine</td>
<td>Optional — The name of a routine. Must be a literal value; a variable cannot be used to specify routine. (Note that the ^ character is a separator character, not part of the routine name.) If the routine is not in the current namespace, you can specify the namespace that contains the routine using an extended routine reference, as follows: ^&quot;namespace&quot;</td>
</tr>
<tr>
<td>@expr_atom</td>
<td>An expression atom that uses indirection to supply a location. Resolves to some form of label+offset^routine.</td>
</tr>
</tbody>
</table>

Description

$TEXT returns a line of source code found at the specified location. If $TEXT does not find source code at the specified location, it returns the null string.

To identify a single line of source code, you must specify either a label, an +offset, or both. By default, $TEXT accesses the currently loaded routine. Either $TEXT is coded in the currently executing routine, or accesses the currently loaded routine as the static routine most recently loaded using ZLOAD. You can use ^routine to specify a routine location other than the currently loaded routine. You can use indirection (@expr_atom) to specify a location.

The +offset argument counts lines using the object code version of the routine. After modifying a routine, you must recompile the routine for $TEXT to correctly count lines and line offsets that correspond to the source version.

In the returned source code, if the first whitespace character in the line is a tab, $TEXT replaces it with a single space character. All other tabs and space characters are returned unchanged. Thus $PIECE($TEXT(line)," ",1) always returns a label, and $PIECE($TEXT(line)," ",2,*) returns all code except a label.

If you invoke $TEXT from the Terminal command line for the currently loaded routine, it can return comment lines of any type. However, entirely blank lines in a MAC routine, including those within a multiline comment, are removed by the compiler and therefore are neither counted nor returned by $TEXT.

When a routine is distributed as object code only, the ;; comment is the only comment type retained in the object code. Thus only ;; comments are available to $TEXT in those routines. For a ;; comment to be returned by $TEXT, it must either appear on its own line, or on the same line as a label. It cannot appear on a line containing a command, or a line declaring a function or subroutine. For further details on the different types of Caché comments, refer to Comments in Using Caché ObjectScript.
You can use the PRINT or ZPRINT commands to display a single line (or multiple lines) of source code from the currently loaded routine.

**Parameters**

**label**

The label within the current routine or, if the *routine* parameter is also supplied, a label in the specified routine. Must be specified as a literal, without quotes. Label names are case-sensitive, and may contain Unicode characters. A label may be longer than 31 characters, but must be unique within the first 31 characters. $TEXT matches only the first 31 characters of a specified label.

If you omit the *offset* option, or specify *label+0*, Caché prints the label line. *label+1* prints the line after the label. If *label* is not found in the routine, $TEXT returns the empty string.

**offset**

A positive integer specifying a line count, or as an expression that evaluates to a positive integer. The leading plus sign (+) is mandatory. If specified alone, the +offset specifies a line count from the beginning of the routine, with +1 being the first line of the routine. If specified with the *label* parameter, the line count is calculated from the label location, with +0 being the label line itself, and +1 being the line after the label. If +offset is larger than the number of lines in the routine (or the number of lines from *label* to the end of the routine) $TEXT returns the empty string.

You can specify an offset of +0. When *label* is specified, $TEXT(*mylabel+0*) is the same as $TEXT(*mylabel*). If you invoke $TEXT(+0*), it returns the name of the currently loaded routine.

Caché resolves an +offset value to a canonical positive integer. A negative integer offset value generates a <NOLINE> error.

Note that Caché resolves numbers and numeric strings to canonical form, which involves removal of the leading plus sign (+). For this reason, you must specify the plus sign in the $TEXT function to use it as an offset.

To use a variable as the offset it must be preceded by a plus sign as shown in the following example:

```
SET x=7
WRITE $TEXT(x)  /* because there is no plus sign, search for a label named x */
WRITE $TEXT(+x) /* locates the offset +7 code line */
```

**routine**

If specified alone, it indicates the first line of code in the routine. If specified with only the *label* parameter, the line found at that specified label within the routine is returned. If specified with only the *offset* parameter, the line at the specified offset within the routine is returned. If both *label* and *offset* are supplied, the line found at the specified offset within the specified label within the routine is returned.

The *routine* argument must be specified as a literal, without quotes. You cannot use a variable to specify the *routine* name. The leading caret (^) is mandatory.

By default, Caché searches for the routine in the current namespace. If the desired routine resides in another namespace, you can specify that namespace using extended global reference. For example, $TEXT(*mylabel+2*^"SAMPLES"|myroutine*). Note that only vertical bars can be used here; square brackets cannot be used. You can specify the namespace portion of ^*routine* as a variable.

$TEXT returns the empty string if the specified routine or namespace does not exist, or if the user does not have access privileges for the namespace.

**expression atom (@expr_atom)**

An indirect argument that evaluates to a $TEXT argument (label+offset^routine). For more information, refer to Indirection in *Using Caché ObjectScript*. 
Examples

The following four examples demonstrate a routine saved with object code only. The first two examples return the referenced line, including the ;; comment. The third and fourth examples return the null string:

Start ;; this comment is on a label line
   WRITE $TEXT(Start)

Start
   ;; this comment is on its own line
   WRITE $TEXT(Start+1)

Start
   SET x="fred" ;; this comment is on a command line
   WRITE $TEXT(Start+1)

MyFunc() ;; this comment is on a function declaration line
   WRITE $TEXT(MyFunc)

The following example shows that only the first 31 characters of label are matched with the specified label:

StartabcdefghijklmnopqrstuvwxyzA ;; 32-character label
   WRITE $TEXT(StartabcdefghijklmnopqrstuvwxyzB)

The following example shows the $TEXT(label) form, which returns the line found at the specified label within the current routine. The label is also returned. If the user enters "?", the Info text is written out, along with the line label, and control returns to the initial prompt:

Start
   READ !,"Array name or ? for Info: ",ary QUIT:ary=""
   IF ary="?” { DO Info
      GOTO Start } ELSE { DO ArrayOutput(ary) }
   QUIT

Info ;; This routine outputs the first-level subscripts of a variable.
   QUIT
   ArrayOutput(val)
      SET i=1
      WHILE $DATA(@val@(i)) { DO Info
         WRITE !,"subscript ",i," is ",@val@(i)
         SET i=i+1
      } QUIT

The following example shows the $TEXT(label+offset) form, which returns the line found at the offset within the specified label, which must be within the current routine. If the offset is 0, the label line, with the label, is returned. This example uses a FOR loop to access multiline text, avoiding displaying the label or the multiline comment delimiters:

Start
   READ !,"Array name or ? for Info: ",ary QUIT:ary=""
   IF ary="?” { DO Info
      GOTO Start } ELSE { DO ArrayOutput(ary) }
   QUIT

Info FOR loop=2:1:6 { WRITE !,$TEXT(Info+loop) }
   /*
   This routine outputs the first-level subscripts of a variable.
   Specifically, it asks you to supply the name of the variable
   and then writes out the current values for each subscript
   node that contains data. It stops when it encounters a node
   that does not contain data.
   */
   QUIT
   ArrayOutput(val)
      SET i=1
      WHILE $DATA(@val@(i)) { DO Info
         WRITE !,"subscript ",i," is ",@val@(i)
         SET i=i+1
      } QUIT
The following example uses extended routine reference to access a line of code from a routine in the SAMPLES namespace. It accesses the first line of code after the ErrorTest label in the routine named myroutine. It can be executed from any namespace:

```
WRITE $TEXT(ErrorTest+1^"SAMPLES"|myroutine)
```

**Notes**

**Argument Indirection**

Indirection of the entire $TEXT argument is a convenient way to make an indirect reference to both the line and the routine. For example, if the variable ENTRYREF contains both a line label and a routine name, you can reference the variable:

```
$TEXT(@ENTRYREF)
```

rather than referencing the line and the routine separately:

```
$TEXT(@$PIECE(ENTRYREF,"^",1)^@$PIECE(ENTRYREF,"^",2))
```

**Edit Pointer**

If you specify a routine in $TEXT other than the current routine, Caché resets the edit pointer (current line location) to +0. This can affect execution of the ZINSERT command. You can access the $ZNAME special variable to determine the current routine.

**See Also**

- PRINT command
- ZINSERT command
- ZLOAD command
- ZPRINT command
- ZREMOVE command
- ZSAVE command
- ZZPRINT command
- Comments in *Using Caché ObjectScript*
- Labels in *Using Caché ObjectScript*
- Indirection in *Using Caché ObjectScript*
$TRANSLATE

Returns a new string that consists of character-for-character replacement of a source string.

$TRANSLATE(string, identifier, associator)
$STR(string, identifier, associator)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The source string. It can be a numeric value, a string literal, the name of a variable, or any valid ObjectScript expression.</td>
</tr>
<tr>
<td>identifier</td>
<td>A string consisting of one or more characters to search for in string. It can be a numeric value, a string literal, the name of a variable, or any valid ObjectScript expression.</td>
</tr>
<tr>
<td>associator</td>
<td>Optional — A string consisting of one or more replacement characters that correspond positionally to each character in identifier. It can be a numeric value, a string literal, the name of a variable, or any valid ObjectScript expression.</td>
</tr>
</tbody>
</table>

Description

The $TRANSLATE function returns a new string that consists of a character-for-character replacement of the source string. A $TRANSLATE operation can replace multiple different characters, but it can only replace a single character with (at most) a single character. It processes the string parameter one character at a time. It compares each character in the input string with each character in the identifier parameter. If $TRANSLATE finds a match, it performs one of the following actions on that character:

- The two-parameter form of $TRANSLATE removes those characters in the identifier parameter from the returned string.
- The three-parameter form of $TRANSLATE replaces the identifier character(s) found in the string with the positionally corresponding character(s) from the associator parameter and returns the resulting string. Replacement is performed on a character, not a string, basis. If the identifier parameter contains fewer characters than the associator parameter, the excess character(s) in the associator parameter are ignored. If the identifier parameter contains more characters than the associator parameter, the excess character(s) in the identifier parameter are deleted in the output string.

$TRANSLATE is case-sensitive.

The string, identifier, and associator parameters are normally specified as quoted strings. If the value of one of these parameters is purely numeric, string quotes are not required; however, because Caché will convert the parameter value to a canonical number before supplying the value to $TRANSLATE, this usage is not recommended.

$TRANSLATE and $REPLACE

$TRANSLATE performs character-for-character matching and replacement. $REPLACE performs string-for-string matching and replacement. $REPLACE can replace a single specified substring of one or more characters with another substring, or remove multiple instances of a specified substring. $TRANSLATE can replace multiple specified characters with corresponding specified replacement characters.

$TRANSLATE matching is always case-sensitive; $REPLACE matching is case-sensitive by default, but can be invoked as not case-sensitive. $TRANSLATE always replaces all matches in the source string; $REPLACE can specify the starting point for matching and/or the number of replacements to perform.
Examples

The following example shows two ways of using $TRANSLATE. The first $TRANSLATE does not change the input string value. The second $TRANSLATE changes the input string value by setting it equal to the function’s return value:

    SET str=“The quick brown fox”
    SET newstr=$TRANSLATE(str,”qbf”,”QBF”)  
    WRITE ”source string: “,str,”,”new string: “,newstr,!”
    // creates a new string, does not change str value
    SET str=$TRANSLATE(str,”qbf”,”QBF”)  
    WRITE ”revised string: “,str
    // creates a new string and replaces str with new string value

In the following example, a two-parameter $TRANSLATE removes Numeric Group Separators based on the setting for the current locale:

    AppropriatInput
    SET ds=##class(%SYS.NLS.Format).GetFormatItem(“DecimalSeparator”)  
    IF ds=”.“ (SET x=“1,462,543,13“)  
    ELSE (SET x=“1.462.543,13“)
    TranslateNum
    WRITE !,”before translation “,x
    SET ngs=##class(%SYS.NLS.Format).GetFormatItem(“NumericGroupSeparator”)  
    IF ngs=”.” {SET x=$TRANSLATE(x,”.”) }  
    ELSEIF ngs=”,” {SET x=$TRANSLATE(x,”,”) }  
    ELSEIF ngs=”.” {SET x=$TRANSLATE(x,”.”) }  
    ELSE {WRITE ”Non-standard NumericGroupSeparator:”, ngs  
    RETURN }  
    WRITE !,”after translation “,x

In the following example, a three-parameter $TRANSLATE replaces various Date Separator Characters with slashes. Note that the *associator* must specify “/” as many times as the number of characters in *identifier*:

    SET x(1)=“06-23-2014“
    SET x(2)=“06.24.2014“
    SET x(3)=“06/25/2014“
    SET x(4)=“06 26 2014“
    SET x(5)=“06 27 2014“
    FOR i=1:1:5{
        SET x(i)=$TRANSLATE(x(i),“- .|”,”////“)  
        WRITE ”x(”,i,”) :”,x(i),!
    }

In the following example, a three-parameter $TRANSLATE “simplifies” Spanish to basic ASCII by replacing accented letters with non-accented letters and removing the question and exclamation sentence prefix punctuation:

    SET esp=”¿Sabes lo que ocurrirá en el año 2016?”
    WRITE ”Spanish:”,!,esp,!
    SET iden=$CHAR(225)_$CHAR(233)_$CHAR(237)_$CHAR(241)_$CHAR(243)_$CHAR(250)_$CHAR(161)_$CHAR(191)
    SET asso=“aeinou”
    WRITE ”Identifier: “,iden,!
    WRITE ”Associator: “,asso,!
    SET spanglish=$TRANSLATE(esp,iden,asso)
    WRITE ”Spanglish:”,!,spanglish

Needless to say, this is not a recommended conversion for use on actual Spanish text.

See Also

- $EXTRACT function
- $PIECE function
- $REPLACE function
- $REVERSE function
- $ZCONVERT function
$VIEW

Returns the contents of memory locations.

$VIEW(offset,mode,length)
$V(offset,mode,length)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset</td>
<td>An offset, in bytes, from a base address within the memory region specified by mode. Interpretation is mode-dependent (see below.)</td>
</tr>
<tr>
<td>mode</td>
<td>Optional — The memory region whose base address will be used to locate the data. Default is -1.</td>
</tr>
<tr>
<td>length</td>
<td>Optional — The length of the data to be returned, in bytes. May also contain a letter “O” reverse order suffix. Default is 1.</td>
</tr>
</tbody>
</table>

**Description**

$VIEW returns the contents of memory locations.

The view buffer is a special memory area used to store blocks of data read from the Caché database (CACHE.DAT) with the VIEW command. After reading a block into the view buffer, you can use the $VIEW function with mode 0 to examine the contents of the view buffer. You must open the view buffer as device 63 in order to access it; when finished, you should close device 63.

You can also use $VIEW to return process summary information.

The $VIEW function is usually used when debugging and repairing Caché databases and system information.

**Parameters**

**offset**

The value of this argument depends upon the mode argument, as follows:

- When mode is 0, -1, or -2, specify a positive integer as the offset from the base address, in bytes, counting from 0.
- When mode is -3, or a positive integer, specify an address in the process address space. The value -1 can be used to retrieve a summary of the process state, as described in “Process Summary Information” below.
- When mode is -5, specify a positive integer that specifies the number of global nodes in the current block. In this case, odd values return full global references and even values return pointers or data.

**mode**

The possible values for mode are shown in the following table. If mode is omitted, the default is -1. Note that some values are implementation specific. Implementation restrictions are noted in the table.
### Mode | Memory Management Region | Base Address
---|---|---
0 | The view buffer | Beginning of view buffer
-1 | The process’s partition (default) | Beginning of partition
-2 | The system table | Beginning of system table
-3 | The address space for the current process. | 0
-5 | Global reference and data | Special. See “Using Mode -5,” later in the Notes section.
-6 | Reserved for InterSystems use | 
-7 | Used only by the integrity checking utility | Special.
n | When n is a positive number, the address space for the running process n, where n is the pid (value of the $JOB special variable) for that process. Treats offset and length the same as mode -3. | 0

**length**

A length in bytes, or a flag character. Interpretation depends on the mode and offset values:

- When mode is 0, -1, or -2, -3, or a positive integer (a pid), and offset is a positive integer, the length parameter can be:
  - A negative integer from -1 to the maximum string length (as a negative integer) to return that length of data as a string. $VIEW returns the specified number of characters starting at the address indicated by offset.
  - A positive integer in the range 1 through 8 (inclusive) to return the decimal value of the data. $VIEW returns from one to four contiguous bytes, or eight contiguous bytes, starting at the address indicated by offset.
  - A letter C or P as a quoted string to indicate a four-byte address on 32-bit systems, or an eight-byte address on 64-bit systems. When specifying C or P, you do not specify a length integer value.

  To return a byte value in reverse order (low-order byte at lowest address) append a letter O suffix to the length value and enclose the resulting string in double quotes.

  If the length parameter is omitted, the default is 1.

- When mode is -3, or a positive integer (a pid), and offset is -1, the length parameter is a flag that specifies the format of the summary information. Specify a length of 1 to return this summary as a delimited string, or 2 to return this summary as a $LIST structure. If the length parameter is omitted, the default is 1.

- When mode is -5, do not specify a length parameter.

### Notes

**$VIEW Usage Restricted**

The $VIEW function is a restricted system capability. It is a protected command because the invoked code is located in the CACHESYS database. For further details, refer to the “CACHESYS Special Capabilities” in the Assets and Resources chapter of the Caché Security Administration Guide.

**Process Summary Information**

When offset is -1, you can use mode -3 to return summary information from the current process address space as a ^ delimited string, as shown in this example:
You can also return the same information as a $LIST structure, as follows:

```objectscript
SET infolist = $VIEW(-1,-3,2)
ZWRITE infolist
```

To return summary information from the address space of a specified process, provide the Process ID (pid) for that process as a positive integer for the mode argument, as shown in this example:

```objectscript
SET pid=$PIECE($IO,"",4)
WRITE $VIEW(-1,pid,1)
```

The summary information return value is in the following format:

```
pid^mode^dev^mem^dir^rou^stat^prio^uic^loc^blk^^^defns^lic^jbstat^mempeak^roles^loginroles
```

The fields are defined as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pid</strong></td>
<td>The process ID. See the Pid property of the SYS.Process class.</td>
</tr>
<tr>
<td><strong>mode</strong></td>
<td>* if in at the Terminal prompt. + or – if the job is part of a callin connection. Omitted for daemons.</td>
</tr>
<tr>
<td><strong>dev</strong></td>
<td>Current open device(s), returned as a comma-separated list. The current device (the $IO device) is indicated by an asterisk (*) suffix. See the OpenDevices property of the SYS.Process class. Note that the dev value includes a trailing comma, the OpenDevices value does not.</td>
</tr>
<tr>
<td><strong>mem</strong></td>
<td>Memory in use in the process partition (in KBs), if the process is not a daemon. Similar to but not identical to the MemoryUsed property of the SYS.Process class.</td>
</tr>
<tr>
<td><strong>dir</strong></td>
<td>Default directory.</td>
</tr>
<tr>
<td><strong>rou</strong></td>
<td>Routine name.</td>
</tr>
<tr>
<td><strong>stat</strong></td>
<td>A comma-separated pair of integer counts, bol,gcnt, where bol is the beginning of line token, specifying the number of lines executed, and gcnt is the global count, specifying the total number of FOR loops and XECUTE commands performed.</td>
</tr>
<tr>
<td><strong>prio</strong></td>
<td>User's current base priority. See the Priority property of the SYS.Process class.</td>
</tr>
<tr>
<td><strong>uic</strong></td>
<td>Obsolete, defaults to 0,0.</td>
</tr>
<tr>
<td><strong>loc</strong></td>
<td>Location, for daemon processes only.</td>
</tr>
<tr>
<td><strong>blk</strong></td>
<td>Number of 2K blocks that can be used for buddy block queues. This is the maximum size of user memory space (also known as partition space). See the MemoryAllocated property of the SYS.Process class.</td>
</tr>
<tr>
<td><strong>defns</strong></td>
<td>Default namespace. See the NameSpace property of the SYS.Process class.</td>
</tr>
<tr>
<td><strong>lic</strong></td>
<td>License bits.</td>
</tr>
<tr>
<td><strong>jbstat</strong></td>
<td>Job status, specified as high,low representing the high and low order bits. Refer to $ZJOB special variable for details.</td>
</tr>
<tr>
<td><strong>mempeak</strong></td>
<td>Peak memory usage for the process, in kilobytes. This value is approximate to the nearest 64K. See the MemoryPeak property of the SYS.Process class.</td>
</tr>
<tr>
<td><strong>roles</strong></td>
<td>The roles that the process currently has, returned as a comma-separated list. Same as $ROLES value. See the Roles property of the SYS.Process class.</td>
</tr>
</tbody>
</table>
loginroles

The roles that the process had when it was initiated, returned as a comma-separated list. See the LoginRoles property of the SYS.Process class.

Using Mode -5

If the current block in the view buffer contains part of a global, specifying -5 for mode returns the global references and the values contained in the block. The length parameter is not valid for a mode of -5.

With a mode of -5, the offset value specifies the number of global nodes in the block, rather than a byte offset from the base address. Odd values return full global references and even values return pointers or data.

For example, to return the full global reference of the nth node in the view buffer, specify n*2-1 for offset. To return the value of the nth node, specify n*2. To return the global reference of the last node in the block, specify -1 for the offset value.

$VIEW returns the nodes in collating sequence (that is, numeric). This is the same sequence that the $ORDER function uses. By writing code that expects this sequence, you can quickly perform a sequential scan through a global in the view buffer. (Several of the Caché utilities use this technique.) $VIEW returns a null string ("") if the offset specifies a location beyond the last node in the view buffer. Be sure to include a test for this value in your code.

If the current block is a pointer block, the values returned are Caché block numbers, which are pointers. If the block is a data block, the values returned are the data values associated with the nodes.

If $VIEW issues a <DATABASE> or <FUNCTION> error, it means that the information in the block is neither a valid global reference nor valid data.

The following example shows generalized code for examining the contents of the view buffer. The code first opens the view buffer and prompts for the number of the block to be read in. The FOR loop then cycles through all the offsets in the current block. The $VIEW function uses a mode of -5 to return the value at each offset. The WRITE commands output the resulting offset-value pairs.

When the end of the block is reached, $VIEW returns a null string (""). The IF command tests for this value and writes out the “End of block” message. The QUIT command then terminates the loop and control returns to the prompt so the user can read in another block.

Start OPEN 63
WRITE !,"Opening view buffer."
READ !,"Number of block to read in: ",block QUIT:block="
VIEW block
FOR i=1:1 {
SET x=$VIEW(i,-5)
IF x="",i#2 {
WRITE !,"Offset = ",i
WRITE !,"Value = ",x
} GOTO Start+2
CLOSE 63
QUIT
}

For a global block, typical output produced by this routine might appear as follows:
Opening view buffer.
Number of block to read in: 3720
Offset = 1
Value = ^client(5)
Offset = 2
Value = John Jones
Offset = 3
Value = ^client(5,1)
Offset = 4
Value = 23 Bay Rd./Boston/MA 02049
Offset = 126
Value = ^client(18,1,1)
Offset = 127
Value = Checking/45673/1248.00
End of block: 3720
Number of block to read in:

**Reverse Order Byte Values (Big-Endian only)**

On big-endian systems, you can return byte values in reverse order by using a letter “O” suffix as part of the `length` parameter. When you specify the letter O in `length`, `$VIEW` returns a byte value in reverse order. (The `length` value must be enclosed in double quotes.) This is shown in the following example:

```objectscript
USE IO
FOR Z=0:0 {
  WRITE *,-6
  SET NEXTBN=$VIEW(LINKA,0,"3O")
  QUIT:NEXTBN=0 }
```

In the example above, the `length` parameter of `$VIEW` is “3O” (3 and the letter O). When run on a big-endian system, this specifies a length of the next three (3) bytes in reverse order (O). Thus, `$VIEW` starts at a position in memory (the view buffer—as indicated by a `mode` of 0) and returns the highest byte, the second highest byte, and the third highest byte.

On little-endian systems, the letter “O” is a no-op. A `length` value of “3O” is the same as a `length` value of “3”.

You can use the `IsBigEndian()` class method to determine which bit ordering is used on your operating system platform: 1=big-endian bit order; 0=little-endian bit order.

```objectscript
WRITE $SYSTEM.Version.IsBigEndian()
```

See Also

- **VIEW** command
- **JOB** command
$WASCII

Returns the numeric code corresponding to a character, recognizing surrogate pairs.

$WASCII(expression,position)
$WA(expression,position)

Parameters

| expression | The character to be converted. |
| position   | **Optional** — The position of a character within a character string, counting from 1. The default is 1. |

Description

$WASCII returns the character code value for a single character specified in expression. $WASCII recognizes a surrogate pair as a single character. The returned value is a positive integer.

The expression parameter may evaluate to a single character or to a string of characters. If expression evaluates to a string of characters, you can include the optional position parameter to indicate which character you want to convert. The position counts a surrogate pair as a single character. You can use the $WISWIDE function to determine if a string contains a surrogate pair.

A surrogate pair is a pair of 16-bit Caché character elements that together encode a single Unicode character. Surrogate pairs are used to represent certain ideographs which are used in Chinese, Japanese kanji, and Korean hanja. (Most commonly-used Chinese, kanji, and hanja characters are represented by standard 16-bit Unicode encodings.) Surrogate pairs provide Caché support for the Japanese JIS X0213:2004 (JIS2004) encoding standard and the Chinese GB18030 encoding standard.

A surrogate pair consists of high-order 16-bit character element in the hexadecimal range D800 through DBFF, and a low-order 16-bit character element in the hexadecimal range DC00 through DFFF.

The $WASCII function recognizes a surrogate pair as a single character. The $ASCII function treats a surrogate pair as two characters. In all other aspects, $WASCII and $ASCII are functionally identical. However, because $ASCII is generally faster than $WASCII, $ASCII is preferable for all cases where a surrogate pair is not likely to be encountered. For further details on character to numeric code conversion, refer to the $ASCII function.

Examples

The following example shows $WASCII returning the Unicode value for a surrogate pair:

```ObjectScript
IF $SYSTEM.Version.IsUnicode()  {
  SET hipart=$CHAR($ZHEX("D806"))
  SET lopart=$CHAR($ZHEX("DC06"))
  WRITE !,$ASCII(hipart)," = high-order value"
  WRITE !,$ASCII(lopart)," = low-order value"
  SET spair=hipart_lopart /* surrogate pair */
  SET xpair=hipart_hipart /* NOT a surrogate pair */
  WRITE !,$WASCII(spair)," = surrogate pair value"
  WRITE !,$WASCII(xpair)," = Not a surrogate pair"
} ELSE {WRITE "This example requires a Unicode installation of Caché"}
```

The following example compares $WASCII and $ASCII return values for a surrogate pair:
The following example shows the effects on position counting of surrogate pairs. It returns both the $WASCII and $ASCII values for each position. $WASCII counts a surrogate pair as one position; $ASCII counts a surrogate pair as two positions:

IF $SYSTEM.Version.IsUnicode()  {
SET hipart=$CHAR($ZHEX("D806"))
SET lopart=$CHAR($ZHEX("DC06"))
WRITE !,$ASCII(hipart)," = high-order value"
WRITE !,$ASCII(lopart)," = low-order value"
SET spair=hipart_lopart /* surrogate pair */
WRITE !,$ASCII(spair)," = $ASCII value for surrogate pair"
WRITE !,$WASCII(spair)," = $WASCII value for surrogate pair"
}
ELSE {WRITE "This example requires a Unicode installation of Caché"}

See Also

- $ASCII function
- $WCHAR function
- $WEXTRACT function
- $WFIND function
- $WISWIDE function
- $WLENGTH function
- $WREVERSE function
$WCHAR

Returns the character corresponding to a numeric code, recognizing surrogate pairs.

$WCHAR(expression,...)
$WC(expression,...)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>The integer value to be converted.</td>
</tr>
</tbody>
</table>

**Description**

$WCHAR returns the character(s) corresponding to a code value(s) specified in expression. Decimal values of 65535 (hex FFFF) and smaller are processed identically by $CHAR and $WCHAR. Values from 65536 (hex 10000) through 1114111 (hex 10FFFF) are used to represent Unicode surrogate pairs; these characters can be returned using $WCHAR.

If expression contains a comma-separated list of code values, $WCHAR returns the corresponding characters as a string. $WCHAR recognizes a surrogate pair as a single character. You can use the $WISWIDE function to determine if a string contains a surrogate pair.

A surrogate pair is a pair of 16-bit Caché character elements that together encode a single Unicode character. Surrogate pairs are used to represent certain ideographs which are used in Chinese, Japanese kanji, and Korean hanja. (Most commonly-used Chinese, kanji, and hanja characters are represented by standard 16-bit Unicode encodings.) Surrogate pairs provide Caché support for the Japanese JIS X0213:2004 (JIS2004) encoding standard and the Chinese GB18030 encoding standard.

A surrogate pair consists of high-order 16-bit character element in the hexadecimal range D800 through DBFF, and a low-order 16-bit character element in the hexadecimal range DC00 through DFFF.

The $WCHAR function treats a surrogate pair as a single character. The $CHAR function treats a surrogate pair as two characters. In all other aspects, $WCHAR and $CHAR are functionally identical. However, because $CHAR is generally faster than $WCHAR, $CHAR is preferable for all cases where a surrogate pair is not likely to be encountered.

For further details on numeric code to character conversion, refer to the $CHAR function.

**See Also**

- $CHAR function
- $WASCII function
- $WEXTRACT function
- $WFIND function
- $WISWIDE function
- $WLENGTH function
- $WREVERSE function
$WEXTRACT

Extracts a substring from a character string by position, or replaces a substring by position, recognizing surrogate pairs.

$WEXTRACT(string,from,to)
$WE(string,from,to)

SET $WEXTRACT(string,from,to)=value
SET $WE(string,from,to)=value

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The target string in which substrings are identified. Specify string as an expression that evaluates to a quoted string or a numeric value. In SET $WEXTRACT syntax, string must be a variable or a multi-dimensional property.</td>
</tr>
</tbody>
</table>
| from      | Optional — The starting position within the target string. Characters are counted from 1. A surrogate pair is counted as a single character. Permitted values are n (a positive integer specifying the start position as a character count from the beginning of string), * (specifying the last character in string), and *-n (offset integer count of characters backwards from end of string). SET $WEXTRACT syntax also supports *+n (offset integer count of characters to append beyond the end of string). If not specified, the default is 1. Different values are used for the two-parameter form $WEXTRACT(string,from), and the three-parameter form $WEXTRACT(string,from,to):
    - Without to: Specifies a single character. To count from the beginning of string, specify an expression that evaluates to a positive integer (counting from 1); a zero (0) or negative number returns the empty string. To count from the end of string specify *, or *-n. If from is omitted it defaults to 1.
    - With to: Specifies the start of a range of characters. To count from the beginning of string, specify an expression that evaluates to a positive integer (counting from 1). A zero (0) or negative number evaluates as 1. To count from the end of string specify *, or *-n. |
| to        | Optional — Specifies the end position (inclusive) for a range of characters. Must be used with from. Permitted values are n (a positive integer equal to or larger than from that specifies the end position as a character count from the beginning of string), * (specifying the last character in string), and *-n (offset integer count of characters backwards from end of string). A surrogate pair is counted as a single character. You can specify a to value that is beyond the end of the string.
    - SET $WEXTRACT syntax also supports *+n (offset integer count of the end of a range of characters to append beyond the end of string). |

Description

$WEXTRACT identifies a substring within string by position, either counting characters from the beginning of string or counting characters by offset from the end of string. A substring can be a single character or a range of characters.

$WEXTRACT recognizes a surrogate pair as a single character.

$WEXTRACT can be used in two ways:

- To return a substring from string. This uses the $WEXTRACT(string,from,to) syntax.
- To replace a substring within string. The replacement substring may be the same length, longer, or shorter than the original substring. This uses the SET $WEXTRACT(string,from,to)=value syntax.
$WEXTRACT and $EXTRACT are functionally identical, except for the handling of surrogate pairs.

**Surrogate Pairs**

The $WEXTRACT from and to parameters count a surrogate pair as a single character. You can use the $WISWIDE function to determine if a string contains a surrogate pair.

A surrogate pair is a pair of 16-bit Caché character elements that together encode a single Unicode character. Surrogate pairs are used to represent certain ideographs which are used in Chinese, Japanese kanji, and Korean hanja. (Most commonly-used Chinese, kanji, and hanja characters are represented by standard 16-bit Unicode encodings.) Surrogate pairs provide Caché support for the Japanese JIS X0213:2004 (JIS2004) encoding standard and the Chinese GB18030 encoding standard.

A surrogate pair consists of high-order 16-bit character element in the hexadecimal range D800 through DBFF, and a low-order 16-bit character element in the hexadecimal range DC00 through DFFF.

The $WEXTRACT function treats a surrogate pair as a single character. The $EXTRACT function treats a surrogate pair as two characters. If a string contains no surrogate pairs, either $WEXTRACT and $EXTRACT can be used and return the same value. However, because $EXTRACT is generally faster than $WEXTRACT, $EXTRACT is preferable for all cases where a surrogate pair is not likely to be encountered. For further details on extracting a substring, refer to the $EXTRACT function.

**Returning a Substring**

$WEXTRACT returns a substring by character position from string. The nature of this substring extraction depends on the parameters used:

- **$WEXTRACT(string)** extracts the first character in the string.
- **$WEXTRACT(string,from)** extracts a single character in the position specified by from. The from value can be an integer count of characters from the beginning of the string, an asterisk specifying the last character of the string, or an asterisk with a negative integer specifying a character count backwards from the end of the string.

The following example extracts single letters from a string containing a surrogate pair. Note that $LENGTH counts a surrogate pair as two characters, but $WEXTRACT counts a surrogate pair as a single character:

```objectscript
IF $SYSTEM.Version.IsUnicode()  {
  SET hipart=$CHAR($ZHEX("D806"))
  SET lopart=$CHAR($ZHEX("DC06"))
  SET spair=hipart_lopart /* surrogate pair */
  WRITE "$length of surrogate pair ",$LENGTH(spair),!
  SET mystr="AB"_spair_"DEFG"
  WRITE "$WEXTRACT(mystr,4)    // "D" the 4th character"!
  WRITE "$WEXTRACT(mystr,*-4) // "G" the last character"!
  WRITE "$WEXTRACT(mystr,*-5) // "B" the offset 5 character from end"!
  WRITE "$WEXTRACT(mystr,*-0) // "G" the last character by 0 offset"!
  ELSE {WRITE "This example requires a Unicode installation of Caché"
}
```

- **$WEXTRACT(string,from,to)** extracts the range of characters starting with the from position and ending with the to position (inclusive). The following $WEXTRACT functions both return the string “Alabama”, counting surrogate pairs as single characters:

```objectscript
IF $SYSTEM.Version.IsUnicode()  {
  SET hipart=$CHAR($ZHEX("D806"))
  SET lopart=$CHAR($ZHEX("DC06"))
  SET spair=hipart_lopart /* surrogate pair */
  SET var2=spair_"XXX"_spair_"Alabama"_spair
  WRITE "$WEXTRACT(var2,6,12)"
  WRITE "$WEXTRACT(var2,*-7,*-1)"
  ELSE {WRITE "This example requires a Unicode installation of Caché"
}
```

If the from and to positions are the same, $WEXTRACT returns a single character. If the to position is closer to the beginning of the string than the from position, $WEXTRACT returns the null string.
Replacing a Substring

You can use `$WEXTRACT` with the `SET` command to replace a specified character or range of characters with another value. You can also use it to append characters to the end of a string. `SET $WEXTRACT` counts a surrogate pair as a single character.

When `$WEXTRACT` is used with `SET` on the left hand side of the equals sign, `string` can be a valid variable name. If the variable does not exist, `SET $WEXTRACT` defines it. The `string` parameter can also be a multidimensional property reference; it cannot be a non-multidimensional object property. Attempting to use `SET $WEXTRACT` on a non-multidimensional object property results in an `<OBJECT DISPATCH>` error.

You cannot use `SET (a,b,c,...)=value` syntax with `$WEXTRACT` (or `$EXTRACT`, `$SPICE`, or `$LIST`) on the left of the equals sign, if the function uses relative offset syntax: `*` representing the end of a string and `*-n` or `*+n` representing relative offset from the end of the string. You must instead use `SET a=value,b=value,c=value,...` syntax.

For further details on replacing a substring, refer to the `$EXTRACT` function.

Examples

The following example shows the two-parameter form of `$WEXTRACT` returning the Unicode value for a surrogate pair:

```OBJECTSCRIPT
IF $SYSTEM.Version.IsUnicode()  {
    SET hipart=$CHAR($ZHEX("D806"))
    SET lopart=$CHAR($ZHEX("DC06"))
    SET spair=hipart_lopart /* surrogate pair */
    SET x="ABC"_spair_"DEFGHIJK"
    WRITE !,"$EXTRACT character ",ZZDUMP $EXTRACT(x,4)
    WRITE !,"$WEXTRACT character ",ZZDUMP $WEXTRACT(x,4)
} ELSE {WRITE "This example requires a Unicode installation of Caché"}
```

The following example shows the three-parameter form of `$WEXTRACT` including a surrogate pair in a substring range:

```OBJECTSCRIPT
IF $SYSTEM.Version.IsUnicode()  {
    SET hipart=$CHAR($ZHEX("D806"))
    SET lopart=$CHAR($ZHEX("DC06"))
    SET spair=hipart_lopart /* surrogate pair */
    SET x="ABC"_spair_"DEFGHIJK"
    WRITE !,"$EXTRACT two characters ",ZZDUMP $EXTRACT(x,3,4)
    WRITE !,"$WEXTRACT two characters ",ZZDUMP $WEXTRACT(x,3,4)
} ELSE {WRITE "This example requires a Unicode installation of Caché"}
```

See Also

- `$EXTRACT` function
- `$WASCII` function
- `$WCHAR` function
- `$WFIND` function
- `$WISWIDE` function
- `$WLENGTH` function
- `$WREVERSE` function
$WFIND

Finds a substring by value and returns an integer specifying its end position in the string, recognizing surrogate pairs.

$WFIND(string,substring,position)
$WF/find(string,substring,position)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The target string that is to be searched. It can be a variable name, a numeric value, a string literal, or any valid ObjectScript expression that resolves to a string.</td>
</tr>
<tr>
<td>substring</td>
<td>The substring that is to be searched for. It can be a variable name, a numeric value, a string literal, or any valid ObjectScript expression that resolves to a string.</td>
</tr>
<tr>
<td>position</td>
<td>Optional — A position within the target string at which to start the search. It must be a positive integer.</td>
</tr>
</tbody>
</table>

Description

$WFIND returns an integer specifying the end position of a substring within a string. In calculating position, it counts each surrogate pair as a single character. $WFIND is functionally identical to $FIND, except that $WFIND recognizes surrogate pairs. It counts a surrogate pair as a single character. You can use the $WISWIDE function to determine if a string contains a surrogate pair.

A surrogate pair is a pair of 16-bit Caché character elements that together encode a single Unicode character. Surrogate pairs are used to represent certain ideographs which are used in Chinese, Japanese kanji, and Korean hanja. (Most commonly-used Chinese, kanji, and hanja characters are represented by standard 16-bit Unicode encodings.) Surrogate pairs provide Caché support for the Japanese JIS X0213:2004 (JIS2004) encoding standard and the Chinese GB18030 encoding standard.

A surrogate pair consists of high-order 16-bit character element in the hexadecimal range D800 through DBFF, and a low-order 16-bit character element in the hexadecimal range DC00 through DFFF.

The $WFIND function counts a surrogate pair as a single character. The $FIND function counts a surrogate pair as two characters. In all other aspects, $WFIND and $FIND are functionally identical. However, because $FIND is generally faster than $WFIND, $FIND is preferable for all cases where a surrogate pair is not likely to be encountered.

For further details on finding a substring, refer to the $FIND function.

Examples

The following example shows how $WFIND counts a surrogate pair as a single character in the return value:

```ObjectScript
IF $SYSTEM.Version.IsUnicode()  {
  SET spair=$CHAR($ZHEX("D806"),$ZHEX("DC06"))
  SET str="ABC"_spair_"DEF"
  WRITE !,$FIND(str,"DE")," $FIND location in string"
  WRITE !,$WFIND(str,"DE")," $WFIND location in string"
} ELSE {WRITE "This example requires a Unicode installation of Caché"}
```

The following example shows how $WFIND counts a surrogate pair as a single character in the position parameter:

```ObjectScript
IF $SYSTEM.Version.IsUnicode()  {
  SET spair=$CHAR($ZHEX("D806"),$ZHEX("DC06"))
  SET str="ABC"_spair_"DEF"
  WRITE !,$FIND(str,"DE",6)," $FIND location in string"
  WRITE !,$WFIND(str,"DE",6)," $WFIND location in string"
} ELSE {WRITE "This example requires a Unicode installation of Caché"}
```
See Also

- $FIND function
- $WASCI1 function
- $WCHAR function
- $WEXTRACT function
- $WISWISE function
- $WLENGTH function
- $WREVERSE function
$WISWIDE

Returns a flag indicating whether a string contains surrogate pairs.

$WISWIDE(string)

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A string or expression that evaluates to a string.</td>
</tr>
</tbody>
</table>

Description

$WISWIDE returns a boolean value indicating whether string contains surrogate pairs. 0=string does not contain any surrogate pairs. 1=string contains one or more surrogate pairs.

A surrogate pair is a pair of 16-bit Caché character elements that together encode a single Unicode character. Surrogate pairs are used to represent certain ideographs which are used in Chinese, Japanese kanji, and Korean hanja. (Most commonly-used Chinese, kanji, and hanja characters are represented by standard 16-bit Unicode encodings.) Surrogate pairs provide Caché support for the Japanese JIS X0213:2004 (JIS2004) encoding standard and the Chinese GB18030 encoding standard.

A surrogate pair consists of high-order 16-bit character element in the hexadecimal range D800 through DBFF, and a low-order 16-bit character element in the hexadecimal range DC00 through DFFF.

Example

The following example shows $WISWIDE returning a boolean for a surrogate pair:

```objectscript
IF $SYSTEM.Version.IsUnicode()  {
    SET spair=$CHAR($ZHEX("D806"),$ZHEX("DC06")) /* surrogate pair */
    SET xpair=$CHAR($ZHEX("DC06"),$ZHEX("D806")) /* NOT a surrogate pair */
    SET str="AB"_spair_"CD"
    WRITE !,$WISWIDE(str)," = surrogate pair(s) in string?"  
    SET xstr="AB"_xpair_"CD"
    WRITE !,$WISWIDE(xstr)," = surrogate pair(s) in string?"
}  
ELSE {WRITE "This example requires a Unicode installation of Caché"}
```

See Also

- $WASCII function
- $WCHAR function
- $WEXTRACT function
- $WFIND function
- $WLENGTH function
- $WREVERSE function
$WLENGTH

Returns the number of characters in a string, recognizing surrogate pairs.

$WLENGTH(string)
$WL(string)

Parameter

| string | A string or expression that evaluates to a string. |

Description

$WLENGTH returns the number of characters in string. $WLENGTH is functionally identical to $LENGTH, except that $WLENGTH recognizes surrogate pairs. It counts a surrogate pair as a single character. You can use the $WISWIDE function to determine if a string contains a surrogate pair.

A surrogate pair is a pair of 16-bit Caché character elements that together encode a single Unicode character. Surrogate pairs are used to represent certain ideographs which are used in Chinese, Japanese kanji, and Korean hanja. (Most commonly-used Chinese, kanji, and hanja characters are represented by standard 16-bit Unicode encodings.) Surrogate pairs provide Caché support for the Japanese JIS X0213:2004 (JIS2004) encoding standard and the Chinese GB18030 encoding standard.

A surrogate pair consists of high-order 16-bit character element in the hexadecimal range D800 through DBFF, and a low-order 16-bit character element in the hexadecimal range DC00 through DFFF.

The $WLENGTH function counts a surrogate pair as a single character. The $LENGTH function counts a surrogate pair as two characters. In all other aspects, $WLENGTH and $LENGTH are functionally identical. However, because $LENGTH is generally faster than $WLENGTH, $LENGTH is preferable for all cases where a surrogate pair is not likely to be encountered.

For further details on string length, refer to the $LENGTH function.

Example

The following example shows how $WLENGTH counts a surrogate pair as a single character:

```
IF $SYSTEM.Version.IsUnicode()  {
    SET spair=$CHAR($ZHEX("D806"),$ZHEX("DC06"))
    SET str="AB"_spair_"CD"
    WRITE !,$LENGTH(str)," $LENGTH characters in string"
    WRITE !,$WLENGTH(str)," $WLENGTH characters in string"
}  ELSE {WRITE "This example requires a Unicode installation of Caché"}
```

See Also

- $LENGTH function
- $WASCII function
- $WCHAR function
- $WEXTRACT function
- $WFIND function
- $WISWIDE function
- $WREVERSE function
$WREVERSE

Returns the characters in a string in reverse order, recognizing surrogate pairs.

$WREVERSE(string)
$WRE(string)

Parameter

| string | A string or expression that evaluates to a string. |

Description

$WREVERSE returns the characters in string in reverse order. $WREVERSE is functionally identical to $REVERSE, except that $WREVERSE recognizes surrogate pairs. You can use the $WISWIDE function to determine if a string contains a surrogate pair.

A surrogate pair is a pair of 16-bit Caché character elements that together encode a single Unicode character. Surrogate pairs are used to represent certain ideographs which are used in Chinese, Japanese kanji, and Korean hanja. (Most commonly-used Chinese, kanji, and hanja characters are represented by standard 16-bit Unicode encodings.) Surrogate pairs provide Caché support for the Japanese JIS X0213:2004 (JIS2004) encoding standard and the Chinese GB18030 encoding standard.

A surrogate pair consists of high-order 16-bit character element in the hexadecimal range D800 through DBFF, and a low-order 16-bit character element in the hexadecimal range DC00 through DFFF.

The $WREVERSE function counts a surrogate pair as a single character. The $REVERSE function treats a surrogate pair as two characters. In all other aspects, $WREVERSE and $REVERSE are functionally identical. However, because $REVERSE is generally faster than $WREVERSE, $REVERSE is preferable for all cases where a surrogate pair is not likely to be encountered.

For further details on reversing strings, refer to the $REVERSE function.

Example

The following example shows how $WREVERSE treats a surrogate pair as a single character:

```ObjectScript
IF $SYSTEM.Version.IsUnicode()  {
  SET spair=$CHAR($ZHEX("D806"),$ZHEX("DC06"))
  SET str="AB"_spair_"CD"
  WRITE !,"String before reversing:"
  ZZDUMP str
  SET wrev=$WREVERSE(str)
  WRITE !,"$WREVERSE did not reverse surrogate pair:"
  ZZDUMP wrev
  SET rev=$REVERSE(str)
  WRITE !,"$REVERSE reversed surrogate pair:"
  ZZDUMP rev
} OTHERWISE {
  WRITE "This example requires a Unicode installation of Caché"
}
```

See Also

- $REVERSE function
- $WASCII function
- $WCHAR function
- $WEXTRACT function
- $WFIND function
- $WISWIDE function
• $WLENGTH function
$XECUTE

Executes a specified command line.

$XECUTE(code, paramlist)

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>An expression that resolves to a valid ObjectScript command line, specified as a quoted string. A command line can contain one or more ObjectScript commands. The final command must be an argumented QUIT.</td>
</tr>
<tr>
<td>paramlist</td>
<td>Optional — A list of parameters to be passed to code. Multiple parameters are separated by commas.</td>
</tr>
</tbody>
</table>

### Description

The $XECUTE function allows you to execute user-written code as a function, supplying passed parameters and returning a value. The code parameter must evaluate to a quoted string containing one or more ObjectScript commands. The code execution must conclude with a QUIT command that returns an argument. Caché then returns this QUIT argument as the $XECUTE return code value.

You can use the paramlist argument to pass parameters to code. If you are passing parameters, there must be a formal parameter list at the beginning of code. Parameters are specified positionally. There must be at least as many formal parameters listed in code as there are actual parameters specified in paramlist.

You can use the CheckSyntax() method of the %Library.Routine class to perform syntax checking on code.

Each invocation of $XECUTE places a new context frame on the call stack for your process. The $STACK special variable contains the current number of context frames on the call stack.

The $XECUTE function performs substantially the same operation as the XECUTE command, with the following differences: The $XECUTE function does not support postconditionals or the use of multiple command line arguments. The $XECUTE function requires every execution path to end with an argumented QUIT; the XECUTE command neither requires a QUIT nor permits an argumented QUIT.

### Parameters

**code**

An expression that evaluates to a valid ObjectScript command line, specified as a quoted string. The code string must not contain a tab character at the beginning or a <Return> at the end. The string can be no longer than a valid ObjectScript program line. The code string must contain a QUIT command that returns an argument at the conclusion of each possible execution path.

If $XECUTE passes parameters to code, the code string must begin with a formal parameter list. A formal parameter list is enclosed in parentheses; within the parentheses, parameters are separated by commas.

**paramlist**

A list of parameters to pass to code, specified as a comma-separated list. Each parameter in paramlist must correspond to a formal parameter within the code string. The number of parameters in paramlist may be less than or equal to the number of formal parameters listed in code.

You can use a dot prefix to pass a parameter by reference. This is useful for passing a value out from code. An example is provided below. For further details, refer to Passing by Reference in the “User-defined Code” chapter of Using Caché ObjectScript.
Examples

In the following example, the $XECUTE function executes the command line specified in cmdline. It passes two parameters, num1 and num2 to this command line.

```objectscript
SET cmd="(dvnd,dvsr) IF dvsr=0 (QUIT 99) ELSE (SET ^testnum=dvnd/dvsr QUIT 0)"
SET rtn=$XECUTE(cmd,num1,num2)
IF rtn=99
    {WRITE !,"Division by zero. ^testnum not set")
ELSE
    {WRITE !,"global ^testnum set to",^testnum}
```

The following example uses passing by reference (.y) to pass a local variable value from the code to the invoking context.

```objectscript
CubeIt
SET x=7
SET rtn=$XECUTE("(in,out) SET out=in*in*in QUIT 0",x,.y)
IF rtn=0 {WRITE !,x," cubed is ",y}
ELSE {WRITE !,"Error code=",SQLCODE}
```

The following example shows how $XECUTE increments the $STACK special variable. This example either writes the $STACK value from within $XECUTE, or has $XECUTE invoke the XECUTE command, which writes the $STACK value:

```objectscript
StackIt
SET stackit=$RANDOM(3)
IF stackit=0 {GOTO StackIt}
WRITE "initial stack level ",$STACK,!
SET cmd="(stackit) IF stackit=1 {WRITE "stack is ",$STACK,!
ELSEIF stackit=2 {WRITE "stack is ",$STACK XECUTE "WRITE "\" stack is \"",$STACK,!
ELSE { QUIT 0}"
```

See Also

- DO command
- XECUTE command
- QUIT command
- $STACK special variable
$ZABS

Absolute value function.

$ZABS (n)

Parameter

| n   | Any number. |

Description

$ZABS returns the absolute value of n.

Parameter

n

Any number. Can be specified as a value, a variable, or an expression. The expression is evaluated, and the result converted to a positive value. Multiple plus and minus signs are permitted. Leading and trailing zeros are deleted.

For evaluation of mixed numeric strings and non-numeric strings, refer to Strings As Numbers in Using Caché ObjectScript.

Example

The following example returns the absolute value of the number you supply:

```
READ "Input a number: ",num
SET abs=$ZABS(num)
WRITE "The absolute value of ",num," is ",abs
```

See Also

- Operators in Using Caché ObjectScript
**$ZARCCOS**

Inverse (arc) cosine function.

### $ZARCCOS (n)

**Parameter**

\[ n \]

A signed decimal number.

**Description**

$ZARCCOS$ returns the trigonometric inverse (arc) cosine of $n$. The result is given in radians (to 18 decimal places).

**Parameter**

\[ n \]

Signed decimal number ranging from 1 to -1 (inclusive). It can be specified as a value, a variable, or an expression. Numbers outside the range generate an <ILLEGAL VALUE> error. A non-numeric string is evaluated as 0.

The following are arc cosine values returned by $ZARCCOS$:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>returns 0</td>
</tr>
<tr>
<td>0</td>
<td>returns 1.570796326794896619</td>
</tr>
<tr>
<td>-1</td>
<td>returns π (3.141592653589793238)</td>
</tr>
</tbody>
</table>

**Examples**

The following example permits you to compare the arc cosine and the arc sine of a number:

```
READ "Input a number: ",num
IF num>1 { WRITE !,"ILLEGAL VALUE: number too big" }
ELSEIF num<-1 { WRITE !,"ILLEGAL VALUE: number too small" }
ELSE {
  WRITE !,"the arc cosine is: ",$ZARCCOS(num)
  WRITE !,"the arc sine is: ",$ZARCSIN(num)
}
QUIT
```

The following example compares the results from Caché fractional numbers ($DECIMAL$ numbers) and $DOUBLE$ numbers. In both cases, the arc cosine of 1 is exactly 0:

```
WRITE !,"the arc cosine is: ",$ZARCCOS(0.0)
WRITE !,"the arc cosine is: ",$ZARCCOS($DOUBLE(0.0))
WRITE !,"the arc cosine is: ",$ZARCCOS(1.0)
WRITE !,"the arc cosine is: ",$ZARCCOS($DOUBLE(1.0))
WRITE !,"the arc cosine is: ",$ZARCCOS(-1.0)
WRITE !,"the arc cosine is: ",$ZARCCOS($DOUBLE(-1.0))
```

**See Also**

- $ZCOS$ function
- $ZPI$ special variable
$ZARCSIN

Inverse (arc) sine function.

$ZARCSIN(n)

Parameter

| n       | A signed decimal number. |

Description

$ZARCSIN returns the trigonometric inverse (arc) sine of $n. The result is given in radians.

Parameter

$n$

Signed decimal number ranging from 1 to -1 (inclusive). It can be specified as a value, a variable, or an expression. Numbers outside the range generate an <ILLEGAL VALUE> error. A non-numeric string is evaluated as 0.

The following are arc sine values returned by $ZARCSIN:

<table>
<thead>
<tr>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>returns 1.570796326794896619</td>
</tr>
<tr>
<td>0</td>
<td>returns 0</td>
</tr>
<tr>
<td>-1</td>
<td>returns -1.570796326794896619</td>
</tr>
</tbody>
</table>

Examples

The following example permits you to compare the arc sine and the arc cosine of a number:

```
READ "Input a number: ",num
IF num>1 { WRITE !,"ILLEGAL VALUE: number too big" }
ELSEIF num<-1 { WRITE !,"ILLEGAL VALUE: number too small" }
ELSE {
  WRITE !,"the arc sine is: ",$ZARCSIN(num)
  WRITE !,"the arc cosine is: ",$ZARCCOS(num)
}
QUIT
```

The following example compares the results from Caché fractional numbers ($DECIMAL numbers) and $DOUBLE numbers. In both cases, the arc sine of 0 is exactly 0:

```
WRITE !,"the arc sine is: ",$ZARCSIN(0.0)
WRITE !,"the arc sine is: ",$ZARCSIN($DOUBLE(0.0))
WRITE !,"the arc sine is: ",$ZARCSIN(1.0)
WRITE !,"the arc sine is: ",$ZARCSIN($DOUBLE(1.0))
WRITE !,"the arc sine is: ",$ZARCSIN(-1.0)
WRITE !,"the arc sine is: ",$ZARCSIN($DOUBLE(-1.0))
```

See Also

- $ZSIN function
- $ZPI special variable
$ZARCTAN

Inverse (arc) tangent function.

$ZARCTAN(n)

Parameter

<table>
<thead>
<tr>
<th>n</th>
<th>Any positive or negative number.</th>
</tr>
</thead>
</table>

Description

$ZARCTAN returns the trigonometric inverse (arc) tangent of n. Possible results range from 1.57079 (half of pi) through zero to –1.57079. The result is given in radians.

Parameter

n

Any positive or negative number. It can be specified as a value, a variable, or an expression. You can use the $ZPI special variable to specify pi. A non-numeric string is evaluated as 0.

The following are arc tangent values returned by $ZARCTAN:

<table>
<thead>
<tr>
<th>n</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>returns 1.107148717794090502</td>
</tr>
<tr>
<td>1</td>
<td>returns .7853981633974483098</td>
</tr>
<tr>
<td>0</td>
<td>returns 0</td>
</tr>
<tr>
<td>-1</td>
<td>returns -.7853981633974483098</td>
</tr>
</tbody>
</table>

Examples

The following example permits you to calculate the arc tangent of a number:

```
READ "Input a number: ",num
WRITE !,"the arc tangent is: ",$ZARCTAN(num)
QUIT
```

The following example compares the results from Caché fractional numbers ($DECIMAL numbers) and $DOUBLE numbers. In both cases, the arc tangent of pi/2 is a fractional number (not 1), but the arc tangent of 0 is 0:

```
WRITE !,"the arc tangent is: ",$ZARCTAN(0.0)
WRITE !,"the arc tangent is: ",$ZARCTAN($DOUBLE(0.0))
WRITE !,"the arc tangent is: ",$ZARCTAN($ZPI)
WRITE !,"the arc tangent is: ",$ZARCTAN($DOUBLE($ZPI))
WRITE !,"the arc tangent is: ",$ZARCTAN($ZPI/2)
WRITE !,"the arc tangent is: ",$ZARCTAN($DOUBLE($ZPI)/2)
```

See Also

- $ZTAN function
- $ZPI special variable
$ZBOOLEAN

Bitwise logical operation function.

$ZBOOLEAN(arg1,arg2,bit_op)
$ZB(arg1,arg2,bit_op)

Parameters

<table>
<thead>
<tr>
<th>arg1</th>
<th>The first argument. An integer or a string, or a variable or expression that resolve to an integer or string. All characters must have an ASCII value between 0 and 255. Cannot be a floating point number.</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg2</td>
<td>The second argument. An integer or a string, or a variable or expression that resolve to an integer or string. All characters must have an ASCII value between 0 and 255. Cannot be a floating point number.</td>
</tr>
<tr>
<td>bit_op</td>
<td>An integer indicating the operation to be performed (see table below.) Permitted values are 0 through 15, inclusive.</td>
</tr>
</tbody>
</table>

Description

$ZBOOLEAN performs the bitwise logical operation specified by bit_op on two arguments, arg1 and arg2. $ZBOOLEAN returns the results of the bitwise combination of arg1 and arg2, as specified by the bit_op value. You can view the results using the ZZDUMP command.

$ZBOOLEAN performs its operations on either character strings or numbers. For character strings, it performs logical AND and OR operations on each character in the string. For numbers, it performs a logical AND and OR operation on the entire number as a unit. To force the evaluation of a numeric string as a number, preface the string with a plus sign (+).

Note: $ZBOOLEAN does not support Unicode characters with a value larger than ASCII 255. To apply $ZBOOLEAN to a string of 16-bit Unicode characters, you must first use $ZWUNPACK, followed by $ZBOOLEAN, followed by $ZWPACK.

$ZBOOLEAN and $BITLOGIC use different data formats. The results of one cannot be used as input to the other.

The bitwise operations includes 16 possible Boolean combinations of arg1 and arg2. The following table lists these combinations.

<table>
<thead>
<tr>
<th>Bit Mask in bit_op</th>
<th>Operation Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>arg1 &amp; arg2 (logical AND)</td>
</tr>
<tr>
<td>2</td>
<td>arg1 &amp; ~arg2</td>
</tr>
<tr>
<td>3</td>
<td>arg1</td>
</tr>
<tr>
<td>4</td>
<td>~arg1 &amp; arg2</td>
</tr>
<tr>
<td>5</td>
<td>arg2</td>
</tr>
<tr>
<td>6</td>
<td>arg1 ^ arg2 (logical XOR (exclusive or))</td>
</tr>
<tr>
<td>7</td>
<td>arg1 ! arg2 (logical OR (inclusive or))</td>
</tr>
<tr>
<td>8</td>
<td>~(arg1 ! arg2)</td>
</tr>
<tr>
<td>Bit Mask in bit_op</td>
<td>Operation Performed</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>9</td>
<td>~(arg1 ^ arg2)</td>
</tr>
<tr>
<td>10</td>
<td>~arg2 (logical NOT)</td>
</tr>
<tr>
<td>11</td>
<td>arg1 ! ~arg2</td>
</tr>
<tr>
<td>12</td>
<td>~arg1 (logical NOT)</td>
</tr>
<tr>
<td>13</td>
<td>~arg1 ! arg2</td>
</tr>
<tr>
<td>14</td>
<td>~(arg1 &amp; arg2)</td>
</tr>
<tr>
<td>15</td>
<td>-1 (one's complement of 0)</td>
</tr>
</tbody>
</table>

Where:

& is logical AND

! is logical OR

~ is logical NOT

^ is exclusive OR

For further details, see Operators in Using Caché ObjectScript.

All $ZBOOLEAN operations parse both arg1 and arg2, including bit_op values 0, 3, 5, 10, 12, and 15.

The $ZBOOLEAN arg1 and arg2 parameters can resolve to one of the following types:

- An integer. A positive or negative whole decimal number of up to 18 digits. No characters other than the numbers 0–9 and, optionally, one or more leading plus and minus signs are permitted. Leading zeros are ignored.
- A string. Enclosed in quotation marks, a string of any length with any contents is permitted. Note that the string “123” and the integer 123 are not the same. A null string is permitted, but if arg2 is the null string, $ZBOOLEAN always returns the value of arg1, regardless of the bit_op value.
- A signed string. A string preceded by a plus or minus sign is parsed as an integer, regardless of the string’s contents. Signed strings are subject to the same length restriction as integers. A signed null string is equivalent to zero.

It is strongly recommended that arg1 and arg2 either both resolve to an integer or both resolve to a string. Generally, arg1 and arg2 should be the same data type; combining an integer and a string in a $ZBOOLEAN operation does not give a useful result in most cases.

**Parameters**

**arg1**

The first argument in the bitwise logical expression. For strings, the length of the returned value is always the same as the length of this argument.

**arg2**

The second argument in the bitwise logical expression.

**bit_op**

The bitwise logical operation to be performed, specified as a numeric code from 0 to 15, inclusive. Because this code is handled as a bit mask, a value of 16=0, 17=1, 18=2, etc.

The bit_op values 0 and 15 return a constant value, but they also evaluate the arguments. If arg1 is an integer (or signed string), bit_op 0 returns 0, and bit_op 15 returns -1 (the one’s complement of 0.) If arg1 is a string, bit_op 0 returns a low
value (hex 00) for each character in \( \text{arg1} \), and \( \text{bit\_op} \) 15 returns a high value (hex FF) for each character in \( \text{arg1} \). If \( \text{arg2} \) is the null string (""), both operations return the literal value of \( \text{arg1} \).

The \( \text{bit\_op} \) values 3, 5, 10 and 12 perform a logical operation on only one of the arguments, but they evaluate both arguments.

- \( \text{bit\_op}=3 \) always returns the value of \( \text{arg1} \), regardless of the value of \( \text{arg2} \).
- \( \text{bit\_op}=5 \) returns the value of \( \text{arg2} \) when the two arguments have the same data type. However if one argument is a string and the other argument is an integer (or signed string) results are unpredictable. If \( \text{arg2} \) is the null string \$ZBOOLEAN always returns the literal value of \( \text{arg1} \).
- \( \text{bit\_op}=10 \) returns the one’s complement value of \( \text{arg2} \) if both arguments are integers. If \( \text{arg1} \) is a string, the operation returns a high order character for each character in \( \text{arg1} \). If \( \text{arg2} \) is a string, and \( \text{arg1} \) is an integer, the bitwise operation is performed on the \( \text{arg2} \) string. If \( \text{arg2} \) is the null string \$ZBOOLEAN always returns the literal value of \( \text{arg1} \).
- \( \text{bit\_op}=12 \) returns the one’s complement value of \( \text{arg1} \) if it is an integer (or signed string), for any value of \( \text{arg2} \) except the null string. If \( \text{arg1} \) is a string, the operation returns the one’s complement (as a hex value) of each character in \( \text{arg1} \). If \( \text{arg2} \) is the null string \$ZBOOLEAN always returns the literal value of \( \text{arg1} \).

**Examples**

The following three examples all illustrate the same AND operation. These examples AND the ASCII values of lowercase letters with the ASCII value of the underscore character, resulting in the ASCII values of the corresponding uppercase letters.

```
WRITE \$ZBOOLEAN("abcd","_",1)
```

displays ABCD.

The lowercase "a" = [01100001] (ASCII decimal 97)
The underscore character "_" = [01011111] (ASCII decimal 95)
The uppercase "A" = [01000001] (ASCII decimal 65)

The following example performs the same AND operation as the previous example, but uses the ASCII decimal values of the arguments. The function \$ASCII("a") returns the decimal value 97 for the first argument:

```
WRITE \$ZBOOLEAN($ASCII("a"),95,1)
```

displays 65.

The following example performs the same AND operation, using a \$CHAR value as the second argument:

```
WRITE \$ZBOOLEAN("a",\$CHAR(95),1)
```

displays A.

The following examples illustrate logical OR:

```
WRITE \$ZBOOLEAN(1,0,7)
```

displays 1.

```
WRITE \$ZBOOLEAN(1,1,7)
```

displays 1.

```
WRITE \$ZBOOLEAN(2,1,7)
```

displays 3.

```
WRITE \$ZBOOLEAN(2,2,7)
```

displays 2.
WRITE $ZBOOLEAN(3,2,7)
displays 3.

The following logical OR examples demonstrate the difference between string comparisons and number comparisons:

WRITE $ZBOOLEAN(64,255,7)
compares the two values as numbers and displays 255.
WRITE $ZBOOLEAN("64","255",?)
compares the two values as strings and displays 65.
WRITE $ZBOOLEAN(+"64",+"255",?)
the plus signs force the comparison of the two values as numbers, and displays 255.

The following examples illustrate exclusive OR:

WRITE $ZBOOLEAN(1,0,6)
displays 1.
WRITE $ZBOOLEAN(1,1,6)
displays 0.
WRITE $ZBOOLEAN(2,1,6)
displays 3.
WRITE $ZBOOLEAN(2,2,6)
displays 0.
WRITE $ZBOOLEAN(3,2,6)
displays 1.
WRITE $ZBOOLEAN(64,255,6)
displays 191.

The following example shows a 4-byte entity with all bytes set to 1:

WRITE $ZBOOLEAN(5,1,15)
displays -1.

The following example will set x to 3 bytes with all bits set to 1:

SET x=$ZBOOLEAN("abc",0,15)
WRITE !,$LENGTH(x)
WRITE !,$ASCII(x,1)," ",$ASCII(x,2)," ",$ASCII(x,3)

The first WRITE displays 3; the second WRITE displays 255 255 255.

Notes

Integer Processing

Before $ZBOOLEAN performs the bitwise operation, it interprets each numeric value as either an 8-byte or a 4-byte signed binary value, depending on size. $ZBOOLEAN always interprets a numeric value as a series of bytes. The boolean operation uses these bytes as a string argument. The result type is the same as the type of arg1.
If either arg1 or arg2 is numeric and cannot be represented as an 8-byte signed integer (larger than 18 decimal digits), a
<FUNCTION> error results. If both arg1 and arg2 are numeric and one of them requires 8 bytes to be represented, then
both values are interpreted as 8-byte binary values.

After the previous transformations are complete, the given Boolean combination is applied bit by bit to arg1 and arg2 to
yield the result. The result returned is always the same length as arg1 (after the above transformations of numeric data). If
the length of arg2 is less than the length of arg1, then arg2 is repeatedly combined with successive substrings of arg1 in
left to right fashion.

$ZBOOLEAN always interprets the numeric value as a series of bytes in little-endian order, with the low-order byte first,
no matter what the native byte order of your machine is.

**Internal Structure of $ZBOOLEAN Values**

The following table lists the internal rules for $ZBOOLEAN. You do not need to understand these rules to use
$ZBOOLEAN; they are presented here for reference purposes only.

There are four possible states of any two bits being compared from within arg1 and arg2. The Boolean operation generates
a true result (=1) if and only if bit_op has the bit mask shown in the table.

<table>
<thead>
<tr>
<th>Bit in arg1</th>
<th>Bit in arg2</th>
<th>Bit Mask in bit_op Decimal</th>
<th>Bit Mask in bit_op Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
</tbody>
</table>

**EQV and IMP Logical Operators**

$ZBOOLEAN indirectly supports EQV and IMP logical operators. These logical operators are defined as follows:

- EQV is a logical equivalence between two expressions. It is represented by $ZBOOLEAN(arg1, arg2, 9). This is
  logically ~(arg1 ^ arg2) which is logically identical to ((~arg1) & (~arg2)) ! (arg1 & arg2).

- IMP is a logical implication between two expressions. It is represented by $ZBOOLEAN(arg1, arg2, 13). This is
  logically ~(arg1) ! arg2.

**See Also**

- ZZDUMP command
- Operators in Using Caché ObjectScript
$ZCONVERT

String conversion function.

$ZCONVERT(string, mode, trantable, handle)
$ZCVT(string, mode, trantable, handle)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string to convert, specified as a quoted string. This string can be specified as a value, a variable, or an expression.</td>
</tr>
<tr>
<td>mode</td>
<td>A letter code specifying the conversion mode, either the type of case conversion or input/output encoding. Specify mode as a quoted string.</td>
</tr>
<tr>
<td>trantable</td>
<td>Optional — The translation table to use, specified as either an integer or a quoted string,</td>
</tr>
<tr>
<td>handle</td>
<td>Optional — An unsubscribed local variable that holds a string value. Used for multiple invocations of $ZCONVERT. The handle parameter contains the remaining portion of string that could not be converted at the end of $ZCONVERT, and supplies this remaining portion to the next invocation of $ZCONVERT.</td>
</tr>
</tbody>
</table>

Description

$ZCONVERT converts a string from one form to another. The nature of the conversion depends on the parameters you use.

$ZCONVERT Returns a Converted String

$ZCONVERT(string, mode) returns string with the characters converted as specified by mode. The conversions are of two types:

- Case conversion
- Encoding translation

Case conversion changes the case of each letter character in the string. You can change all letter characters in a string to their lowercase, uppercase, or titlecase form. Characters that are already in the specified case, and characters with no case (usually any nonalphabetic character) in the string are passed through unchanged. To output a literal quote character (“”) within a string, input two quote characters (““”). For further case conversion options, including non-ASCII and customized case conversion, refer to the section on “System Classes for National Language Support” in Caché Specialized System Tools and Utilities.

Encoding translation translates string between the internal encoding style used on your system and another encoding style. You can perform input translation; that is, translate string from an external encoding style to the encoding style of your system. You can also perform output translation; that is, translate string from the encoding style of your system to an external encoding style. For further I/O translation options, including non-ASCII and customized translation, refer to the section on “System Classes for National Language Support” in Caché Specialized System Tools and Utilities.

The values you can use for mode are as follows:
Meaning Mode Code

U or u Uppercase translation: Convert all characters in string to uppercase.

L or l Lowercase translation: Convert all characters in string to lowercase.

T or t Titlecase translation: Convert all characters in string to titlecase. Titlecase is only meaningful for those alphabets (principally Eastern European) that have three forms for a letter: uppercase, lowercase, and titlecase. For all other letters, titlecase translation is the same as uppercase translation.

W or w Word translation: Convert the first character of each word in string to uppercase. Any character preceded by a blank space, a quotation mark ("), an apostrophe (‘), or an open parenthesis () is considered the first character of a word. Word translation converts all other characters to lowercase. Word translation is locale specific; the above syntax rules for English may differ for other language locales.

S or s Sentence translation: Convert the first character of each sentence in string to uppercase. The first non-blank character of string, and any character preceded by a period (.), question mark (?), or exclamation mark (!) is considered the first character of a sentence. (Blank spaces between the preceding punctuation character and the letter are ignored.) If this character is a letter, it is converted to uppercase. Sentence translation converts all other letter characters to lowercase. Sentence translation is locale specific; the above syntax rules for English may differ for other language locales.

I or i Perform input encoding translation on a specified string. For the two-argument form, the translation is performed using the current process I/O translation handle. If a current process I/O translation handle has not been defined, Caché performs translation based on the default process I/O translation table name.

O or o Perform output encoding translation on a specified string. For the two-argument form, the translation is performed using the current process I/O translation handle. If a current process I/O translation handle has not been defined, Caché performs translation based on the default process I/O translation table name.

If mode is a null string or any value other than the valid characters, you receive a <FUNCTION> error.

Letter Case Translation

You can convert letters in strings to all uppercase letters or all lowercase letters. Conversion works on Unicode letters as well as ASCII letters. The following example converts the Greek alphabet from lowercase to uppercase:

```
IF $SYSTEM.Version.IsUnicode() {
   FOR i=945:1:969 {WRITE $ZCONVERT($CHAR(i),"U")}
} else {WRITE "This example requires a Unicode installation of Caché"}
```

However, a small number of letters only have a lowercase letter form. For example, the German eszett ($CHAR(223)) is only defined as a lowercase letter. Attempting to convert it to an uppercase letter results in the same lowercase letter:

```
IF $ZCONVERT($CHAR(223),"U")=$ZCONVERT($CHAR(223),"L") {
   WRITE "uppercase and lowercase letter are the same"
} else {WRITE "uppercase and lowercase are different"}
```

For this reason, when converting alphanumeric strings to a single letter case it is always preferable to convert to lowercase.

You can perform similar letter case translations using the $TRANSLATE function, as shown in the following example:

```
WRITE $TRANSLATE(text,"ABCDEFGHIJKLMNOPQRSTUVWXYZ","abcdefghijklmnopqrstuvwxyz")
```
**Word and Sentence Translation**

“W” and “S” modes determine whether a non-blank character is the first character of a word or the first character of a sentence, and if that character is a letter, translate it to uppercase. All other letters are translated to lowercase. Case translation works on letters in any alphabet, as shown in the following example which converts Greek letters ($\text{CHAR}(945)$ is lowercase alpha; $\text{CHAR}(913)$ is uppercase alpha):

```
IF $\text{SYSTEM}.\text{Version}.\text{IsUnicode}()$
   SET greek=$\text{CHAR}(945,946,947,913,914,915)$
   WRITE $\text{ZCONVERT}($greek,$"W")$
ELSE {WRITE "This example requires a Unicode installation of Caché")
```

However the rules determining what constitutes a word or sentence are locale dependent. For example, the following example uses the Spanish inverted exclamation point $\text{CHAR}(161)$. The default (English) locale does not recognize this character as beginning a sentence or word. In this example, all letters in `spanish` are translated to lowercase:

```
SET spanish=$\text{CHAR}(161)\_"ola MuNdO! "$\text{CHAR}(161)\_"olA!"
SET english="hElLo wOrLd! heLLo!"
WRITE !,$\text{ZCONVERT}(english,\"S")
WRITE !,$\text{ZCONVERT}(spanish,\"S")
```

**Titlecase Translation**

Titlecase (“T”) mode converts every letter in the string to its titlecase form. Titlecase does not selectively uppercase letters based on their position in a word or string. Titlecase is the case that a letter is represented in when it is the first character of a word in a title. For standard Latin letters, the titlecase form is the same as the uppercase form.

Some languages (for example, Croatian) represent particular letters by two letter glyphs. For example, “lj” is a single letter in the Croatian alphabet. This letter has three forms: lowercase “lj”, uppercase “LJ”, and titlecase “Lj”. `$\text{ZCONVERT}` titlecase translation is used for this type of letter conversion.

**Three-Parameter Form: Encoding Translation**

$\text{ZCONVERT}($string, mode, trantable$)$ performs either an input encoding translation or an output encoding translation on $\text{string}$. In the three-argument form, the mode values you can use are either "I" or "O". You must define the mode value. For “I” translations, the string may be a hexadecimal string, such as %4B (the letter “K”); hexadecimal strings are not case-sensitive.

You can use $\text{ZZDUMP}$ to display the hexadecimal encoding for a string of characters. You can use $\text{SCHAR}$ to specify a character (or string of characters) by its decimal (base 10) encoding; you can use $\text{SZHEX}$ to converts a hexadecimal number to a decimal number, or a decimal number to a hexadecimal number. If the translated value is a non-printing character, Caché displays it as a null string. If the target device cannot represent a translated character, Caché substitutes a question mark (?) character for the non-displayable character.

The trantable value can be a numeric character or a string that specifies the translation table or translation handle to use. The trantable value can be:

- “RAW” which performs no translation for 8-bit characters or 16-bit Latin-1 characters (Unicode characters in which the high-order byte has the value 00). RAW translation should not be used for InterSystems IRIS systems using non-Latin-1 locales, such as rusw.
- “SAME” which translates 8-bit characters to the corresponding Unicode characters.
- “HTML” which adds (output mode) or removes (input mode) HTML escape characters to a string.
- “JS” (or “JSML”) which uses a supplied JavaScript translation table to escape characters in the string for use within JavaScript. For output translations see the table below. For comparison of JS and JSML, see $\text{JS}$ and $\text{JSML}$, $\text{JSON}$ and $\text{JSONML}$ Conversions. For input translations, “\"0\"”, “\"00\"”, “\"x00\””, and “\"u0000\”” are all valid escape sequences for NULL.
• “JSON” (or “JSONML”) which uses a supplied translation table to convert to JSON format. For output translations see the table below. For comparison of JSON and JSONML, see JS and JSML, JSON and JSONML Conversions. For input translations, “\0”, “\000”, “\x00”, and “\u0000” are all valid escape sequences for NULL.

• “URI” which adds (output mode) or removes (input mode) URI parameter escape characters to a string. URI encodes the characters :/?#[]@!$&’()*+,;= as follows: %3A%2F%3F%23%5B%5D%40%21%24%26%27%28%29%2A%2B%2C%3B%3D; it does not encode the tilde (~) character. URI encodes characters higher than $CHAR(255) (Unicode characters) as UTF-8 and then % encodes the UTF-8 values in hexadecimal notation.

• “URL” which adds (output mode) or removes (input mode) URL parameter escape characters to a string. URL encodes the characters :/?#[]@+,;=~ as follows: %3A%3F%23%5B%5D%40%2B%2C%3B%3D%7E. Characters higher than $CHAR(255) are represented in Unicode hexadecimal notation: $CHAR(256) = %u100.

• “UTF8” (UTF-8 encoding) which converts (output mode) 16-bit Unicode characters to a series of 8-bit characters. An ASCII 16–bit Unicode character translates to a single 8–bit character; for example, hex 0041 (the letter “A”) translates to the 8-bit character hex 41. A non-ASCII Unicode character is converted to two or three 8–bit characters. Unicode hex 0080 through 07FF convert to two 8–bit characters; these include the Latin-1 Supplement and Latin Extended characters and the Greek, Cyrillic, Hebrew, and Arabic alphabets. Unicode hex 0800 through FFFF convert to three 8–bit characters; these comprise the rest of the Unicode Basic Multilingual Plane. Thus, the ASCII characters $CHAR(0) through $CHAR(127) are the same in RAW and UTF8 mode; characters $CHAR(128) and above are converted. Input mode reverse this conversion. Refer to Unicode in Using ObjectScript for further details.

• “XML” which adds (output mode) or removes (input mode) XML escape characters to a string.

The following is a table of Output mode escape characters:

<table>
<thead>
<tr>
<th></th>
<th>HTML</th>
<th>JS</th>
<th>JSON</th>
<th>URL</th>
<th>XML</th>
</tr>
</thead>
<tbody>
<tr>
<td>null $CHAR(0)</td>
<td>\x00</td>
<td>\u0000</td>
<td>%00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$CHAR(1) through $CHAR(7)</td>
<td>\x01 through \x07</td>
<td>\u0001 through \u0007</td>
<td>%01 through %07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>backspace $CHAR(8)</td>
<td>\b</td>
<td>\b</td>
<td>%08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>horizontal tab $CHAR(9)</td>
<td>\t</td>
<td>\t</td>
<td>%09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>line feed $CHAR(10)</td>
<td>\n</td>
<td>\n</td>
<td>%0A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vertical tab $CHAR(11)</td>
<td>\v</td>
<td>\u000B</td>
<td>%0B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>form feed $CHAR(12)</td>
<td>\f</td>
<td>\f</td>
<td>%0C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>carriage return $CHAR(13)</td>
<td>\r</td>
<td>\r</td>
<td>%0D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$CHAR(14) through $CHAR(31)</td>
<td>\u000E through \u001F</td>
<td>%0E through %1F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$CHAR(32)</td>
<td>%20</td>
<td></td>
<td></td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot; (doubled)</td>
<td>&quot;</td>
<td>&quot;</td>
<td>%22</td>
<td>&quot;</td>
<td></td>
</tr>
</tbody>
</table>
### URL and URI Conversions

A URL or URI can only contain certain 8-bit ASCII characters. All other characters must be represented by an escape sequence beginning with %. If you wish to convert a string containing Unicode characters to a URL or URI, you must first convert your local representation to an 8-bit intermediate representation, using UTF8 encoding. You then convert the UTF8

### URL and URI Conversions

<table>
<thead>
<tr>
<th>HTML</th>
<th>JS</th>
<th>JSON</th>
<th>URL</th>
<th>XML</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td></td>
<td></td>
<td>%23</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td></td>
<td></td>
<td>%25</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>&amp;</td>
<td></td>
<td>%26</td>
<td>&amp;</td>
</tr>
<tr>
<td>`</td>
<td>#39;</td>
<td>\</td>
<td></td>
<td>'</td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td>%2B</td>
<td></td>
</tr>
<tr>
<td>,</td>
<td></td>
<td></td>
<td>%2C</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>\</td>
<td></td>
<td>%3A</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
<td></td>
<td>%3B</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td></td>
<td>%3C</td>
<td>&lt;</td>
</tr>
<tr>
<td>=</td>
<td></td>
<td></td>
<td>%3D</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td></td>
<td>%3E</td>
<td>&gt;</td>
</tr>
<tr>
<td>?</td>
<td></td>
<td></td>
<td>%3F</td>
<td></td>
</tr>
<tr>
<td>@</td>
<td></td>
<td></td>
<td>%40</td>
<td></td>
</tr>
<tr>
<td>[</td>
<td></td>
<td></td>
<td>%5B</td>
<td></td>
</tr>
<tr>
<td>\</td>
<td>\ \</td>
<td>\ \</td>
<td>%5C</td>
<td></td>
</tr>
<tr>
<td>]</td>
<td></td>
<td></td>
<td>%5D</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td></td>
<td></td>
<td>%5E</td>
<td></td>
</tr>
<tr>
<td>'</td>
<td></td>
<td></td>
<td>%60</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
<td></td>
<td>%7B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>%7C</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
<td>%7D</td>
<td></td>
</tr>
<tr>
<td>~</td>
<td></td>
<td></td>
<td>%7E</td>
<td></td>
</tr>
<tr>
<td>$CHAR(127) through $CHAR(159)</td>
<td></td>
<td></td>
<td>%7F through %9F</td>
<td></td>
</tr>
<tr>
<td>$CHAR(160)</td>
<td> </td>
<td></td>
<td>%A0</td>
<td></td>
</tr>
<tr>
<td>$CHAR(161) through $CHAR(255)</td>
<td></td>
<td></td>
<td>%A1 through %FF</td>
<td></td>
</tr>
</tbody>
</table>

---

URL and URI Conversions

A URL or URI can only contain certain 8-bit ASCII characters. All other characters must be represented by an escape sequence beginning with %. If you wish to convert a string containing Unicode characters to a URL or URI, you must first convert your local representation to an 8-bit intermediate representation, using UTF8 encoding. You then convert the UTF8 encoding to a string suitable for representation in a URL or URI.
results to URL encoding. To convert a URL back to its original Unicode string, you perform the reverse operation. This is shown in the following example:

```objectscript
IF $SYSTEM.Version.IsUnicode() {
    SET ustring="US$ to " _$CHAR(8364) _" échange"
    WRITE "initial string is: ", ustring, !
    ConvertUnicodeToURL
    SET utf0 = $ZCONVERT(ustring,"O","UTF8")
    SET urlo = $ZCONVERT(utf0,"O","URL")
    WRITE "Unicode to URL conversion: ", urlo, !
    ConvertURLtoUnicode
    SET urli = $ZCONVERT(urlo,"I","URL")
    SET utfi = $ZCONVERT(urli,"I","UTF8")
    WRITE "URL to Unicode conversion: ", utfi
} ELSE {WRITE "This example requires a Unicode installation of Caché"}
```

### JS and JSML, JSON and JSONML Conversions

The JS and JSON translations use UTF-8 encoding for Unicode characters. The JSML and JSONML translations render Unicode characters without encoding. For ASCII characters ($CHAR(0) through $CHAR(127)), JS and JSML encodings are identical. For ASCII characters ($CHAR(0) through $CHAR(127)), JSON and JSONML encodings are identical.

The following example compares the translation of JS and JSML characters:

```objectscript
FOR i=1:1:256 {SET x=$ZCVT($C(i),"O","JS") SET y=$ZCVT($C(i),"O","JSML")
    IF x=y {WRITE "."}
    ELSE {WRITE !,$ZHEX(i),!,"JS: " ZZDUMP x WRITE !,"JSML: " ZZDUMP y }
}
```

### Four-Parameter Form: Input/Output String

The `handle` parameter is a local variable that `$ZCONVERT` reads at the beginning of execution and writes when it completes execution. It is used to hold information between consecutive invocations of the `$ZCONVERT` function. It can be used for two purposes: concatenating a string to the beginning of `string`, and converting extremely long strings.

To concatenate a string to the beginning of `string`, set `handle` before invoking `$ZCONVERT`:

```objectscript
SET handle="the 
WRITE $ZCVT("quick brown fox","O","URL",handle),!
/* the%20quick%20brown%20fox  */
WRITE $ZCVT("quick brown fox","O","URL",handle),!
/* quick%20brown%20fox */
```

Note that `$ZCONVERT` resets `handle` when it completes execution. In the previous example, it resets `handle` to the empty string.

This `handle` parameter may be used for input conversions. Specifying a `handle` is useful when dealing with multibyte character sequences when working with partial sets of characters, such as a stream read. In these cases, `$ZCONVERT` uses the `handle` parameter to hold a partial character sequence that may be the leading bytes of a multibyte sequence. If there are input characters left in the buffer at the end of a `$ZCONVERT` which do not make a complete translation unit, these leftover characters are returned in the `handle`. At the beginning of next `$ZCONVERT`, if the `handle` contains data, these leftover characters are prepended to the normal input data. This is particularly valuable for use in UTF8 conversions, as shown in the following example:

```objectscript
SET handle="" 
WHILE 'stream.AtEnd() {
    WRITE $ZCONVERT(stream.Read(20000),"I","UTF8",handle)
}
```

To convert an extremely long string, it may be necessary to perform more than one string conversions by invoking `$ZCONVERT` multiple times. `$ZCONVERT` provides the optional `handle` parameter to hold the remaining unconverted portion of `string`. If you specify a `handle` parameter, it is updated by each invocation of `$ZCONVERT`. When the string conversion completes, `$ZCONVERT` sets `handle` to the empty string.
SET handle=""
SET out = $ZCVT(hugestring,"O","HTML",handle)
IF handle ' = "" { 
    SET out2 = $ZCVT(handle,"O","HTML",handle)
    WRITE "Converted string is: ",out,out2 
} ELSE { 
    WRITE "Converted string is: ",out }

Examples

The following example returns "HELLO":

    WRITE $ZCONVERT("Hello","U")

The following example returns "hello":

    WRITE $ZCVT("Hello","L")

The following example returns "HELLO":

    WRITE $ZCVT("Hello","T")

The following example uses the concatenate operator (_) to append and case-convert an accented character:

    WRITE "CACH"_SCHAR(201),!, $ZCVT("CACH"_SCHAR(201),"L")
returns:
CACHÉ
caché

The following example converts the angle brackets in the string to HTML escape characters for output, returning "&TAG;":

    WRITE $ZCVT("<TAG>","O","HTML")

Note that how these angle brackets display depends on the output device; try running this program here and then running it from the Terminal prompt.

The following example shows how $ZCONVERT substitutes a ? character for a translated character it cannot display. Both the UTF8 and the current process I/O translation object (trantable 0) conversions in this example display SCHAR(63), which is the actual ? character. UTF8 cannot display translated characters above SCHAR(127). Translation table 0 cannot display translated characters above SCHAR(255):

    FOR i=1:1:300 {IF $ZCONVERT($CHAR(i),"I","UTF8") ' = '?' { CONTINUE } ELSE {WRITE "UTF8 ",i,"=",$ZCONVERT($CHAR(i),"I","UTF8")
    IF $ZCONVERT($CHAR(i),"I",0) ' = '?' {WRITE " trantable 0 ",i,"=",$ZCONVERT($CHAR(i),"I",0),! }
    ELSE {WRITE ! }
    }

See Also

- $ASCII function
- $CHAR function
- $ZSTRIP function
- Pattern Matching operators in Using Caché ObjectScript
$ZCOS

Cosine function.

$ZCOS(n)

Parameter

$n \quad \text{An angle in radians ranging from Pi to 2 Pi (inclusive). Other supplied numeric values are converted to a value within this range.}$

Description

$ZCOS$ returns the trigonometric cosine of $n$. The result is a signed decimal number ranging from -1 to +1. $ZCOS(0)$ returns 1. $ZCOS($ZPI) returns -1.

Parameter

$n$

An angle in radians ranging from Pi to 2 Pi (inclusive). It can be specified as a value, a variable, or an expression. You can specify the value Pi by using the $ZPI$ special variable. You can specify positive or negative values smaller than Pi or larger than 2 Pi; Caché resolve these values to the corresponding multiple of Pi. For example, 3 Pi is equivalent to Pi, negative Pi is equivalent to Pi, and zero is equivalent to 2 Pi.

A non-numeric string is evaluated as 0.

Examples

The following example permits you to compute the cosine of a number:

```objectscript
READ "Input a number: ",num
IF $ZABS(num)>(2*$ZPI) { WRITE !,"number is a larger than 2 pi" }
ELSE {
    WRITE !,"the cosine is: ",ZCOS(num)
}
QUIT
```

The following example compares the results from Caché fractional numbers ($DEcimal$ numbers) and $DOUBLE$ numbers. In both cases, the cosine of 0 is exactly 1, the cosine of pi is exactly -1:

```objectscript
WRITE !,"the cosine is: ",$ZCOS(0.0)
WRITE !,"the cosine is: ",$ZCOS($DOUBLE(0.0))
WRITE !,"the cosine is: ",$ZCOS($PI)
WRITE !,"the cosine is: ",$ZCOS($DOUBLE($PI))
```

See Also

- $ZSIN$ function
- $ZARCCOS$ function
- $ZPI$ special variable
$ZCOT

Cotangent function.

$ZCOT(n)

Parameter

$n$
An angle in radians.

Description

$ZCOT$ returns the trigonometric cotangent of $n$. The result is a signed decimal number.

Parameter

$n$
An angle in radians, specified as a nonzero value. It can be specified as a value, a variable, or an expression. A value of 0 generates an <ILLEGAL VALUE> error. A non-numeric string is evaluated as 0.

Examples

The following example permits you to compute the cotangent of a number:

    READ "Input a number: ",num
    IF num=0 { WRITE !,"zero is an illegal value" }
    ELSE {
        WRITE !,"the cotangent is: ",ZCOT(num)
    }
    QUIT

The following example compares the results from Caché fractional numbers ($DECIMAL$ numbers) and $DOUBLE$ numbers:

    WRITE !,"the cotangent is: ",$ZCOT(1.0)
    WRITE !,"the cotangent is: ",$ZCOT($DOUBLE(1.0))
    WRITE !,"the cotangent is: ",$ZCOT(-1.0)
    WRITE !,"the cotangent is: ",$ZCOT($DOUBLE(-1.0))
    WRITE !,"the cotangent is: ",$ZCOT($ZPI/2)
    WRITE !,"the cotangent is: ",$ZCOT($DOUBLE($ZPI)/2)

Note that the cotangent of pi/2 is a fractional number, not 0.

See Also

- $ZTAN$ function
- $ZPI$ special variable
$ZCRC

Checksum function.

$ZCRC(string, mode, expression)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A string on which a checksum operation is performed.</td>
</tr>
<tr>
<td>mode</td>
<td>An integer code specifying the checksum mode to use.</td>
</tr>
<tr>
<td>expression</td>
<td>Optional — The initial &quot;seed&quot; value, specified as an integer. If omitted, defaults to zero (0).</td>
</tr>
</tbody>
</table>

Description

$ZCRC performs a cyclic redundancy check on string and returns an integer checksum value. The value returned by $ZCRC depends on the parameters you use.

- $ZCRC(string, mode) computes and returns a checksum on string. The value of mode determines the type of checksum $ZCRC computes.
- $ZCRC(string, mode, expression) computes and returns a checksum on string using the mode specified by mode. expression supplies an initial "seed" value when checking multiple strings. It allows you to run $ZCRC calculations sequentially on multiple strings and obtain the same checksum values as if you had concatenated those strings and then run $ZCRC on the resulting string.

Parameters

string

A byte string. Can be specified as a value, a variable, or an expression. Only use a byte string or you will receive a <FUNCTION> error.

mode

The checksum algorithm to use. All checksum modes can be used with 8-bit (ASCII) or 16-bit Unicode (wide) characters. Legal values for mode are:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Computes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>An 8-bit byte sum. Simply sums the ASCII values of the characters in the string. Thus $ZCRC(2,0)=50, $ZCRC(22,0)=100, $ZCRC(23,0)=101, and $ZCRC(32,0)=101.</td>
</tr>
<tr>
<td>1</td>
<td>An 8-bit XOR of the bytes</td>
</tr>
<tr>
<td>2</td>
<td>A 16-bit DataTree CRC-CCITT</td>
</tr>
<tr>
<td>3</td>
<td>A 16-bit DataTree CRC-16</td>
</tr>
<tr>
<td>4</td>
<td>A 16-bit CRC for XMODEM protocols</td>
</tr>
<tr>
<td>5</td>
<td>A correct 16-bit CRC-CCITT</td>
</tr>
<tr>
<td>6</td>
<td>A correct 16-bit CRC-16</td>
</tr>
<tr>
<td>7</td>
<td>A correct 32-bit CRC-32. This corresponds to the cksum utility algorithm 3 on OS X, and the CRC32 class in the Java utilities package.</td>
</tr>
</tbody>
</table>
Caché in MSM language mode supports \textit{mode} values 0 and 1. All other \textit{mode} values result in a <FUNCT> error.

\textit{expression}

An argument that is an initial "seed" value. \$ZCRC adds \textit{expression} to the checksum generated for \textit{string}. This allows you to run \$ZCRC calculations on multiple strings sequentially and obtain the same checksum value as if you had concatenated those strings and run \$ZCRC on the resulting string.

\textbf{Examples}

The following example uses \textit{mode}=0 on strings containing the letters A, B, and C and in each case returns the checksum 198:

```objectscript
WRITE $ZCRC("ABC",0),!
WRITE $ZCRC("CAB",0),!
WRITE $ZCRC("BCA",0),!
```

The checksum is derived as follows:

```objectscript
WRITE $ASCII("A")+$ASCII("B")+$ASCII("C") /* 65+66+67 = 198 */
```

The following example shows the values returned by each \textit{mode} for the string “ABC”:

```objectscript
FOR i=0:1:7 {
  WRITE !,"mode ",i,="",$ZCRC("ABC",i)
}
```

\textbf{See Also}

- \$ZCYC function
Cosecant function.

$ZCSC(n)$

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>An angle in radians.</td>
</tr>
</tbody>
</table>

Description

$ZCSC$ returns the trigonometric cosecant of $n$. The result is a signed decimal number.

Parameter

$n$

An angle in radians, specified as a nonzero number. It can be specified as a value, a variable, or an expression. Specifying 0 generates an <ILLEGAL VALUE> error; specifying $DOUBLE(0)$ generates a <DIVIDE> error. A non-numeric string is evaluated as 0.

Examples

The following example permits you to compute the cosecant of a number:

```objectscript
READ "Input a number: ", num
IF num=0 { WRITE !,"ILLEGAL VALUE: zero not permitted" }
ELSE {
  WRITE !,"the cosecant is: ",$ZCSC(num)
}
QUIT
```

The following example compares the results from Caché fractional numbers ($DECIMAL$ numbers) and $DOUBLE$ numbers. In both cases, the cosecant of pi/2 is exactly 1:

```objectscript
WRITE !,"the cosecant is: ",$ZCSC($ZPI)
WRITE !,"the cosecant is: ",$ZCSC($DOUBLE($ZPI))
WRITE !,"the cosecant is: ",$ZCSC($ZPI/2)
WRITE !,"the cosecant is: ",$ZCSC($DOUBLE($ZPI)/2)
WRITE !,"the cosecant is: ",$ZCSC($DOUBLE($ZPI/2))
```

See Also

- $ZSEC$ function
- $ZCOT$ function
- $ZPI$ special variable
$ZCYC

Cyclical-redundancy check for data integrity.

$ZCYC(string)
$ZC(string)

Parameter

| string | A string. |

Description

$ZCYC(string) computes and returns the cyclical-redundancy check value for the string. It allows two intercommunicating programs to check for data integrity.

The sending program transmits a piece of data along with a matching check value that it calculates using $ZCYC. The receiving program verifies the transmitted data by using $ZCYC to calculate its check value. If the two check values match, the received data is the same as the sent data.

$ZCYC calculates the check value by performing an exclusive OR (XOR) on the binary representations of all the characters in the string.

Use caution when transmitting data between 8-bit and Unicode (16-bit) implementations of Caché; if a data string does not contain any wide characters, the cyclical-redundancy check values should match.

Note that the $ZCYC value of an 8-bit string is identical to the $ZCRC mode 1 value.

Parameter

string

A string. Can be specified as a value, a variable, or an expression. String values are enclosed in quotation marks.

Example

In this example, the first $ZCYC returns 65; the second returns 3; and the third returns 64.

```
SET x= $ZCYC("A")
  ; 1000001 (only one character; no XOR )
SET y= $ZCYC("AB")
  ; 1000001 XOR 1000010 -> 0000011
SET z= $ZCYC("ABC")
  ; 1000001 XOR 1000010 -> 0000011 | 1000011 -> 100000
WRITE !,"x=",x," y=",y," z=",z
```

See Also

- $ZCRC function
$ZDASCII

Converts an eight-byte string to a $DOUBLE floating point number.

$ZDASCII(string,position)
$ZDA(string,position)

Parameters

| string | A string or number. It can be a value, a variable, or an expression. It must be a minimum of eight bytes in length. A number is converted to canonical form by removing a leading plus sign, and leading and trailing zeros before it is supplied to $ZDASCII; the resulting canonical number must be a minimum of eight bytes in length. |
| position | Optional — A starting position in the string, expressed as a positive, non-zero integer. The default is 1. Position is counted in single bytes, not eight-byte strings. There must exist at least eight bytes of string from the specified position (inclusive). A numeric position value is parsed as an integer by truncating decimal digits, removing leading zeros and plus signs, etc. |

Description

The value that $ZDASCII returns depends on the parameters you use.

- $ZDASCII(string) returns a $DOUBLE (IEEE floating point) numeric interpretation of an eight-byte string starting at the first character position of string.
- $ZDASCII(string,position) returns a $DOUBLE (IEEE floating point) numeric interpretation of an eight-byte string beginning at the starting byte position specified by position.

$ZDASCII can return either a positive or a negative number.

$ZDASCII issues a <FUNCTION> error if string is less than eight bytes in length, if the specified position results in a remaining string of less than eight bytes, or if position is an invalid value.

Examples

The following example determines the numeric interpretation of the numeric 12345678:

WRITE $ZDASCII(12345678)

It returns: .0000000000000000000000000000000000000068213200517013251261

The following examples also return the same value:

WRITE !, $ZDASCII("12345678",1)
WRITE !, $ZDASCII("+001234567899",1)
WRITE !, $ZDASCII("+-001234567899",5)

The following example determines the numeric interpretation of the negative numeric −1234567. Note that the minus sign is counted as a byte:

WRITE $ZDASCII(-1234567)

It returns: .00000000000000000000000000000000000000099583343788967399388

The following example determines the numeric interpretation of the character string “abcdefgh”:

WRITE $ZDASCII("abcdefgh")
It returns: 85408832230361244356000000000000000000000000000000000000000000000000000

Notes

$ZDASCII and Other $ASCII Functions

$ZDASCII converts an eight byte (64-bit) character string to an IEEE floating point number. $ZQASCII converts an eight byte (64-bit) character string to an integer. To convert an 8-bit byte string to an integer use $ASCII. To convert a 16-bit (wide) character string to an integer use $ZWASCII. To convert a 32-bit (long) character string to an integer use $ZLASCII.

See Also

• $ASCII function
• $DOUBLE function
• $ZDCHAR function
• $ZLCHAR function
• $ZWASCII function
• $ZQASCII function
$ZDATE

Validates a date and converts it from internal format to the specified display format.

$ZDATE(hdate,dformat,monthlist,yearopt,startwin,endwin,mindate,maxdate,erropt,localeopt)
$ZD(hdate,dformat,monthlist,yearopt,startwin,endwin,mindate,maxdate,erropt,localeopt)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>hdate</strong></td>
<td>An integer specifying an internal date format value. This integer represents</td>
</tr>
<tr>
<td></td>
<td>the number of days elapsed since December 31, 1840. If $SHOROLOG$ is</td>
</tr>
<tr>
<td></td>
<td>specified for <em>hdate</em>, only the date portion of $SHOROLOG$ is used. See</td>
</tr>
<tr>
<td></td>
<td><em>hdate</em> below.</td>
</tr>
<tr>
<td><strong>dformat</strong></td>
<td>Optional — An integer code specifying the format for the returned date. See</td>
</tr>
<tr>
<td></td>
<td><em>dformat</em> below.</td>
</tr>
<tr>
<td><strong>monthlist</strong></td>
<td>Optional — A string or the name of a variable that specifies a set of</td>
</tr>
<tr>
<td></td>
<td>month names. This string must begin with a delimiter character, and its 12</td>
</tr>
<tr>
<td></td>
<td>entries must be separated by this delimiter character. See <em>monthlist</em></td>
</tr>
<tr>
<td></td>
<td>below.</td>
</tr>
<tr>
<td><strong>yearopt</strong></td>
<td>Optional — An integer code that specifies whether to represent years as two-</td>
</tr>
<tr>
<td></td>
<td>or four-digit values. See <em>yearopt</em> below.</td>
</tr>
<tr>
<td><strong>startwin</strong></td>
<td>Optional — The start of the sliding window during which dates must be</td>
</tr>
<tr>
<td></td>
<td>represented with two-digit years. See <em>startwin</em> below.</td>
</tr>
<tr>
<td><strong>endwin</strong></td>
<td>Optional — The end of the sliding window during which dates are represented</td>
</tr>
<tr>
<td></td>
<td>with two-digit years. See <em>endwin</em> below.</td>
</tr>
<tr>
<td><strong>mindate</strong></td>
<td>Optional — The lower limit of the range of valid dates. Specified as a $</td>
</tr>
<tr>
<td></td>
<td>SHOROLOG integer date count, with 0 representing December 31, 1840. Can be</td>
</tr>
<tr>
<td></td>
<td>specified as a positive or negative integer. See <em>mindate</em> below.</td>
</tr>
<tr>
<td><strong>maxdate</strong></td>
<td>Optional — The upper limit of the range of valid dates. Specified as a $</td>
</tr>
<tr>
<td></td>
<td>SHOROLOG integer date count. See <em>maxdate</em> below.</td>
</tr>
<tr>
<td><strong>erropt</strong></td>
<td>Optional — An expression to return when <em>hdate</em> is invalid. Specifying a</td>
</tr>
<tr>
<td></td>
<td>value for this parameter suppresses error codes associated with invalid or</td>
</tr>
<tr>
<td></td>
<td>out of range <em>hdate</em> values. Instead of issuing an error message, $ZDATE$</td>
</tr>
<tr>
<td></td>
<td>returns <em>erropt</em>. See <em>erropt</em> below.</td>
</tr>
<tr>
<td><strong>localeopt</strong></td>
<td>Optional — A boolean flag that specifies which locale to use for the</td>
</tr>
<tr>
<td></td>
<td><em>dformat</em>, <em>monthlist</em>, <em>yearopt</em>, <em>mindate</em> and <em>maxdate</em> default values,</td>
</tr>
<tr>
<td></td>
<td>and other date characteristics, such as the date separator character:</td>
</tr>
<tr>
<td></td>
<td><em>localeopt</em>=0: the current locale property settings determine these</td>
</tr>
<tr>
<td></td>
<td>parameter defaults.</td>
</tr>
<tr>
<td></td>
<td><em>localeopt</em>=1: the ODBC standard locale determines these parameter</td>
</tr>
<tr>
<td></td>
<td>defaults.</td>
</tr>
<tr>
<td></td>
<td><em>localeopt</em> not specified: the <em>dformat</em> value determines these parameter</td>
</tr>
<tr>
<td></td>
<td>defaults. For all other <em>dformat</em> values, current locale property settings</td>
</tr>
<tr>
<td></td>
<td>are used as defaults.</td>
</tr>
</tbody>
</table>

Omitted parameters between specified parameter values are indicated by placeholder commas. Trailing placeholder commas are not required, but are permitted. Blank spaces are permitted between the commas that indicate an omitted parameter.
Description

The $ZDATE function converts a specified date in internal storage format ($HOROLOG format) to one of several alternate date display formats. The value returned by $ZDATE depends on the parameters you use.

Simple $ZDATE format

$ZDATE(hdate), the most basic form of $ZDATE, returns the date in a display format that corresponds to the specified hdate. hdate is an integer count of the number of days elapsed since December 31, 1840. It can range from 0 to 2980013 (12/31/1840 to 12/31/9999).

By default, $ZDATE(hdate) represents years between 1900 and 1999 with two digits. It represents years that fall before 1900 or after 1999 with four digits. For example:

```
WRITE $ZDATE(21400),! ; returns 08/04/1899
WRITE $ZDATE(50000),! ; returns 11/23/77
WRITE $ZDATE(60000),! ; returns 04/10/2005
WRITE $ZDATE(0),!      ; returns 12/31/1840
```

When you supply a $HOROLOG date to $ZDATE, only the date portion is used. In $HOROLOG format, date and time are presented as two integers separated by a comma. Upon encountering the comma (a non-numeric character) $ZDATE ignores the rest of the string. In the following example, $ZDATE returns 04/10/2005 and the current date using $HOROLOG format values:

```
WRITE !,$ZDATE("60000,12345")
WRITE !,$ZDATE($HOROLOG)
```

Customizable Date Default

Upon Caché startup, the default date format is initialized to dformat=1, which is the American date format with a slash date separator (MM/DD/[YY]YY). To set this and other default formats to the values for your current locale, set the following global variable: SET ^SYS("NLS","Config","LocaleFormat")=1. This sets all format defaults for all processes to your current locale values. These defaults persist until this global is changed.

Note: This section describes the user locale definitions applied when localeopt is undefined or set to 0. When localeopt=1, $ZDATE uses a predefined ODBC locale.

You can use NLS (National Language Support) to override format defaults for the current process. You can either change the all format defaults to the values for a specified locale, or change individual format values.

- To set all of the format defaults (including the date format default) to the properties of a specified locale, invoke the following method call: SET fmt=##class(%SYS.NLS.Format).%New("lname"), where lname is the NLS name of the desired locale. (For example, deuw=German, espw=Spanish, fraw=French, ptbw=Brazilian Portuguese, rusw=Russian, jpnw=Japanese. A complete list of locales is found in the Management Portal: [System] > [Configuration] > [NLS Settings] > [Locale Definitions].) To set these defaults to the properties of the current locale, specify a lname of "current", or the empty string (""").

- To set the default date format to a specified dformat format, invoke the SetFormatItem() method: SET rtn=##class(%SYS.NLS.Format).SetFormatItem("DateFormat",n), where n is the number of the dformat value you wish to make the default.

The following example demonstrates setting all format defaults to the Russian locale, returning a date from $ZDATE in the default format (Russian), then resetting the format defaults to the current locale defaults. Note that the Russian locale uses a period, rather than a slash as the date part separator:
IF $SYSTEM.Version.IsUnicode()
    WRITE !,$ZDATE($HOROLOG)
    SET fmt=##class(%SYS.NLS.Format).%New("rusw")
    WRITE !,$ZDATE($HOROLOG)
    SET fmt=##class(%SYS.NLS.Format).%New("current")
    WRITE !,$ZDATE($HOROLOG)
} ELSE (WRITE "This example requires a Unicode installation of Caché")

The following example demonstrates setting individual format defaults. The first $ZDATE returns a date in the default format. The first SetFormatItem() method changes the default to dformat=4, or the European date format (DD/MM/[YY]YY), as is shown by the second $ZDATE. The second SetFormatItem() method changes the default for the date separator character (which affects the dformat -1, 1, 4, and 15). In this example, the date separator character is set to a dot ("."), as is shown by the third $ZDATE. Finally, this program restores the original date format values:

InitialVals
   SET fmt=##class(%SYS.NLS.Format).GetFormatItem("DateFormat")
   SET sep=##class(%SYS.NLS.Format).GetFormatItem("DateSeparator")
   WRITE !,$ZDATE($HOROLOG)
ChangeVals
   SET x=##class(%SYS.NLS.Format).SetFormatItem("DateFormat",4)
   WRITE !,$ZDATE($HOROLOG)
   SET y=##class(%SYS.NLS.Format).SetFormatItem("DateSeparator",".")
   WRITE !,$ZDATE($HOROLOG)
RestoreVals
   SET x=##class(%SYS.NLS.Format).SetFormatItem("DateFormat",fmt)
   SET y=##class(%SYS.NLS.Format).SetFormatItem("DateSeparator",sep)
   WRITE !,$ZDATE($HOROLOG)

For further details on default date formats for supported locales, refer to Dates in Using Caché ObjectScript.

**Parameters**

**hdate**

The internal date format value representing the number of days elapsed since December 31, 1840. By default, it must be an integer in the range 0 through 2980013. You can specify hdate as a numeric, a string literal, or an expression. Caché converts hdate to canonical form. It truncates a numeric string (such as a $HOROLOG value) at its first non-numeric character. It evaluates a non-numeric string as the integer 0. A floating-point number that does not resolve to an integer generates an <ILLEGAL VALUE> error.

By default, the earliest valid hdate is 0 (December 31, 1840). Dates are limited to positive integers by default because the DateMinimum property defaults to 0. You can specify earlier dates as negative integers, provided the DateMinimum property of the current locale is set to a greater or equal negative integer. The lowest valid DateMinimum value is -672045, which corresponds to January 1, 0001. Caché uses the proleptic Gregorian calendar, which projects the Gregorian calendar back to “Year 1”, in conformance with the ISO 8601 standard. This is, in part, because the Gregorian calendar was adopted at different times in different countries. For example, much of continental Europe adopted it in 1582; Great Britain and the United States adopted it in 1752. Thus Caché dates prior to your local adoption of the Gregorian calendar may not correspond to historical dates that were recorded based on the local calendar then in effect. For further details on dates prior to 1840, refer to the mindate parameter.

Invalid and out-of-range hdate values and resulting errors are described in the erropt parameter.

**dformat**

Format for the returned date. Valid values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MM/DD/[YY]YY (07/01/97 or 03/27/2002) — American numeric format. The dateseparator character (/ or .) is taken from the current locale setting.</td>
</tr>
<tr>
<td>2</td>
<td>DD Mmm [ YY ]YY (01 Jul 97 or 27 Mar 2002)</td>
</tr>
<tr>
<td>Value</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>3</td>
<td>YYYY-MM-DD (1997-07-01 or 2002-03-27) — ODBC format. By default this format is independent of your current locale settings (localeopt=1), thus specifying dates in an ODBC standard interchange format. To use your current date locale settings with this format, set localeopt=0.</td>
</tr>
<tr>
<td>4</td>
<td>DD/MM/[YY][YY] (01/07/97 or 27/03/2002) — European numeric format. The dateseparator character (/ or .) is taken from the current locale setting.</td>
</tr>
<tr>
<td>5</td>
<td>Mmm [D]D, YYYY (Jul 1, 1997 or Mar 27, 2002)</td>
</tr>
<tr>
<td>6</td>
<td>Mmm [D]D YYYY (Jul 1 1997 or Mar 27 2002)</td>
</tr>
<tr>
<td>7</td>
<td>Mmm DD [YY][YY] (Jul 01 97 or Mar 27 2002)</td>
</tr>
<tr>
<td>8</td>
<td>YYYYMMDD (19970701 or 20020327) — Numeric format</td>
</tr>
<tr>
<td>9</td>
<td>Mmmmm [D]D, YYYY (July 1, 1997 or March 27, 2002)</td>
</tr>
<tr>
<td>10</td>
<td>W (2) — Day number for the week, numbered from 0 (Sunday) through 6 (Saturday). Compare with the $SYSTEM.SQL.DAYOFWEEK() method.</td>
</tr>
<tr>
<td>11</td>
<td>Www (Tue) — Abbreviated day name</td>
</tr>
<tr>
<td>12</td>
<td>Wwwwww (Tuesday) — Full day name</td>
</tr>
<tr>
<td>13</td>
<td>[D]D/[M]M/YYYY (1/7/2549 or 27/11/2549) — Thai date format. Day and month are identical to European usage, except no leading zeros. The year is the Buddhist Era (BE) year, calculated by adding 543 years to the Gregorian year.</td>
</tr>
<tr>
<td>14</td>
<td>nnn (354) — Day number for the year</td>
</tr>
<tr>
<td>15</td>
<td>DD/MM/[YY][YY] (01/07/97 or 27/03/2002) — European format (same as dformat=4). The dateseparator character (/ or .) is taken from the current locale setting.</td>
</tr>
<tr>
<td>16</td>
<td>YYYYc[M]Mc[D]Dc — Japanese date format. Year, month, and day numbers are the same as other date formats; leading zeros are omitted. The Japanese characters for “year”, “month”, and “day” (shown here as c) are inserted after the year, month, and day numbers. These characters are Year=$CHAR(24180), Month=$CHAR(26376), and Day=$CHAR(26085).</td>
</tr>
<tr>
<td>17</td>
<td>YYYYc [M]Mc [D]Dc — Japanese date format. Same as dformat 16, except that a blank space is inserted after the “year” and “month” Japanese characters.</td>
</tr>
<tr>
<td>18</td>
<td>[D]D Mmmmm YYYY — Tabular Hijri (Islamic) date format with full month name. Day leading zeros are omitted; year leading zeros are included. Caché date -445031 (07/19/0622 C.E.) = 1 Muharram 0001.</td>
</tr>
<tr>
<td>19</td>
<td>[D]D [M]M YYYY — Tabular Hijri (Islamic) date format with month number. Day and month leading zeros are omitted; year leading zeros are included. Caché date -445031 (07/19/0622 C.E.) = 1 1 0001.</td>
</tr>
<tr>
<td>20</td>
<td>[D]D Mmmmm YYYY — Observed Hijri (Islamic) date format with full month name. Defaults to Tabular Hijri (dformat 18). To override tabular calculation, use the class %Calendar.Hijri to add observations of new moon crescents.</td>
</tr>
<tr>
<td>21</td>
<td>[D]D [M]M YYYY — Observed Hijri (Islamic) date format with month number. Defaults to Tabular Hijri (dformat 19). To override tabular calculation, use the class %Calendar.Hijri to add observations of new moon crescents.</td>
</tr>
</tbody>
</table>
Meaning

-1  Get effective dformat value either from the user's locale (if localeopt=0 or undefined), or from the ODBC locale (which defaults dformat=3). If dformat is taken from the user's locale, it is the value of fmt.DateFormat, where fmt is an instance of ##class(%SYS.NLS.Format) associated with the current process. This is the default behavior if you do not specify dformat. See “Customizable Date Default” for further details.

where:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YYYY</td>
<td>YYYY is a four-digit year. [YY]YY is a two-digit year if hdate falls within the active window for two-digit years; otherwise it is a four-digit year.</td>
</tr>
<tr>
<td>MM</td>
<td>Two-digit month: 01 through 12. [M]M indicates that the leading zero is omitted for months 1 through 9.</td>
</tr>
<tr>
<td>DD</td>
<td>Two-digit day: 01 through 31. [D]D indicates that the leading zero is omitted for days 1 through 9.</td>
</tr>
<tr>
<td>Mmm</td>
<td>Month abbreviation extracted from the MonthAbbr property of the current locale. An alternate month abbreviation (or name of any length) can be extracted from an optional list specified as the monthlist parameter to $ZDATE. The default MonthAbbr values are: “Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec”</td>
</tr>
<tr>
<td>Mmmmm</td>
<td>Full name of the month as specified by the MonthName property of the current locale. The default values are: “January February March ... November December”</td>
</tr>
<tr>
<td>W</td>
<td>Number 0-6 indicating the day of the week: Sunday=0, Monday=1, Tuesday=2, etc.</td>
</tr>
<tr>
<td>Www</td>
<td>Weekday name abbreviation as specified by the WeekdayAbbr property of the current locale. The default values are: “Sun Mon Tue Wed Thu Fri Sat”</td>
</tr>
<tr>
<td>Wwww</td>
<td>Weekday full name as specified by the WeekdayName property of the current locale. The default values are: “Sunday Monday Tuesday ... Friday Saturday”</td>
</tr>
<tr>
<td>nnn</td>
<td>Day number for the specified year, always three digits, with leading zeros if necessary. Values are 001 through 365 (or 366 on leap years).</td>
</tr>
</tbody>
</table>

dformat Default

If you omit dformat or set it to -1, the dformat default depends on the localeopt parameter and the NLS DateFormat property:

- If localeopt=1 the dformat default is ODBC format. The monthlist, yearopt, mindate and maxdate parameter defaults are also set to ODBC format. This is the same as setting dformat=3.
- If localeopt=0 or is unspecified, the dformat default is taken from the NLS DateFormat property. If DateFormat=3, the dformat default is ODBC format. However, DateFormat=3 does not affect the monthlist, yearopt, mindate and maxdate parameter defaults, which are as specified in the current NLS locale definition.

To determine the default date format for your locale, invoke the GetFormatItem() NLS class method:

```ObjectScript
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DateFormat")
```
European date format (dformat=4, DD/MM/YYYY order) is the default for many (but not all) European languages, including British English, French, German, Italian, Spanish, and Portuguese (which use a “/” DateSeparator character), as well as Czech (csyw), Russian (rusw), Slovak (skyw), Slovenian (svnw), and Ukrainian (ukrw) (which use a “.” DateSeparator character). For further details on default date formats for supported locales, refer to Dates in Using Caché ObjectScript.

**dformat Settings**

If dformat is 3 (ODBC date format), ODBC format defaults are also used for the monthlist, yearto, mindate and maxdate parameter defaults. Current locale defaults are ignored.

If dformat is -1, 1, 4, 13, or 15 (numeric date formats), $ZDATE uses the value of the DateSeparator property of the current locale as the delimiter between months, days, and the year. When dformat is 3 the ODBC date separator (“-”) is used. For all other dformat values, a space is used as the date separator. The default value of DateSeparator in English is “/” and all documentation uses this delimiter.

If dformat is 11 or 12 (day names) and localeopt=0 or is unspecified the day name values come from the current locale properties. If localeopt=1, day names come from the ODBC locale. To determine the default weekday names and weekday abbreviations for your locale, invoke the following NLS class methods:

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("WeekdayName"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("WeekdayAbbr"),!
```

If dformat is 16 or 17 (Japanese date formats), the returned date format is independent of the locale setting. Japanese-format dates can be returned from any Unicode Caché instance. On an 8-bit Caché instance specifying dformat is 16 or 17 causes a <ERWIDECHAR> error.

If dformat is 18, 19, 20, or 21 (Islamic date formats) and localeopt is unspecified, parameters default to Islamic defaults, rather than current locale defaults. The monthlist parameter defaults to Arabic month names transliterated with Latin characters. The tformat, yearto, mindate and maxdate parameters default to ODBC defaults. The date separator defaults to the Islamic default (a space), not the ODBC default or the current locale DateSeparator property value. If localeopt=0 current locale property defaults are used for these parameters. If localeopt=1 ODBC defaults are used for these parameters.

**monthlist**

An expression that resolves to a string of month names or month name abbreviations, separated by a delimiter character. The names in monthlist replace the default month abbreviation values from the MonthAbbr property or the month name values from the MonthName property of the current locale.

monthlist is valid only if dformat is 2, 5, 6, 7, 9, 18, or 20. If dformat is any other value $ZDATE ignores monthlist.

The monthlist string has the following format:

- The first character of the string is a delimiter character (usually a space). The same delimiter must appear before the first month name and between each month name in monthlist. You can specify any single-character delimiter; this delimiter appears between the month, day, and year portions of the returned date value, which is why a space is usually the preferred character.
- The month names string should contain twelve delimited values, corresponding to January through December. It is possible to specify more or less than twelve month names, but if there is no month name corresponding to the month in hdate an <ILLEGAL VALUE> error is generated.

If you omit monthlist or specify a monthlist value of -1, $ZDATE uses the list of month names defined in the MonthAbbr or MonthName property of the current locale, unless one of the following is true: If localeopt=1, the monthlist default is the ODBC month list (in English). If localeopt is unspecified and dformat is 18 or 20 (Islamic date formats) the monthlist default is the Islamic month list (Arabic names expressed using Latin letters), ignoring the MonthAbbr or MonthName property value.

To determine the default month names and month abbreviations for your locale, invoke the GetFormatItem() NLS class method:
The following example lists the month names the default locale, changes the locale for this process to the Russian locale, then lists the Russian month names and displays the current date with a Russian month name. It then reverts the locale defaults to current locale and again displays the current date, this time with the default month name.

```objectscript
IF $SYSTEM.Version.IsUnicode() {
    WRITE ##class(%SYS.NLS.Format).GetFormatItem("MonthName"),!
    SET fmt=##class(%SYS.NLS.Format).%New("rusw")
    WRITE fmt.MonthName,!
    WRITE $ZDATE($HOROLOG,9),!
    SET fmt=##class(%SYS.NLS.Format).%New()
    WRITE $ZDATE($HOROLOG,9)
} ELSE {WRITE "This example requires a Unicode installation of Caché"
}
```

### yearopt

With `dformat` values 1, 2, 4, 7, or 15, an integer code that specifies the temporal window in which to display the year as a two-digit value. For all other `dformat` values, the `yearopt` is ignored. Valid `yearopt` values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Get effective <code>yearopt</code> value from <code>YearOption</code> property of current locale which defaults to a value of 0. This is the default behavior if you do not specify <code>yearopt</code>.</td>
</tr>
<tr>
<td>0</td>
<td>Represent 20th century dates (1900 through 1999) with two-digit years and all other dates with four-digit years, unless a process-specific sliding window (established via the <code>%DATE</code> utility) is in effect. If such a window is in effect, represent only those dates falling within the sliding window by two-digit years, and all other dates with four-digit years.</td>
</tr>
<tr>
<td>1</td>
<td>Represent 20th century dates with two-digit years and all other dates with four-digit years.</td>
</tr>
<tr>
<td>2</td>
<td>Represent all dates with two-digit years.</td>
</tr>
<tr>
<td>3</td>
<td>Represent with two-digit years those dates falling within the sliding temporal window defined by <code>startwin</code> and (optionally) <code>endwin</code>. Represent all other dates with four-digit years. When <code>yearopt</code> = 3, <code>startwin</code> and <code>endwin</code> are absolute dates in <code>$HOROLOG</code> format.</td>
</tr>
<tr>
<td>4</td>
<td>Represent all dates with four-digit years. ODBC year option.</td>
</tr>
<tr>
<td>5</td>
<td>Represent with two-digit years all dates falling within the sliding temporal window defined by <code>startwin</code> and (optionally) <code>endwin</code>. Represent all other dates with four-digit years. When <code>yearopt</code>=5, <code>startwin</code> and <code>endwin</code> are relative years.</td>
</tr>
<tr>
<td>6</td>
<td>Represent all dates in the current century with two-digit years and all other dates with four-digit years.</td>
</tr>
</tbody>
</table>

To determine the default year option for your locale, invoke the `GetFormatItem()` NLS class method:

```objectscript
WRITE ##class(%SYS.NLS.Format).GetFormatItem("YearOption")
```

If you omit `yearopt` or specify a `yearopt` value of -1, `$ZDATE` uses the `YearOption` property of the current locale, unless one of the following is true: If `localeopt`=1, the `yearopt` default is the ODBC year option. If `localeopt`=0 or is unspecified and `dformat` is 18, 19, 20, or 21 (Islamic date formats) the `yearopt` default is the ODBC year option (4-digit years); the `YearOption` property value is ignored for Islamic dates.

### startwin

A numeric value that specifies the start of the sliding window during which dates must be represented with two-digit years. See parameter section. You must supply `startwin` when `yearopt` is 3 or 5. `startwin` is not valid with any other `yearopt` values.
When $\text{yearopt} = 3$, $\text{startwin}$ is an absolute date in $\text{SHOROLOG}$ date format that indicates the start date of the sliding window.

When $\text{yearopt} = 5$, $\text{startwin}$ is a numeric value that indicates the start year of the sliding window expressed as the number of years before the current year. The sliding window always begins on January 1st of the year specified in $\text{startwin}$.

$\text{endwin}$

A numeric value that specifies the end of the sliding window during which dates are represented with two-digit years. You may optionally supply $\text{endwin}$ when $\text{yearopt}$ is 3 or 5. $\text{endwin}$ is not valid with any other $\text{yearopt}$ values.

When $\text{yearopt} = 3$, $\text{endwin}$ is an absolute date in $\text{SHOROLOG}$ date format that indicates the end date of the sliding window.

When $\text{yearopt} = 5$, $\text{endwin}$ is a numeric value that indicates the end year of the sliding window expressed as the number of years past the current year. The sliding window always ends on December 31st of the year specified in $\text{endwin}$. If $\text{endwin}$ is not specified, it defaults to December 31st of the year 100 years after $\text{startwin}$.

If $\text{endwin}$ is omitted (or specified as -1) the effective sliding window will be 100 years long. The $\text{endwin}$ value of -1 is a special case that always returns a date value, even when higher and lower $\text{endwin}$ values return $\text{erropt}$. For this reason, it is preferable to omit $\text{endwin}$ when specifying a 100-year window, and to avoid the use of negative $\text{endwin}$ values.

If you supply both $\text{startwin}$ and $\text{endwin}$, the sliding window they specify must not have a duration of more than 100 years.

$\text{mindate}$

An expression that specifies the lower limit of the range of valid dates (inclusive). Can be specified as a $\text{SHOROLOG}$ integer date count (for example, 1/1/2013 is represented as 62823) or a $\text{SHOROLOG}$ string value. You can include or omit the time portion of a $\text{SHOROLOG}$ date string (for example “62823,43200”), but only the date portion of $\text{mindate}$ is parsed. Specifying an $\text{hdate}$ value earlier than $\text{mindate}$ generates a <VALUE OUT OF RANGE> error.

The following are supported $\text{mindate}$ values:

• Positive integer: Most commonly $\text{mindate}$ is specified as a positive integer to establish the earliest allowed date as some date after December 31, 1840. For example, a $\text{mindate}$ of 21550 would establish the earliest allowed date as January 1, 1900. The highest valid value is 2980013 (December 31, 9999).

• 0: specifies the minimum date as December 31, 1840. This is the DateMinimum property default.

• Negative integer -2 or larger: specifies a minimum date counting backwards from December 31, 1840. For example, a $\text{mindate}$ of -14974 would establish the earliest allowed date as January 1, 1800. Negative $\text{mindate}$ values are only meaningful if the DateMinimum property of the current locale has been set to an equal or greater negative number. The lowest valid value is -672045.

• If omitted (or specified as -1), $\text{mindate}$ defaults to the DateMinimum property value for the current locale, unless one of the following is true: If $\text{localeopt}=1$, the $\text{mindate}$ default is 0. If $\text{localeopt}$ is unspecified and $\text{dformat}=3$, the $\text{mindate}$ default is 0. If $\text{localeopt}$ is unspecified and $\text{dformat}$ is 18, 19, 20, or 21 (Islamic date formats) the $\text{mindate}$ default is 0.

You can get and set the DateMinimum property as follows:

```
SET min=##class(%SYS.NLS.Format).GetFormatItem("DateMinimum")
WRITE "Initial DateMinimum value is ",min,!
Permit18thCenturyDates
SET x=##class(%SYS.NLS.Format).SetFormatItem("DateMinimum",-51498)
SET newmin=##class(%SYS.NLS.Format).GetFormatItem("DateMinimum")
WRITE "set DateMinimum value is ",newmin,!!
RestrictTo19thCenturyDates
WRITE $ZDATE(-13000,1,,,,,-14974),!!
ResetDateMinimumToDefault
SET oldmin=##class(%SYS.NLS.Format).SetFormatItem("DateMinimum",min)
WRITE "reset DateMinimum value from ",oldmin," to ",min
```

You may specify $\text{mindate}$ with or without $\text{maxdate}$. Specifying a $\text{mindate}$ larger than $\text{maxdate}$ generates an <ILLEGAL VALUE> error.
**ODBC Date Format (dformat 3)**

The application of the DateMinimum property is governed by the localeopt setting. When localeopt=1 (which is the default for dformat=3) the date minimum is 0, regardless of the current locale setting. Therefore, in ODBC format (dformat=3) the following can be used to specify a date prior to December 31, 1840:

- Specify a mindate earlier than the specified date:
  ```objectscript
  WRITE $ZDATE(-30,3,,,,,,,,,-365)
  ```

- Specify a DateMinimum property value earlier than the specified date and set localeopt=0:
  ```objectscript
  DO ##class(%SYS.NLS.Format).SetFormatItem("DateMinimum",-365)
  WRITE $ZDATE(-30,3,,,,,,,,,0)
  ```

**maxdate**

An expression that specifies the upper limit of the range of valid dates (inclusive). Can be specified as a $HOROLOG integer date count (for example, 1/1/2100 is represented as 94599) or a $HOROLOG string value. You can include or omit the time portion of the $HOROLOG date (for example “94599,43200”), but only the date portion of maxdate is parsed.

If maxdate is omitted or if specified as -1, the maximum date limit is obtained from the DateMaximum property of the current locale, which defaults to the maximum permissible value for the date portion of $HOROLOG: 2980013 (corresponding to December 31, 9999 CE). However, the application of the DateMaximum property is governed by the localeopt setting. When localeopt=1 (which is the default for dformat=3) the date maximum default is the ODBC value (2980013), regardless of the current locale setting. Islamic date formats also take the ODBC default. The maximum date for Thai date format (dformat=13) is $HOROLOG 2781687 which corresponds to 31/12/9999 BE.

Specifying a hdate larger than maxdate generates a <VALUE OUT OF RANGE> error.

Specifying a maxdate larger than 2980013 generates an <ILLEGAL VALUE> error.

You may specify maxdate with or without mindate. Specifying a maxdate smaller than mindate generates an <ILLEGAL VALUE> error.

**erropt**

Specifying a value for this parameter suppresses errors associated with invalid or out of range hdate values. Instead of generating <ILLEGAL VALUE> or <VALUE OUT OF RANGE> errors, the $ZDATE function returns the erropt value.

- Validation: Caché performs canonical numeric conversion on hdate. Parsing of an hdate string halts at the first non-numeric character. Therefore, an hdate string such as 64687AD is the same as 64687. A non-numeric date (including the null string) evaluates to 0. Thus an empty string hdate returns the $HOROLOG initial date: December 31, 1840. However, if hdate does not evaluate to an integer (contains a non-zero fractional number) it generates an <ILLEGAL VALUE> error.

- Range: hdate must evaluate to an integer within the mindate/maxdate range. By default, date values greater than 2980013 or less than 0 generate a <VALUE OUT OF RANGE> error. By setting mindate to a negative number, you can extend the range of valid dates before December 31, 1840. However, for dformat 18, 19, 20, or 21 (Hijri Islamic calendar) dates, any date prior to -445031 generates an <ILLEGAL VALUE> error, even if mindate is set to an earlier date.

The erropt parameter only suppresses errors generated due to invalid or out of range values of hdate. Errors generated due to invalid or out of range values of other parameters will always generate errors whether or not erropt has been supplied. For example, an <ILLEGAL VALUE> error is always generated when $ZDATE specifies a sliding window where endwin is earlier than startwin. Similarly, an <ILLEGAL VALUE> error is generated when maxdate is less than mindate.
Invalid Date Handling with ZDateNull

The behavior of $ZDATE when given an invalid value for $hdate can be set using ZDateNull. To set this behavior for the current process, use the ZDateNull() method of the %SYSTEM.Process class. The system-wide default behavior can be established by setting the ZDateNull property of the Config.Miscellaneous class. $ZDATE can either issue an error, or return a null value.

The system-wide default behavior is configurable. Go to the Management Portal, select [System] > [Configuration] > [Compatibility Settings]. View and edit the current setting of ZDateNull. The default is “false”, meaning that $ZDATE returns an error.

localeopt

This Boolean parameter specifies either the user’s current locale definition or the ODBC locale definition as the source for defaults for the locale-specified parameters dformat, monthlist, yearopt, mindate and maxdate:

- If localeopt=0, all of these parameters take the current locale definition defaults.
- If localeopt=1, all of these parameters take the ODBC defaults.
- If localeopt is not specified, the dformat parameter determine the default for these parameters. If dformat=3, the ODBC defaults are used. If dformat is 18, 19, 20, or 21 the Islamic date format defaults are used, regardless of the current locale definition. For all other dformat values, the current locale definition defaults are used. Refer to the dformat description for further details.

The ODBC locale cannot be changed; it is used to format date strings that are portable between Caché processes that have made different National Language Support (NLS) choices. If localeopt=1, the ODBC locale date definitions are as follows:

- Date format defaults to 3. Therefore, if dformat is undefined or -1, date format 3 is used.
- Date separator defaults to "/". However, date format defaults to 3, which always uses "-" as the date separator.
- Year option defaults to 4 digits.
- Date minimum and maximum: 0 and 2980013 ($HOROLOG date count).
- English month names, month abbreviations, weekday names, and weekday abbreviations are used.

Examples

Date Format Examples

The following example illustrates how $ZDATE returns the various dformat formats for the current date. The yearopt takes the default values. The date separator, and the names and abbreviations of months and days of the week are, of course, locale-dependent. This example uses the current user locale definition:

```
WRITE $ZDATE($HOROLOG), "   default date format",!
WRITE $ZDATE($HOROLOG,1), "   1=American numeric format",!
WRITE $ZDATE($HOROLOG,2), "   2=Month abbreviation format",!
WRITE $ZDATE($HOROLOG,3), "   3=ODBC numeric format",!
WRITE $ZDATE($HOROLOG,4), "   4=European numeric format",!
WRITE $ZDATE($HOROLOG,5), "   5=Month abbreviation format",!
WRITE $ZDATE($HOROLOG,6), "   6=Month abbreviation format",!
WRITE $ZDATE($HOROLOG,7), "   7=Month abbreviation format",!
WRITE $ZDATE($HOROLOG,8), "   8=Numeric format no spaces",!
WRITE $ZDATE($HOROLOG,9), "   9=Month name format",!
WRITE $ZDATE($HOROLOG,10), "  10=Day-of-week format",!
WRITE $ZDATE($HOROLOG,11), "  11=Day abbreviation format",!
WRITE $ZDATE($HOROLOG,12), "  12=Day name format",!
WRITE $ZDATE($HOROLOG,13), "  13=Thai numeric format",!
WRITE $ZDATE($HOROLOG,14), "  14=Day-of-year format",!
WRITE $ZDATE($HOROLOG,15), "  15=European date format",!
WRITE $ZDATE($HOROLOG,16), "  16=Japanese date format",!
WRITE $ZDATE($HOROLOG,17), "  17=Japanese date format with spaces"
```
The following example compares dates with the locale defaulting to the current user locale with dates when `localeopt=1` activates the ODBC locale definition. To make this example more interesting, the current user locale is set to French:

```objectscript
SET fmt=##class(%SYS.NLS.Format).%New("fraw")
WRITE "default: local=",$ZDATE($HOROLOG)," ODBC=",$ZDATE($HOROLOG,,,,,,,,,1),!
WRITE ":-1: local=",$ZDATE($HOROLOG,-1)," ODBC=",$ZDATE($HOROLOG,-1,,,,,,,,,1),!!
FOR x=1:1:17 {
  WRITE x," local=",$ZDATE($HOROLOG,x)," ODBC=",$ZDATE($HOROLOG,x,,,,,,,,,1),! }
```

### Two-digit Year Sliding Window Example

To illustrate how to use an explicit sliding window, suppose you enter the following function call in 1997. The `hdate` of 59461 represents October 19, 2003; the `dformat` of 1 allows it to return two-digit or four-digit years, and the `yearopt` of 5 specifies a sliding window for four-digit years. Because of the `yearopt` setting, the `startwin` and `endwin` are calculated relative to the current year (in this case 1997) by addition and subtraction.

```objectscript
WRITE $ZDATE(59461,1,,5,90,10)
```

The sliding window for displaying the year as two digits extends from 1/1/1907 to 12/31/2006. Thus Caché displays the date as 10/19/03.

### Date Range Example

The following example uses `mindate` and `maxdate` to test for plausible birth dates. The `maxdate` assumes that a birth date cannot be in the future; the `mindate` assumes that no person listed will be more that 124 years old. The dates are specified in `$HOROLOG` format:

```objectscript
PlausibleBirthdate
SET bdateh(1)=62142
SET bdateh(2)=16800
SET bdateh(3)=70000
DO $SYSTEM.Process.ZDateNull(1)
  SET maxdate=$PIECE($HOROLOG,,1)+1
  SET mindate=maxdate-(365.25*124)
FOR x=1:1:3 {
  SET bdate=$ZDATE(bdateh(x),,,,,,mindate,maxdate)
  IF bdate="" {WRITE "Birth date ",bdateh(x)," is out of range",!}
  ELSE {WRITE "Birth date ",bdateh(x)," is ",bdate,!} }
```

Two of the above `$ZDATE` input values fall outside of the date range for a birth date test: 16800 (12/30/1886) is more than 124 years ago and 70000 (08/26/2032) is in the future. By default, these invocations of `$ZDATE` would generate a `<VALUE OUT OF RANGE>` error, but because `ZDateNull(1)` is set, they return the empty string (`""`).

### Notes

#### Invalid Values with `$ZDATE`

You receive a `<FUNCTION>` error in the following conditions:

- If you specify an invalid `dformat` code (an integer value less than -1 or greater than 17, or a non-integer value).
- If you do not specify a `startwin` value when `yearopt` is 3 or 5.

You receive an `<ILLEGAL VALUE>` error under the following conditions:

- If you specify an invalid value for `hdate` and do not either supply an `erropt` value or set `ZDateNull` (as described below).
- If the given month number is greater than the number of month values in `monthlist`.
- If `maxdate` is less than `mindate`.
- If `endwin` is less than `startwin`.
- If `startwin` and `endwin` specify a sliding temporal window whose duration is greater than 100 years.
You receive a <VALUE OUT OF RANGE> error under the following conditions:

- If you specify an \textit{hdate} value that is out of the range of valid dates. For standard Caché this is 0 through 298013. For ISM-compatible Caché this is 1 through 94232. You can use the \texttt{ZDateNull()} method of the \%SYSTEM.Process class to set the date range and invalid date behavior for the current process.

- If you specify an otherwise valid date which is outside the range defined by the values assumed for \textit{maxdate} and \textit{mindate}, and do not supply an \textit{erropt} value.

\textbf{Using $ZDATE$ Instead of Utilities}

Keep the following points in mind when you need to choose between the $ZDATE$ function and a date utility:

- You can use the $ZDATE$ function in place of most existing entry points of the \%DO or \%D utilities.

- You can invoke $ZDATE($HOROLOG,7) directly instead of calling INT ^%D. This provides the current date in “Mmm DD [YY]YY” format.

- $ZDATEH$ and $ZDATE$ are much faster than calling entry points of \%DATE, \%DI or \%DO.

\textbf{See Also}

- JOB command
- $ZDATEH$ function
- $ZDATETIME$ function
- $ZDATETIMEH$ function
- $ZTIME$ function
- $HOROLOG$ special variable
- $ZTIMESTAMP$ special variable
- \%DATE utility, which is documented in the “Legacy Documentation” chapter in \textit{Using InterSystems Documentation}

Caché ObjectScript Reference
**$ZDATEH**

Validates a date and converts it from display format to Caché internal format.

```objectscript
$ZDATEH(date,dformat,monthlist,yearopt,startwin,endwin,mindate,maxdate,erropt,localeopt)
$ZDH(date,dformat,monthlist,yearopt,startwin,endwin,mindate,maxdate,erropt,localeopt)
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>date</strong></td>
<td>An expression that evaluates to a date string in display format. $ZDATEH converts this date string to SHOROLOG format. This can be either an explicit date (specified in various formats) or the string “T” or “t”, representing the current date. The ‘T’ or ‘t’ string can optionally include a signed integer offset. For example “T-7” meaning seven days before the current date. See date below.</td>
</tr>
<tr>
<td><strong>dformat</strong></td>
<td>Optional — An integer code that specifies a date format option for date. If date is “T”, dformat must be 5, 6, 7, 8, 9, or 15. See dformat below.</td>
</tr>
<tr>
<td><strong>monthlist</strong></td>
<td>Optional — A string or the name of a variable that specifies a set of month names. This string must begin with a delimiter character, and its 12 entries must be separated by this delimiter character. See monthlist below.</td>
</tr>
<tr>
<td><strong>yearopt</strong></td>
<td>Optional — An integer code that specifies whether to represent years as two- or four-digit values. See yearopt below.</td>
</tr>
<tr>
<td><strong>startwin</strong></td>
<td>Optional — The start of the sliding window during which dates must be represented with two-digit years. See startwin below.</td>
</tr>
<tr>
<td><strong>endwin</strong></td>
<td>Optional — The end of the sliding window during which dates are represented with two-digit years. See endwin below.</td>
</tr>
<tr>
<td><strong>mindate</strong></td>
<td>Optional — The lower limit of the range of valid date dates. Specified as a SHOROLOG integer date count, with 0 representing December 31, 1840. Can be specified as a positive or negative integer. See mindate below.</td>
</tr>
<tr>
<td><strong>maxdate</strong></td>
<td>Optional — The upper limit of the range of valid dates. Specified as a SHOROLOG integer date count. See maxdate below.</td>
</tr>
<tr>
<td><strong>erropt</strong></td>
<td>Optional — An expression to return when date is invalid. Specifying a value for this parameter suppresses error codes associated with invalid or out of range date values. Instead of issuing an error message, $ZDATEH returns erropt. See erropt below.</td>
</tr>
</tbody>
</table>
| **localeopt** | Optional — A boolean flag that specifies which locale to use for the dformat, monthlist, yearopt, mindate and maxdate default values, and other date characteristics, such as the date separator character: 

- localeopt=0: the current locale property settings determine these parameter defaults.
- localeopt=1: the ODBC standard locale determines these parameter defaults.
- localeopt not specified: the dformat value determines these parameter defaults. If dformat=3, ODBC defaults are used. Japanese and Islamic date dformat values use their own defaults. For all other dformat values, current locale property settings are used as defaults. See localeopt below. |

Omitted parameters between specified parameter values are indicated by placeholder commas. Trailing placeholder commas are not required, but are permitted. Blank spaces are permitted between the commas that indicate an omitted parameter.
Description

The $ZDATEH function validates a specified date and converts it from any of the formats supported by the $ZDATE function to $SHOROLOG format. The exact action $ZDATEH performs depends on the parameters you use.

Simple $ZDATEH Format

$ZDATEH(date) converts a date in the form MM/DD/[YY]YY to the first integer in the $SHOROLOG format. (The $SHOROLOG format consists of two integers: the first integer is a date, the second integer is a time.) Two or four digits may be specified for years in the range 1900 to 1999. Four digits must be specified for years before 1900 or after 1999.

Customizable $ZDATEH Format

$ZDATEH(date,dformat,monthlist,yearopt,startwin,endwin,mindate,maxdate,erropt) converts a date in the specified dformat to $SHOROLOG format. The dformat, monthlist, yearopt, startwin, endwin, mindate, maxdate and erropt values are identical to the values used by $ZDATE. However, when you use a dformat of 5, 6, 7, 8, or 9, $ZDATEH recognizes and converts a date in any of the external date formats defined for dformat codes 1, 2, 3, 5, 6, 7, 8, and 9. (But not dformat code 4.) It also recognizes a special relative date format that consists of a string beginning with the letter T or t (indicating “today.”) optionally followed by a plus (+) or a minus (-) sign, and an integer number of days after or before the current date.

Parameters

date

The date you want converted to $SHOROLOG format, specified as a quoted string. This can be an explicit date, or the implicit current date, represented by the string “T” or “t”.

An explicit date must be specified in one of the formats supported by dformat. The permitted format(s) depends on the dformat parameter. If dformat is not specified or is 1, 2, 3, or 4, only one date format is permitted. If dformat is 5, 6, 7, 8, 9 or 15, multiple date formats are permitted.

If dformat is 5, 6, 7, 8, or 9, $ZDATEH accepts all unambiguous American date formats. If dformat is 15, $ZDATEH accepts all unambiguous European date formats. A list of valid date formats is provided below. Note that $ZDATEH does not use dformat code 4 is not a valid American date format because $ZDATEH cannot differentiate between 02/03/02 (meaning February 3, 2002) and the European 02/03/02 (meaning March 2, 2002). If you specify a date in a non-permitted format, or a nonexistent date (such as February 31, 2002), $ZDATEH generates an <ILLEGAL VALUE> error code. ($ZDATEH does check for leap year dates, permitting Feb. 29, 2004 but not Feb. 29, 2003.)

In the Russian, Ukrainian, and Czech locales, a date must be specified with periods, rather than slashes, as date part separators: DD.MM.YYYY.

An implicit date is specified as a string consisting of the letter “T” or “t”, indicating the current date (today). This string can optionally include a plus or minus sign and an integer, which specify the number of days offset from the current date. For example, “t+9” (nine days after the current date) or “t-12” (twelve days before the current date). Implicit dates are only permitted if dformat is 5, 6, 7, 8, 9, or 15. The only permitted implicit date forms are “T” (or “t”), and “T” (or “t”) followed by a sign and integer. Caché generates an <ILLEGAL VALUE> error if you specify a noninteger number, an arithmetic expression, an integer without a sign, or a sign without an integer. “T+0” and “T-0” are permitted, and return the current date. Caché generates a <VALUE OUT OF RANGE> error if you specify an offset that would result in a $SHOROLOG date beyond the range of valid dates.

By default, the earliest valid date is December 31, 1840 (0 in internal $SHOROLOG representation). Dates are limited to positive integers by default because the DateMinimum property defaults to 0. You can specify earlier dates as negative integers, provided the DateMinimum property of the current locale is set to a greater or equal negative integer. The lowest valid DateMinimum value is -672045, which corresponds to January 1, 0001. Caché uses the proleptic Gregorian calendar, which projects the Gregorian calendar back to “Year 1”, in conformance with the ISO 8601 standard. This is, in part, because the Gregorian calendar was adopted at different times in different countries. For example, much of continental...
Europe adopted it in 1582; Great Britain and the United States adopted it in 1752. Thus Caché dates prior to your local adoption of the Gregorian calendar may not correspond to historical dates that were recorded based on the local calendar then in effect. For further details on dates prior to 1840, refer to the `mindate` parameter.

**dformat**

Format for the date. Valid values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Get effective <code>dformat</code> value from the <code>DateFormat</code> property of the current locale. This is the default behavior if you do not specify <code>dformat</code>.</td>
</tr>
<tr>
<td>1</td>
<td><code>MM/DD/[YY]YYYY</code> (07/01/97 or 03/27/2002) — American numeric format. You must specify the correct dateseparator character (/ or .) for the current locale.</td>
</tr>
<tr>
<td>2</td>
<td><code>DD Mmm [YY]YYYY</code> (01 Jul 97)</td>
</tr>
<tr>
<td>3</td>
<td><code>[YY]YY-MM-DD</code> (1997-07-01) - ODBC format</td>
</tr>
<tr>
<td>4</td>
<td><code>DD/MM/[YY]YYYY</code> (01/07/97 or 03/27/2002) — European numeric format. You must specify the correct dateseparator character (/ or .) for the current locale.</td>
</tr>
<tr>
<td>5</td>
<td><code>Mmm D, YYYY</code> (Jul 1, 1997) or any unambiguous American date format.</td>
</tr>
<tr>
<td>6</td>
<td><code>Mmm D YYYY</code> (Jul 1 1997) or any unambiguous American date format.</td>
</tr>
<tr>
<td>7</td>
<td><code>Mmm DD [YY]YYYY</code> (Jul 01 1997) or any unambiguous American date format.</td>
</tr>
<tr>
<td>8</td>
<td><code>[YY]YYMMDD</code> (19970701) - Numeric format, or any unambiguous American date format.</td>
</tr>
<tr>
<td>9</td>
<td><code>Mmmmm D, YYYY</code> (July 1, 1997), or any unambiguous American date format.</td>
</tr>
<tr>
<td>13</td>
<td><code>[D]D/[M]M/YYYY</code> (1/7/2549 or 27/11/2549) — Thai date format. Day and month are identical to European usage, except no leading zeros. The year is the Buddhist Era (BE) year, calculated by adding 543 years to the Gregorian year.</td>
</tr>
<tr>
<td>15</td>
<td><code>DD/MM/[YY]YYYY</code> or <code>YYYY-MM-DD</code> or any unambiguous European date format with any dateseparator character, or <code>YYYYMMDD</code> with no date separators. The dateseparator character may be any non-alphanumeric character, including blank spaces, regardless of the dateseparator character specified in the current locale. Also accepts <code>monthlist</code> names and “T”.</td>
</tr>
<tr>
<td>16</td>
<td><code>YYYYc[M]Mc[D]Dc</code> — Japanese date format. Year, month, and day numbers are the same as other date formats; leading zeros are omitted. The Japanese characters for “year”, “month”, and “day” (shown here as c) are inserted after the year, month, and day numbers. These characters are <code>Year=$CHAR(24180)</code>, <code>Month=$CHAR(26376)</code>, and <code>Day=$CHAR(26085)</code>.</td>
</tr>
<tr>
<td>17</td>
<td><code>YYYYc [M]Mc [D]Dc</code> — Japanese date format. Same as <code>dformat</code> 16, except that a blank space is inserted after the “year” and “month” Japanese characters.</td>
</tr>
<tr>
<td>18</td>
<td><code>[D]D Mmmmm YYYY</code> — Tabular Hijri (Islamic) date format with full month name. Day leading zeros are omitted; year leading zeros are included. Caché date -445031 (07/19/0622 C.E.) = 1 Muharram 0001.</td>
</tr>
<tr>
<td>19</td>
<td><code>[D]D [M]YYYY</code> — Tabular Hijri (Islamic) date format with month number. Day and month leading zeros are omitted; year leading zeros are included. Caché date -445031 (07/19/0622 C.E.) = 1 1 0001.</td>
</tr>
<tr>
<td>20</td>
<td><code>[D]D Mmmmm YYYY</code> — Observed Hijri (Islamic) date format with full month name. Defaults to Tabular Hijri (<code>dformat</code> 18). To override tabular calculation, use the class <code>%Calendar.Hijri</code> to add observations of new moon crescents.</td>
</tr>
<tr>
<td>Value</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>21</td>
<td>[D]DM YYYY — Observed Hijri (Islamic) date format with month number. Defaults to Tabular Hijri (dformat 19). To override tabular calculation, use the class %Calendar.Hijri to add observations of new moon crescents.</td>
</tr>
</tbody>
</table>

Where:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YYYY</td>
<td>YYYY is a four-digit year. [YY]YY is a two-digit year if the date falls within the active window for two-digit years; otherwise it is a four-digit years. You must supply the year value when using date formats (dformat) 1 through 4; these date formats do not supply a missing year value. Date formats 5 through 9 assume the current year if the date you specify does not include a year.</td>
</tr>
<tr>
<td>MM</td>
<td>Two-digit month.</td>
</tr>
<tr>
<td>D</td>
<td>One-digit day if the day number &lt;10. Otherwise, two digits.</td>
</tr>
<tr>
<td>DD</td>
<td>Two-digit day.</td>
</tr>
<tr>
<td>Mmm</td>
<td>Month abbreviation extracted from the MonthAbbr property of the current locale. The default values in English are: “Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec”. Or an alternate month abbreviation (or name of any length) extracted from an optional list specified as the monthlist parameter to $ZDATEH.</td>
</tr>
<tr>
<td>MMMMM</td>
<td>Full name of the month as specified by the MonthName property of the current locale. The default values in English are: “January February March ... November December”.</td>
</tr>
</tbody>
</table>

**dformat Default**

If you omit dformat or set it to -1, the dformat default depends on the localeopt parameter and the NLS DateFormat property:

- If localeopt=1 the dformat default is ODBC format. The monthlist, yearopt, mindate and maxdate parameter defaults are also set to ODBC format. This is the same as setting dformat=3.
- If localeopt=0 or is unspecified, the dformat default is taken from the NLS DateFormat property. If DateFormat=3, the dformat default is ODBC format. However, DateFormat=3 does not affect the monthlist, yearopt, mindate and maxdate parameter defaults, which are as specified in the current NLS locale definition.

To determine the default date properties for your locale, invoke the GetFormatItem() NLS class method:

```objectscript
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DateFormat"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DateSeparator")
```

$ZDATEH will use the value of the DateSeparator property of the current locale (either / or .) as the delimiter between months, days, and the year when dformat=1 or 4.

European date format (dformat=4, DD/MM/YYYY order) is the default for many (but not all) European languages, including British English, French, German, Italian, Spanish, and Portuguese (which use a “/” DateSeparator character), as well as Czech (csyw), Russian (rusw), Slovak (skyw), Slovenian (svnw), and Ukrainian (ukrw) (which use a “.” DateSeparator character). For further details on default date formats for supported locales, refer to Dates in Using Caché ObjectScript.

**dformat Settings**

If dformat is 3 (ODBC format date), ODBC format defaults are also used for the monthlist, yearopt, mindate and maxdate parameter defaults. The date separator will always be a “.”. Current locale defaults are ignored.
If `dformat` is 16 or 17 (Japanese date formats), the date format is independent of the locale setting. To use Japanese-format dates you must have a Unicode Caché instance. On an 8-bit Caché instance specifying `dformat` is 16 or 17 causes a `<ERWIDECHAR>` error.

If `dformat` is 18, 19, 20, or 21 (Islamic date formats) and `localeopt` is unspecified, parameters default to Islamic defaults, rather than current locale defaults. The `monthlist` parameter defaults to Arabic month names transliterated with Latin characters. The `yearopt`, `mindate` and `maxdate` parameters default to ODBC defaults. The date separator defaults to the Islamic default (a space), not the ODBC default or the current locale `DateSeparator` property value. If `localeopt`=0 current locale property defaults are used for these parameters. If `localeopt`=1 ODBC defaults are used for these parameters.

`monthlist`

An expression that resolves to a string of month names or month name abbreviations, separated by a delimiter character. The names in `monthlist` replace the default month abbreviation values from the `MonthAbbr` property or the month name values from the `MonthName` property of the current locale.

`monthlist` is valid only if `dformat` is 2, 5, 6, 7, 8, 9, 15, 18, or 20. If `dformat` is any other value `$ZDATEH` ignores `monthlist`.

The `monthlist` string has the following format:

- The first character of the string is a delimiter character (usually a space). The same delimiter must appear before the first month name and between each month name in `monthlist`. You can specify any single-character delimiter; this delimiter must be specified between the month, day, and year portions of the specified `date` value, which is why a space is usually the preferred character.

- The month names string should contain twelve delimited values, corresponding to January through December. It is possible to specify more or less than twelve month names, but if there is no month name corresponding to the month in `date` an `<ILLEGAL VALUE>` error is generated.

If you omit `monthlist` or specify a `monthlist` value of -1, `$ZDATEH` uses the list of month names defined in the `MonthAbbr` or `MonthName` property of the current locale, unless one of the following is true: If `localeopt`=1, the `monthlist` default is the ODBC month list (in English). If `localeopt` is unspecified and `dformat` is 18 or 20 (Islamic date formats) the `monthlist` default is the Islamic month list (Arabic names expressed using Latin letters), ignoring the `MonthAbbr` or `MonthName` property value.

To determine the default month names and month abbreviations for your locale, invoke the `GetFormatItem()` NLS class method:

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("MonthName"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("MonthAbbr"),!
```

`yearopt`

A numeric code that specifies whether to represent years as either two-digit values or four-digit values. Valid values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| -1    | Get effective `yearopt` value from `YearOption` property of current locale which defaults to a value of 0. This is the default behavior if you do not specify `yearopt`.
| 0     | Represent 20th century dates (1900 through 1999) with two-digit years, unless a process-specific sliding window (established via the `%DATE` utility) is in effect. If such a window is in effect, represent only those dates falling within the sliding window by two-digit years. Represent all dates falling outside the 20th century or outside the process-specific sliding window by four-digit years.
<p>| 1     | Represent 20th century dates with two-digit years and all other dates with four-digit years, regardless of any sliding temporal window in effect. |</p>
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Represent all dates with two-digit years, regardless of any sliding temporal window in effect. All dates are assumed to be in the 20th century. Because this option deletes two digits from four-digit years, its use results in a nonreversible loss of century information. (This loss may be trivial if all dates are in the same century).</td>
</tr>
<tr>
<td>3</td>
<td>Represent with two-digit years those dates falling within the sliding temporal window defined by startwin and (optionally) endwin. Represent all other dates with four-digit years. When yearopt=3, startwin and endwin are absolute dates in $HOROLOG format.</td>
</tr>
<tr>
<td>4</td>
<td>Represent all dates with four-digit years. Dates input with two-digit years are rejected as invalid.</td>
</tr>
<tr>
<td>5</td>
<td>Represent with two-digit years all dates falling within the sliding temporal window defined by startwin and (optionally) endwin. Represent all other dates with four-digit years. When yearopt=5, startwin and endwin are relative years.</td>
</tr>
<tr>
<td>6</td>
<td>Represent all dates in the current century with two-digit years and all other dates with four-digit years.</td>
</tr>
</tbody>
</table>

If you omit yearopt or specify a yearopt value of -1, $ZDATEH uses the YearOption property of the current locale, unless one of the following is true: If localeopt=1, the yearopt default is the ODBC year option. If localeopt=0 or is unspecified and dformat is 18, 19, 20, or 21 (Islamic date formats) the yearopt default is the ODBC year option (4-digit years); the YearOption property value is ignored for Islamic dates.

**startwin**

A numeric value that specifies the start of the sliding window during which dates must be represented with two-digit years. You must supply startwin when you use a yearopt of 3 or 5. startwin is not valid with any other yearopt values.

When yearopt=3, startwin is an absolute date in $HOROLOG date format that indicates the start date of the sliding window.

When yearopt=5, startwin is a numeric value that indicates the start year of the sliding window expressed in the number of years before the current year. The sliding window always begins on the first day of the year (January 1) specified in startwin.

**endwin**

A numeric value that specifies the end of the sliding window during which dates are represented with two-digit years. You may optionally supply endwin when yearopt is 3 or 5. endwin is not valid with any other yearopt values.

When yearopt=3, endwin is an absolute date in $HOROLOG date format that indicates the end date of the sliding window.

When yearopt=5, endwin is a numeric value that indicates the end year of the sliding window expressed as the number of years past the current year. The sliding window always ends on the last day of the year (December 31) of the year specified in endwin or of the implied end year (if you omit endwin).

If endwin is omitted (or specified as -1) the effective sliding window will be 100 years long. The endwin value of -1 is a special case that always returns a date value, even when higher and lower endwin values return erropt. For this reason, it is preferable to omit endwin when specifying a 100-year window, and to avoid the use of negative endwin values.

If you supply both startwin and endwin, the sliding window they specify must not have a duration of more than 100 years.

**mindate**

An expression that specifies the lower limit of the range of valid dates (inclusive). Can be specified as a $HOROLOG integer date count (for example, 1/1/2013 is represented as 62823) or a $HOROLOG string value. You can include or omit the time portion of the $HOROLOG date (for example “62823,43200”), but only the date portion of mindate is parsed. Specifying a date value earlier than mindate generates a <VALUE OUT OF RANGE> error.

The following are supported mindate values:
Positive integer: Most commonly *mindate* is specified as a positive integer to establish the earliest allowed date as some date after December 31, 1840. For example, a *mindate* of 21550 would establish the earliest allowed date as January 1, 1900. The highest valid value is 2980013 (December 31, 9999).

0: specifies the minimum date as December 31, 1840. This is the *DateMinimum* property default.

Negative integer -2 or larger: specifies a minimum date counting backwards from December 31, 1840. For example, a *mindate* of -14974 would establish the earliest allowed date as January 1, 1800. Negative *mindate* values are only meaningful if the *DateMinimum* property of the current locale has been set to an equal or greater negative number. The lowest valid value is -672045.

If omitted (or specified as -1), *mindate* defaults to the *DateMinimum* property value for the current locale, unless one of the following is true: If *localeopt*=1, the *mindate* default is 0. If *localeopt* is unspecified and *dformat*=3, the *mindate* default is 0. If *localeopt* is unspecified and *dformat* is 18, 19, 20, or 21 (Islamic date formats) the *mindate* default is 0.

You can get and set the *DateMinimum* property as follows:

```
SET min=#class(%SYS.NLS.Format).GetFormatItem("DateMinimum")
WRITE "initial DateMinimum value is ",min,!
Set18thCenturyDates
SET x=#class(%SYS.NLS.Format).SetFormatItem("DateMinimum",-51498)
WRITE "set DateMinimum value is ",newmin,!!
RestrictTo19thCenturyDates
WRITE $ZDATEH("05/29/1805",1,,,,,-14974),!!
ResetDateMinimumToDefault
SET oldmin=#class(%SYS.NLS.Format).SetFormatItem("DateMinimum",min)
WRITE "reset DateMinimum value from ",oldmin," to ",min
```

You may specify *mindate* with or without *maxdate*. Specifying a *mindate* larger than *maxdate* generates an <ILLEGAL VALUE> error.

**maxdate**

An expression that specifies the upper limit of the range of valid dates (inclusive). Can be specified as a $HOROLOG integer date count (for example, 1/1/2100 is represented as 94599) or a $HOROLOG string value. You can include or omit the time portion of the $HOROLOG date (for example “94599,43200”), but only the date portion of *maxdate* is parsed.

If *maxdate* is omitted or if specified as -1, the maximum date limit is obtained from the *DateMaximum* property of the current locale, which defaults to the maximum permissible value for the date portion of $HOROLOG: 2980013 (corresponding to December 31, 9999 CE). However, the application of the *DateMaximum* property is governed by the *localeopt* setting. When *localeopt*=1 (which is the default for *dformat*=3) the date maximum default is the ODBC value (2980013), regardless of the current locale setting. Islamic date formats also take the ODBC default. The maximum date for Thai date format (*dformat*=13) is 31/12/9999 BE, which corresponds to $HOROLOG 2781687.

Specifying a *date* larger than *maxdate* generates a <VALUE OUT OF RANGE> error.

Specifying a *maxdate* larger than 2980013 generates an <ILLEGAL VALUE> error.

You may specify *maxdate* with or without *mindate*. Specifying a *maxdate* smaller than *mindate* generates an <ILLEGAL VALUE> error.

**erropt**

Specifying a value for this parameter suppresses errors associated with invalid or out of range *date* values. Instead of generating <ILLEGAL VALUE> or <VALUE OUT OF RANGE> errors, the $ZDATEH function returns the erropt value.

Caché performs standard numeric evaluation on *date*, which must evaluate to an integer within the *mindate/maxdate* range. Thus, 7, "7", 47, 0007, 7.0, "7 dwarves", and -7 all evaluate to the same date value: 01/07/1841. By default, values greater than 2980013 or less than 0 generate a <VALUE OUT OF RANGE> error. Fractional values generate an <ILLEGAL VALUE> error.
error. Non-numeric strings (including the null string) evaluate to 0, and thus return the $HOROLOG initial date: 12/31/1840.

The erropt parameter only suppresses errors generated due to invalid or out of range values of date. Errors generated due to invalid or out of range values of other parameters will always generate errors whether or not erropt has been supplied. For example, an <ILLEGAL VALUE> error is always generated when $ZDATEH specifies a sliding window where endwin is earlier than startwin. Similarly, an <ILLEGAL VALUE> error is generated when maxdate is less than mindate.

localeopt

This Boolean parameter specifies either the user’s current locale definition or the ODBC locale definition as the source for defaults for the locale-specified parameters dformat, monthlist, yearopt, mindate and maxdate:

- If localeopt=0, all of these parameters take the current locale definition defaults.
- If localeopt=1, all of these parameters take the ODBC defaults.
- If localeopt is not specified, the dformat parameter determine the default for these parameters. If dformat=3, the ODBC defaults are used. If dformat is 18, 19, 20, or 21 the Islamic date format defaults are used, regardless of the current locale definition. For all other dformat values, the current locale definition defaults are used. Refer to the dformat description for further details.

The ODBC locale cannot be changed; it is used to format date strings that are portable between Caché processes that have made different National Language Support (NLS) choices. If localeopt=1, the ODBC locale date definitions are as follows:

- Date format defaults to 3. Therefore, if dformat is undefined or -1, date format 3 is used.
- Date separator defaults to "/". However, date format defaults to 3, which always uses "-" as the date separator.
- Year option defaults to 4 digits.
- Date minimum and maximum: 0 and 2980013 ($HOROLOG date count).
- English month names, month abbreviations, weekday names, and weekday abbreviations are used.

Examples

The following example returns the $HOROLOG date for June 12, 1983:

```
WRITE $ZDATEH("06/12/83")
```

returns 52027.

The following example returns the $HOROLOG date for June 12, 1902 (which may not have not been your intent):

```
WRITE $ZDATEH("06/12/02")
```

returns 22442.

Note: Two-digit years, by default, are considered 20th Century dates; for 21st Century dates, specify a four-digit year, or change the two-digit sliding window by specifying the yearopt, startwin and endwin parameters. This sliding window can also be set for your locale.

The following example shows how the dformat parameter is used to permit multiple date entry formats:

```
WRITE $ZDATEH("November 2, 1954",5)
WRITE $ZDATEH("Nov 2, 1954",5)
WRITE $ZDATEH("Nov. 2 1954",5)
WRITE $ZDATEH("11/2/1954",5)
WRITE $ZDATEH("11.02.54",5)
WRITE $ZDATEH("11 02 1954",5)
```

all return 41578.
In the following examples, suppose the current date is January 16, 2007:

```
WRITE $HOROLOG
```

returns 60646,37854, the first integer of which is the current date (the second integer is the current time, in elapsed seconds).

The next example uses the “T” date to return today’s date (here, January 16, 2007):

```
WRITE $ZDATEH("T",5)
```

returns 60646.

The next examples returns the current date with an offset of plus 2 days and minus 2 days:

```
WRITE $ZDATEH("T+2",5)
WRITE $ZDATEH("T-2",5)
```

returns 60648 and 60644.

The final example illustrates that when no year is specified, $ZDATEH assumes the current year (in this case, 2007):

```
WRITE $ZDATEH("25 Nov",5)
```

returns 60959.

**Notes**

*Invalid Values with $ZDATEH*

You receive a <FUNCTION> error in the following conditions:

- If you specify an invalid dformat code (an integer other than -1, 1, 2, 3, 4, 5, 6, 7, 8, 9, or 15, a zero, or a noninteger value)
- If you specify an invalid yearopt code (an integer less than -1 or greater than 6, a value of zero, or a noninteger value)
- If you do not specify a startwin value when yearopt is 3 or 5

You receive a <ILLEGAL VALUE> error under the following conditions:

- If you specify an invalid value for any date unit (day, month, or year). If specified, the erropt value is returned rather than issuing an <ILLEGAL VALUE>.
- If you specify excess leading zeros for any date unit (day, month, or year) in an ODBC date. For example, you can represent the February 3, 2007 as “2007–2–3” or “2007–02–03”, but will receive an <ILLEGAL VALUE> for “2007–002–03”. If specified, the erropt value is returned rather than issuing an <ILLEGAL VALUE>.
- If the given month number is greater than the number of month values in monthlist.
- If maxdate is less than mindate.
- If endwin is less than startwin.
- If startwin and endwin specify a sliding temporal window whose duration is greater than 100 years.

You receive a <VALUE OUT OF RANGE> error under the following conditions:

- If you specify a date (or an offset to “T”) which is earlier than Dec. 31, 1840 or later than Dec. 31, 9999, and do not supply an erropt value
- If you specify an otherwise valid date (or an offset to “T”) which is outside the range of mindate and maxdate and do not supply an erropt value.
Acceptable Date Formats with dformat 5 through 9 and 15

The $ZDATEH dformat date formats 5 through 9 accept any American format date value that is unambiguous. $ZDATEH dformat date format 15 accepts any European format date value that is unambiguous. These date formats assume the current year if the date you specify does not include a year.

The following formats are supported:

<table>
<thead>
<tr>
<th>American Formats: dformat 5, 6, 7, 8, or 9</th>
<th>European Formats: dformat 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM/DD</td>
<td>DD/MM</td>
</tr>
<tr>
<td>MM-DD</td>
<td>DD-MM</td>
</tr>
<tr>
<td>MM DD</td>
<td>DD MM</td>
</tr>
<tr>
<td>MM/DD/YY</td>
<td>DD/MM/YY</td>
</tr>
<tr>
<td>MM-DD-YY</td>
<td>DD-MM-YY</td>
</tr>
<tr>
<td>MM DD YY</td>
<td>DD MM YY</td>
</tr>
<tr>
<td>MM/DD/YYYY</td>
<td>DD/MM/YYYY</td>
</tr>
<tr>
<td>MM-DD-YYYY</td>
<td>DD-MM-YYYY</td>
</tr>
<tr>
<td>MM DD YYYY</td>
<td>DD MM YYYY</td>
</tr>
<tr>
<td>YYYYMMDD</td>
<td>YYYYMMDD</td>
</tr>
<tr>
<td>YYMMDD</td>
<td>YYMMDD</td>
</tr>
<tr>
<td>YYYY-MM-DD</td>
<td>YYYY-MM-DD</td>
</tr>
<tr>
<td>YYYY MM DD</td>
<td>YYYY MM DD</td>
</tr>
<tr>
<td>Mmm D</td>
<td>Mmm D</td>
</tr>
<tr>
<td>Mmm D, YY</td>
<td>Mmm D, YY</td>
</tr>
<tr>
<td>Mmm D, YYYY</td>
<td>Mmm D, YYYY</td>
</tr>
<tr>
<td>Mmm D YY</td>
<td>Mmm D YY</td>
</tr>
<tr>
<td>Mmm D YYYY</td>
<td>Mmm D YYYY</td>
</tr>
<tr>
<td>Mmm DD</td>
<td>Mmm DD</td>
</tr>
<tr>
<td>Mmm DD YY</td>
<td>Mmm DD YY</td>
</tr>
<tr>
<td>Mmm DD YYYY</td>
<td>Mmm DD YYYY</td>
</tr>
<tr>
<td>DD Mmm</td>
<td>DD Mmm</td>
</tr>
<tr>
<td>DD Mmm YY</td>
<td>DD Mmm YY</td>
</tr>
<tr>
<td>DD Mmm YYYY</td>
<td>DD Mmm YYYY</td>
</tr>
<tr>
<td>DD-Mmm</td>
<td>DD-Mmm</td>
</tr>
<tr>
<td>DD-Mmm-YY</td>
<td>DD-Mmm-YY</td>
</tr>
<tr>
<td>DD-Mmm-YYYY</td>
<td>DD-Mmm-YYYY</td>
</tr>
<tr>
<td>YYYY Mmm DD</td>
<td>YYYY Mmm DD</td>
</tr>
</tbody>
</table>

MMDD is not an implemented format.
Using $ZDATEH Instead of Utilities

Keep the following points in mind when you need to choose between the $ZDATEH function and a date utility:

- You can use the $ZDATEH function in place of the existing entry points of the %DATE and %DI utility.
- $ZDATEH and $ZDATE are much faster than calling entry points of %DATE, %DI or %DO.

See Also

- JOB command
- $ZDATE function
- $ZDATETIME function
- $ZDATETIMEH function
- $ZTIME function
- $HOROLOG special variable
- $ZTIMESTAMP special variable
- %DATE utility, which is documented in the “Legacy Documentation” chapter in Using InterSystems Documentation
$ZDATETIME

Validates a date and time and converts it from internal format to the specified display format.

```objectscript
$ZDATETIME(hdatetime, dformat, tformat, precision, monthlist, yearopt, startwin, endwin, mindate, maxdate, erropt, localeopt)
$ZDT(hdatetime, dformat, tformat, precision, monthlist, yearopt, startwin, endwin, mindate, maxdate, erropt, localeopt)
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hdatetime</code></td>
<td>The date and time value, specified in internal date and time format. See hdatetime below.</td>
</tr>
<tr>
<td><code>dformat</code></td>
<td>Optional — An integer code specifying the format for the returned date value. See dformat below.</td>
</tr>
<tr>
<td><code>tformat</code></td>
<td>Optional — An integer code specifying the format for the returned time value. See tformat below.</td>
</tr>
<tr>
<td><code>precision</code></td>
<td>Optional — An integer specifying the number of decimal places of precision (fractional seconds) for the returned time value. See precision below.</td>
</tr>
<tr>
<td><code>monthlist</code></td>
<td>Optional — A string or the name of a variable that specifies a set of month names. This string must begin with a delimiter character, and its 12 entries must be separated by this delimiter character. See monthlist below.</td>
</tr>
<tr>
<td><code>yearopt</code></td>
<td>Optional — An integer code that specifies whether to represent years as two- or four-digit values. See yearopt below.</td>
</tr>
<tr>
<td><code>startwin</code></td>
<td>Optional — The start of the sliding window during which dates are represented with two-digit years. See startwin below.</td>
</tr>
<tr>
<td><code>endwin</code></td>
<td>Optional — The end of the sliding window during which dates are represented with two-digit years. See endwin below.</td>
</tr>
<tr>
<td><code>mindate</code></td>
<td>Optional — The lower limit of the range of valid dates. Specified as a $HOROLOG integer date count, with 0 representing December 31, 1840. Can be specified as a positive or negative integer. See mindate below.</td>
</tr>
<tr>
<td><code>maxdate</code></td>
<td>Optional — The upper limit of the range of valid dates, specified as an integer $HOROLOG date count. See maxdate below.</td>
</tr>
<tr>
<td><code>erropt</code></td>
<td>Optional — An expression to return when <code>hdatetime</code> is invalid. Specifying a value for this parameter suppresses error codes associated with invalid or out of range <code>hdatetime</code> values. Instead of issuing an error message, $ZDATETIME returns <code>erropt</code>. See erropt below.</td>
</tr>
<tr>
<td><code>localeopt</code></td>
<td>Optional — A boolean flag that specifies which locale to use for the <code>dformat</code>, <code>tformat</code>, <code>monthlist</code>, <code>yearopt</code>, <code>mindate</code> and <code>maxdate</code> default values, and other date and time characteristics: <code>localeopt</code>=0: the current locale property settings determine these parameter defaults. <code>localeopt</code>=1: the ODBC standard locale determines these parameter defaults. <code>localeopt</code> not specified: the <code>dformat</code> value determines these parameter defaults. If <code>dformat</code>=3, ODBC defaults are used; otherwise current locale property settings are used. See localeopt below.</td>
</tr>
</tbody>
</table>
Omitted parameters between specified parameter values are indicated by placeholder commas. Trailing placeholder commas are not required, but are permitted. Blank spaces are permitted between the commas that indicate an omitted parameter.

**Description**

$ZDATETIME validates a specified date and time and converts them from $HOROLOG or $ZTIMESTAMP internal format to one of several alternative date and time display formats. The exact value returned depends on the parameters you specify.

- $ZDATETIME(hdatetime) returns the date and time in the default display format for the current locale.
- $ZDATETIME(hdatetime,dformat,tformat,precision,monthlist,yearopt,startwin,endwin,mindate,maxdate) returns the date and time in the display format specified by dformat and tformat, further defined by the other parameters you specify. The range of valid dates may be restricted by the mindate and maxdate parameters.

**Parameters**

**hdatetime**

The date and time, specified as an internal format value. Caché internal format represents dates as a count of days from an arbitrary starting point (Dec. 31, 1840), and represents times as a count of seconds in the current day. The hdatetime value must be a string in one of the following formats:

- **$HOROLOG**: two unsigned integers separated by comma. The first is an integer specifying the date (in days), the second is an integer specifying the time (in seconds).
- **$ZTIMESTAMP**: two unsigned numbers separated by comma: the first is an integer specifying the date (in days), the second is a number specifying the time (in seconds and fractions of a second). The time value can have up to nine digits of precision (fractional seconds) to the right of the decimal point.

You can specify hdatetime as a string value, a variable, or an expression.

If hdatetime specifies only the date portion value and no comma, only the date is returned. If hdatetime specifies the date portion value followed by a comma, but no time value, the system supplies a time value of 00:00:00.

By default, the earliest valid hdatetime date is 0 (December 31, 1840). Dates are limited to positive integers by default because the DateMinimum property defaults to 0. You can specify earlier dates as negative integers, provided the DateMinimum property of the current locale is set to a greater or equal negative integer. The lowest valid DateMinimum value is -672045, which corresponds to January 1, 0001. Caché uses the proleptic Gregorian calendar, which projects the Gregorian calendar back to “Year 1”, in conformance with the ISO 8601 standard. This is, in part, because the Gregorian calendar was adopted at different times in different countries. For example, much of continental Europe adopted it in 1582; Great Britain and the United States adopted it in 1752. Thus Caché dates prior to your local adoption of the Gregorian calendar may not correspond to historical dates that were recorded based on the local calendar then in effect. For further details on dates prior to 1840, refer to the mindate parameter.

Invalid and out-of-range hdatetime values and resulting errors are described in the erropt parameter.

**dformat**

Format for the returned date. Valid values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MM/DD/YY (07/01/97 or 03/27/2002) — American numeric format. The dateseparator character (/ or .) is taken from the current locale setting.</td>
</tr>
<tr>
<td>2</td>
<td>DD Mmm [YY]YY (01 Jul 97)</td>
</tr>
<tr>
<td>Value</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>3</td>
<td><strong>YYYY-MM-DD</strong> <em>(1997-07-01) — ODBC format.</em> By default this format is independent of your current locale settings, thus specifying dates and times in an ODBC standard interchange format. <em>(The ODBC time format default is described in the <em>format</em> section, below.</em>) To use your current date and time locale settings with this format, set <em>localeopt</em> to 0.</td>
</tr>
<tr>
<td>4</td>
<td><strong>DD/MM/YYYY</strong> <em>(01/07/97 or 27/03/2002) — European numeric format.</em> The date separator character (/ or .) is taken from the current locale setting.</td>
</tr>
<tr>
<td>5</td>
<td><strong>Mmm [D]D, YYYY</strong> <em>(Jul 1, 1997)</em></td>
</tr>
<tr>
<td>6</td>
<td><strong>Mmm [D]D YYYY</strong> <em>(Jul 1 1997)</em></td>
</tr>
<tr>
<td>7</td>
<td><strong>Mmm DD [YY]YY</strong> <em>(Jul 01 1997)</em></td>
</tr>
<tr>
<td>8</td>
<td><strong>YYYYMMDD</strong> <em>(19970701) — Numeric format</em></td>
</tr>
<tr>
<td>9</td>
<td><strong>Mmmmm [D]D, YYYY</strong> <em>(July 1, 1997)</em></td>
</tr>
<tr>
<td>10</td>
<td><strong>W</strong> <em>(2) — Day number for the week, numbered from 0 (Sunday) through 6 (Saturday). Compare with the $SYSTEM.SQL.DAYOFWEEK() method.</em></td>
</tr>
<tr>
<td>11</td>
<td><strong>Www</strong> <em>(Tue) — Abbreviated day name</em></td>
</tr>
<tr>
<td>12</td>
<td><strong>Wwwwwww</strong> <em>(Tuesday) — Full day name</em></td>
</tr>
<tr>
<td>13</td>
<td><strong>[D][M]M/YYYY</strong> <em>(1/7/2549 or 27/11/2549) — Thai date format. Day and month are identical to European usage, except no leading zeros. The year is the Buddhist Era (BE) year, calculated by adding 543 years to the Gregorian year.</em></td>
</tr>
<tr>
<td>14</td>
<td><strong>nnn</strong> <em>(354) — Day number for the year</em></td>
</tr>
<tr>
<td>15</td>
<td><strong>DD/MM/YYYY</strong> <em>(01/07/97 or 27/03/2002) — European format (same as <em>dformat=4</em>). The date separator character (/ or .) is taken from the current locale setting.</em></td>
</tr>
<tr>
<td>16</td>
<td><strong>YYYYc[MMc[D]Dc</strong> — Japanese date format. Year, month, and day numbers are the same as other date formats; leading zeros are omitted. The Japanese characters for “year”, “month”, and “day” (shown here as c) are inserted after the year, month, and day numbers. These characters are Year=$CHAR(24180), Month=$CHAR(26376), and Day=$CHAR(26085).</td>
</tr>
<tr>
<td>17</td>
<td><strong>YYYYc [M]Mc [D]Dc</strong> — Japanese date format. Same as <em>dformat</em> 16, except that a blank space is inserted after the “year” and “month” Japanese characters.</td>
</tr>
<tr>
<td>18</td>
<td><strong>[D]D Mmmmm YYYY</strong> — Tabular Hijri (Islamic) date format with full month name. Day leading zeros are omitted; year leading zeros are included. Caché date -445031 (07/19/0622 C.E.) = 1 Muharram 0001.</td>
</tr>
<tr>
<td>19</td>
<td><strong>[D]D [M]M YYYY</strong> — Tabular Hijri (Islamic) date format with month number. Day and month leading zeros are omitted; year leading zeros are included. Caché date -445031 (07/19/0622 C.E.) = 1 1 0001.</td>
</tr>
<tr>
<td>20</td>
<td><strong>[D]D Mmmmm YYYY</strong> — Observed Hijri (Islamic) date format with full month name. Defaults to Tabular Hijri (<em>dformat</em> 18). To override tabular calculation, use the class %Calendar.Hijri to add observations of new moon crescents.</td>
</tr>
<tr>
<td>21</td>
<td><strong>[D]D [M]M YYYY</strong> — Observed Hijri (Islamic) date format with month number. Defaults to Tabular Hijri (<em>dformat</em> 19). To override tabular calculation, use the class %Calendar.Hijri to add observations of new moon crescents.</td>
</tr>
</tbody>
</table>
Meaning | Value  
---|---
Get effective *dformat* value from the user’s locale, *fmt*.DateFormat, where *fmt* is an instance of ##class(%SYS.NLS.Format) associated with the current process. This is the default behavior if you do not specify *dformat*. See “Customizable Date and Time Defaults” for further details. | -1

$ZDATETIME$ returns an integer specifying the count of seconds from a platform-specific origin date/time. This is the value returned by the time() library function, as defined in the ISO C Programming Language Standard. For example, on POSIX-compliant systems this value is the count of seconds from January 1, 1970 00:00:00 UTC. Fractional seconds in the input value are permitted, but ignored.

(Currently, this date conversion potentially has the “local time variant boundary day” time conversion anomaly described for *tformat* values 5, 6, 7, and 8.)

The following platform-specific formats are supported: 32-bit Linux: signed 32-bit integer; 64-bit Linux: signed 64-bit integer; Windows: unsigned 64-bit integer. Refer to the MultiValue Basic SYSTEM(99) function in *Caché MultiValue Basic Reference*.

The *tformat*, *precision*, *monthlist*, *yearopt*, *startwin*, and *endwin* parameters are ignored. | -2

$ZDATETIME$ takes a *datetime* value specified in $HOROLOG$ internal format, converts that value from local time to UTC Universal time, and returns the resulting value in the same internal format. The *tformat*, *monthlist*, *yearopt*, *startwin*, and *endwin* parameters are ignored. $ZDATETIMEH$ performs the inverse operation. (Currently, this date conversion has the time conversion anomalies described for *tformat* values 5, 6, 7, and 8. These potentially affect dates prior to 1970, dates after 2038, and local time variant boundary days, such as the beginning date or end date for Daylight Saving Time.) | -3

Where:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YYYY</td>
<td>YYYY is a four-digit year. [YY]YY is a two-digit year if <em>hdatetime</em> falls within the active window for two-digit years; otherwise it is a four-digit year.</td>
</tr>
<tr>
<td>MM</td>
<td>Two-digit month: 01 through 12. [M]M indicates that the leading zero is omitted for months 1 through 9.</td>
</tr>
<tr>
<td>DD</td>
<td>Two-digit day: 01 through 31. [D]D indicates that the leading zero is omitted for days 1 through 9.</td>
</tr>
<tr>
<td>Mmm</td>
<td>Month abbreviation extracted from the MonthAbbr property of the current locale. An alternate month abbreviation (or name of any length) can be extracted from an optional list specified as the <em>monthlist</em> parameter to $ZDATETIME$. The MonthAbbr default values are: “Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec”</td>
</tr>
<tr>
<td>Mmmmm</td>
<td>Full name of the month as specified by the MonthName property of the current locale. The default values are: “January February March ... November December”</td>
</tr>
<tr>
<td>W</td>
<td>Number 0-6 indicating the day of the week: Sunday=0, Monday=1, Tuesday=2, etc.</td>
</tr>
<tr>
<td>Wwww</td>
<td>Weekday name abbreviation as specified by the WeekdayAbbr property of the current locale. The default values are: “Sun Mon Tue Wed Thu Fri Sat”</td>
</tr>
<tr>
<td>Wwwwww</td>
<td>Weekday full name as specified by the WeekdayName property of the current locale. The default values are: “Sunday Monday Tuesday Wednesday Thursday Friday Saturday”</td>
</tr>
<tr>
<td>Syntax</td>
<td>Meaning</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td><code>nnn</code></td>
<td>Day number for the specified year, always three digits, with leading zeros if necessary. Values are 001 through 365 (or 366 on leap years).</td>
</tr>
</tbody>
</table>

**dformat Default**

If you omit `dformat` or set it to -1, the `dformat` default depends on the `localeopt` parameter and the NLS DateFormat property:

- If `localeopt`=1 the `dformat` default is ODBC format. The `tformat`, `monthlist`, `yearopt`, `mindate` and `maxdate` parameters are also set to ODBC format. This is the same as setting `dformat`=3.
- If `localeopt`=0 or is unspecified, the `dformat` default is taken from NLS DateFormat property. If `DateFormat`=3, the `dformat` default is ODBC format. However, `DateFormat`=3 does not affect the `tformat`, `monthlist`, `yearopt`, `mindate` and `maxdate` parameter defaults, which are as specified in the current NLS locale definition.

To determine the default date format for your locale, invoke the `GetFormatItem()` NLS class method:

```cachéobjectscript
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DateFormat")
```

European date format (`dformat`=4, DD/MM/YYYY order) is the default for many (but not all) European languages, including British English, French, German, Italian, Spanish, and Portuguese (which use a “/” DateSeparator character), as well as Czech (csyw), Russian (rusw), Slovak (skyw), Slovenian (svnw), and Ukrainian (ukrw) (which use a “.” DateSeparator character). For further details on default date formats for supported locales, refer to Dates in Using Caché ObjectScript.

**dformat Settings**

If `dformat` is 3 (ODBC date format), ODBC format defaults are also used for the `monthlist`, `yearopt`, `mindate` and `maxdate` parameter defaults. Current locale defaults are ignored.

If `dformat` is -1, 1, 4, 13, or 15 (numeric date formats), `$ZDATETIME` uses the value of the DateSeparator property of the current locale as the delimiter between months, days, and the year. When `dformat` is 3 the ODBC date separator (“.”) is used. For all other `dformat` values, a space is used as the date separator. The default value of DateSeparator in English is “/” and all documentation uses this delimiter.

If `dformat` is 11 or 12 (day names) and `localeopt`=0 or is unspecified the day name values come from the current locale properties. If `localeopt`=1, day names come from the ODBC locale. To determine the default weekday names and weekday abbreviations for your locale, invoke the following NLS class methods:

```cachéobjectscript
WRITE ##class(%SYS.NLS.Format).GetFormatItem("WeekdayName"),! WRITE ##class(%SYS.NLS.Format).GetFormatItem("WeekdayAbbr"),!
```

If `dformat` is 16 or 17 (Japanese date formats), the returned date format is independent of the locale setting. Japanese-format dates can be returned from any Unicode Caché instance. On an 8-bit Caché instance specifying `dformat` is 16 or 17 causes a `<ERWIDECHAR> error.

If `dformat` is 18, 19, 20, or 21 (Islamic date formats) and `localeopt` is unspecified, parameters default to Islamic defaults, rather than current locale defaults. The `monthlist` parameter defaults to Arabic month names transliterated with Latin characters. The `tformat`, `yearopt`, `mindate` and `maxdate` parameters default to ODBC defaults. The date separator defaults to the Islamic default (a space), not the ODBC default or the current locale DateSeparator property value. If `localeopt`=0 current locale property defaults are used for these parameters. If `localeopt`=1 ODBC defaults are used for these parameters.

**tformat**

A numeric value that specifies the format in which you want to express the time value. Supported values are:
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Get the effective <code>tformat</code> value from the <code>TimeFormat</code> property of the current locale, which defaults to a value of 1. This is the default behavior if you do not specify <code>tformat</code> for all <code>dformat</code> values except 3.</td>
</tr>
<tr>
<td>1</td>
<td>Express time in the form &quot;hh:mm:ss&quot; (24-hour clock).</td>
</tr>
<tr>
<td>2</td>
<td>Express time in the form &quot;hh:mm&quot; (24-hour clock)</td>
</tr>
<tr>
<td>3</td>
<td>Express time in the form &quot;hh:mm:ss[AM/PM]&quot; (12-hour clock)</td>
</tr>
<tr>
<td>4</td>
<td>Express time in the form &quot;hh:mm[AM/PM]&quot; (12-hour clock)</td>
</tr>
<tr>
<td>5</td>
<td>Express time in the form &quot;hh:mm:ss+-hh:mm&quot; (24-hour clock). The time is expressed as local time. The plus (+) or minus (−) suffix shows the system-defined offset of local time from Coordinated Universal Time (UTC). A minus sign (-hh:mm) indicates that the local time is earlier (westward) of the Greenwich meridian by the returned offset number of hours and minutes. A plus sign (+hh:mm) indicates that the local time is later (eastward) of the Greenwich meridian by the returned offset number of hours and minutes. Further details are described below.</td>
</tr>
<tr>
<td>6</td>
<td>Express time in the form &quot;hh:mm+-hh:mm&quot; (24-hour clock). The time is expressed as local time. The plus (+) or minus (−) suffix shows the system-defined offset of local time from Coordinated Universal Time (UTC). A minus sign (-hh:mm) indicates that the local time is earlier (westward) of the Greenwich meridian by the returned offset number of hours and minutes. A plus sign (+hh:mm) indicates that the local time is later (eastward) of the Greenwich meridian by the returned offset number of hours and minutes. Further details are described below.</td>
</tr>
<tr>
<td>7</td>
<td>Express time in the form &quot;hh:mm:ssZ&quot; (24-hour clock). The “Z” suffix indicates that the time is expressed as Coordinated Universal Time (UTC), rather than as local time.</td>
</tr>
<tr>
<td>8</td>
<td>Express time in the form &quot;hh:mmZ&quot; (24-hour clock). The “Z” suffix indicates that the time is expressed as Coordinated Universal Time (UTC), rather than as local time.</td>
</tr>
<tr>
<td>9</td>
<td>For MultiValue support. Same as <code>tformat</code> 1 (&quot;hh:mm:ss&quot; 24-hour clock) for numeric values in the range 0 through 86399. Also accepts negative time values and time values greater than 86399, as described below.</td>
</tr>
<tr>
<td>10</td>
<td>For MultiValue support. Same as <code>tformat</code> 2 (&quot;hh:mm&quot; 24-hour clock) for numeric values in the range 0 through 86399. Also accepts negative time values and time values greater than 86399, as described below.</td>
</tr>
</tbody>
</table>

If you omit `tformat` or set it to -1, the `tformat` default depends on the `localeopt` parameter and the NLS `TimeFormat` property:

For all `dformat` values except 3, 18, 19, 20, and 21 all time formats default to the current locale definition `TimeSeparator` and `DecimalSeparator` property values. For `dformat`=3 (ODBC date format) and `dformat`=18, 19, 20, or 21 (Islamic date formats) the time separator is a colon (:) and the `DecimalSeparator` is a period (.) regardless of the current locale property values. These defaults can be overridden by setting `localeopt`.  

To determine the default time properties for your locale, invoke the `GetFormatItem()` NLS class method:

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("TimeFormat"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("TimeSeparator"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DecimalSeparator")
```

For `tformat` values 1 through 4 (which return local time), the date is separated from the time by a space. For `tformat` values 5 through 8 the date is separated from the time by the letter “T”.
12-hour Clock (tformat 3 and 4)

In 12-hour clock formats, morning and evening are represented by time suffixes, here shown as AM and PM. To determine the default time suffixes for your locale, invoke the GetFormatItem() NLS class method, as follows:

```objectscript
WRITE ##class(%SYS.NLS.Format).GetFormatItem("AM"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("PM"),!
```

For all dformat values except 3, 18, 19, 20, and 21 the AM and PM properties default to the current locale definition. For dformat=3 (ODBC date format) and dformat=18, 19, 20, or 21 (Islamic date formats) the time suffixes are always “AM” and “PM”, regardless of the current locale property values. The AM and PM property defaults are “AM” and “PM” for all locales except the Japanese locale jpww.

By default, Midnight and Noon are represented as “MIDNIGHT” and “NOON” for all locales except Japanese locales (jpww, jpuw, jpww, zdsw, zdtw, zduw), Portuguese (ptbw), Russian (rusw), and Ukrainian (ukrw).

However, when dformat=3, $ZDATETIME always uses the ODBC standard values, regardless of the default settings for your locale.

Local Time (tformat 5, 6, 7, and 8)

When tformat is set to 5 or 6, the hdatetime input value is assumed to be local date and time, and is displayed as local date and time. If hdatetime is the current local date and time ($HOROLOG), changing $ZTIMEZONE will change this current date and time for the current process.

The offset suffix specifies the local time variant setting as a positive or negative offset in hours and minutes from the Greenwich meridian. Note that this local time variant is not necessarily the time zone offset. For example, the Eastern United States time zone is 5 hours west of Greenwich (-5:00), but the local time variant (Daylight Saving Time) offsets the time zone time by one hour to -04:00. Setting $ZTIMEZONE changes the current process date and time returned by $ZDATETIME($HOROLOG,1,5), but does not change the system local time variation setting.

**Note:** tformat 5 or 6 return the local time variation offset from UTC time. This is neither the local time zone offset, nor is it a comparison of your local time with local time at Greenwich England. The term Greenwich Mean Time (GMT) may be confusing; local time at Greenwich England is the same as UTC during the winter; during the summer it differs from UTC by one hour. This is because a local time variant, known as British Summer Time, is applied.

The following example shows how the tformat 5 value is affected by the operating system’s local time variation setting and by changing the time zone for the current process. (Note that this example checks whether a local time variation boundary occurs during program execution):

```objectscript
LocalDatetimeOffset
SET dst=$SYSTEM.Util.IsDST()
SET local=$ZDATETIME($HOROLOG,1,5)
WRITE local," is the local date/time and offset",!!
SET off=$PIECE(local,"+",2)
IF off="" {SET off=$PIECE(local,"-",2)
    WRITE ",,",off," is local offset",!}
ELSE {WRITE "",off," is local offset",!}
SET tz=$ZTIMEZONE
WRITE tz/60," is the local timezone offset, in hours",!!
IF dst=1 {WRITE " DST in effect, ",off," offset is not ",tz/60," time zone offset"!,} ELSEIF dst=0 {WRITE " DST not in effect, ",off,"=timezone ",tz/60,!,}
ELSE {WRITE " DST setting cannot be determined",!}
ChangeTimezoneForCurrentProcess
SET $ZTIMEZONE=tz+180
WRITE ",,changed the process time zone westward 3 hours",!
WRITE $ZDATETIME($HOROLOG,1,5)," is new local date/time and offset",!
WRITE "note that time has changed, but offset has not changed"
SET $ZTIMEZONE=tz
ConfirmNoDSTBoundary
SET dst2=$SYSTEM.Util.IsDST()
GOTO:dst'=dst2 LocalDatetimeOffset
```
When $tformat$ is set to 7 or 8, the $hdatetime$ input value is assumed to be local date and time. The time is changed to correspond to UTC time (calculated using the local timezone setting). The date returned may also be changed (if necessary) to correspond to this UTC time value. Thus the returned date may differ from the local date.

**Note:** Conversions involving $dformat$ values -2 and -3 and $tformat$ values 7 and 8 and the UTC offsets generated by $tformat$ values 5 and 6 have the following platform-dependent anomalies:

- Conversions from local time to UTC time depend on local time variant boundary behavior, which may differ on different operating system platforms:
  
  When the local clock shifts forwards (“Spring ahead” at the start of Daylight Saving Time) the local time loses an hour. This “lost” hour is an illegal local time value. Caché $SHOROLOG$ should never return an illegal local time value. However, if a user manually enters this illegal local time value (for example, by setting $SHOROLOG$), $SZDATETIME$ conversion results are undefined and highly platform-dependent.

  When the local clock shifts backwards (“Fall back” at the end of Daylight Saving Time) the local time hour is repeated. Within this two-hour period, a Caché time conversion operation cannot determine whether it is being applied to the first occurrence of that local time hour, or the second occurrence of the same hour. $SZDATETIME$ uses whichever assumption is used by the platform-specific runtime library. Therefore, within this temporal window, different operating system platforms may give different time conversion results.

- Conversions between local time and UTC time must use the local time variant rules in force for the specified year and location. Because these rules are established by local laws, may have changed in the past, and are subject to change in the future, $SZDATETIME$ conversions depend upon the completeness and accuracy of the rules as encoded by the operating system platform. Predictions for future years must necessarily use the current rules, and these rules may change.

- Conversions between local time and UTC time depend on the date range supported by the operating system platform:

  If a date is earlier than the earliest date supported by the platform, Caché uses the standard time offset for 1902–01–01 (if this date is supported by the platform). If the date 1902–01–01 is not supported by the platform, Caché uses the standard time offset for 1970–01–01. Any local time variant offset (such as Daylight Saving Time) is ignored.

  If a date is later than the latest date supported by the platform, Caché calculates a corresponding date within the range 2010–01–01 to 2037–12–31 and uses the standard time offset for that corresponding date. This algorithm should provide accurate time offsets for dates up to 2100–02–28, provided there are no future changes to the laws governing date/time observances.

**Note:** $SZDATETIME$ has no way to determine if an $hdatetime$ input value is in UTC time or local time. Therefore, do not use $tformat$ values 5, 6, 7, or 8 with an $hdatetime$ that is already in UTC, such as a $SZTIMESTAMP$ value. If you use the output from a $tformat$ 7 or 8 conversion in an operation that converts the time back to local time be aware that the date may have been changed in the local-to-UTC conversion.

### MultiValue Time ($tformat$ 9 and 10)

Time formats 9 and 10 are provided for MultiValue support. They are identical to $tformats$ 1 and 2 for all time values within the allowed range for ObjectScript: 0 through 86399. Within this range they handle fractional seconds in the same manner as other time formats. Like other time formats, -0 is treated as 0.

Time formats 9 and 10 accept negative numeric values and returns a corresponding negative time value. $tformat$ 9 returns a negative time value as “-hh:mm:ss” (24-hour clock). $tformat$ 10 returns a negative time value as “-hh:mm” (24-hour clock). For time format 10, “--59.9” returns “--00:00” and “--60” returns “--00:01”. Negative fractional seconds are always truncated. Therefore, -4.9 returns “--00:00:04”, “--0” and “--0.9” both return the positive value “00:00:00”.
Time formats 9 and 10 accept numeric values greater than 86399, and return the corresponding time. Thus 86400 returns "24:00:00", and 400000 returns "111:06:40". Fractional seconds are returned for values in the range 0 through 86399. Fractional seconds are truncated for values of 86400 and greater.

If precision is specified, and a time value outside the Caché range is specified, any specified fractional seconds are truncated and the fractional precision portion is returned as zeros.

**precision**

An integer value that specifies the number of decimal places of fractional seconds precision used to express the time. That is, if you enter a value of 3 as precision, **$ZDATETIME** displays the seconds carried out to three decimal places. If you enter a value of 9 as precision, **$ZDATETIME** displays the seconds carried out to nine decimal places. This argument specifies the number of fractional digits to return; the actual number of meaningful digits of precision is determined by the **hdatetime** source. For example, **$HOROLOG** does not return fractional seconds; **$ZTIMESTAMP** and **$NOW()** return fractional seconds.

Supported values are as follows:

-1: Get the precision value from the TimePrecision property of the current locale, which defaults to a value of 0. This is the default behavior if you do not specify precision.

A value of n that is greater than or equal to zero (0) results in the expression of time to n decimal places.

Precision is only applicable if the **hdatetime** format can include a fractional time value (**$ZTIMESTAMP** format), and if the dformat option selected includes seconds. Trailing zeros are not suppressed. If the precision specified exceeds the precision available on your system, the excess digits of precision are returned as trailing zeros. You can use the Normalize() method to suppress excess trailing zeros, as shown in the following example, which specifies a precision of 9:

```objectscript
WRITE $ZDATETIME($ZTIMESTAMP,3,,9),!
WRITE ##class(%TimeStamp).Normalize($ZDATETIME($ZTIMESTAMP,3,,9))
```

To determine the default time precision for your locale, invoke the GetFormatItem() NLS class method:

```objectscript
WRITE ##class(%SYS.NLS.Format).GetFormatItem("TimePrecision")
```

You can set the locale’s TimePrecision to the desired number of digits, to a maximum of 15.

**monthlist**

An expression that resolves to a string of month names or month name abbreviations, separated by a delimiter character. The names in **monthlist** replace the default month abbreviation values from the MonthAbbr property or the month name values from the MonthName property of the current locale.

**monthlist** is valid only if **dformat** is 2, 5, 6, 7, 9, 18, or 20. If **dformat** is any other value **$ZDATETIME** ignores **monthlist**.

The **monthlist** string has the following format:

- The first character of the string is a delimiter character (usually a space). The same delimiter must appear before the first month name and between each month name in **monthlist**. You can specify any single-character delimiter; this delimiter appears between the month, day, and year portions of the returned date value, which is why a space is usually the preferred character.

- The month names string should contain twelve delimited values, corresponding to January through December. It is possible to specify more or less than twelve month names, but if there is no month name corresponding to the month in **hdatetime** an <ILLEGAL VALUE> error is generated.

If you omit **monthlist** or specify a **monthlist** value of -1, **$ZDATETIME** uses the list of month names defined in the MonthAbbr or MonthName property of the current locale, unless one of the following is true: If localeopt=1, the **monthlist** default is the ODBC month list (in English). If localeopt is unspecified and **dformat** is 18 or 20 (Islamic date formats) the
monthlist default is the Islamic month list (Arabic names expressed using Latin letters), ignoring the MonthAbbr or MonthName property value.

To determine the default month names and month abbreviations for your locale, invoke the GetFormatItem() NLS class method:

```ObjectScript
WRITE ##class(%SYS.NLS.Format).GetFormatItem("MonthName"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("MonthAbbr"),!
```

**yearopt**

With dfomat values 0, 1, 2, 4, 7, or 15, a numeric code that specifies the temporal window in which to display the year as a two-digit value. yearopt can be:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Get effective yearopt value from YearOption property of current locale which defaults to a value of 0. This is the default behavior if you do not specify yearopt.</td>
</tr>
<tr>
<td>0</td>
<td>Represent 20th century dates (1900 through 1999) with two-digit years, unless a process-specific sliding window (established via the %DATE utility) is in effect. If such a window is in effect, represent only those dates falling within the sliding window by two-digit years. Represent all dates falling outside the 20th century or outside the process-specific sliding window by four-digit years.</td>
</tr>
<tr>
<td>1</td>
<td>Represent 20th century dates with two-digit years and all other dates with four-digit years.</td>
</tr>
<tr>
<td>2</td>
<td>Represent all dates with two-digit years.</td>
</tr>
<tr>
<td>3</td>
<td>Represent with two-digit years those dates falling within the sliding temporal window defined by startwin and (optionally) endwin. Represent all other dates with four-digit years. When yearopt=3, startwin and endwin are absolute dates in $HOROLOG format.</td>
</tr>
<tr>
<td>4</td>
<td>Represent all dates with four-digit years.</td>
</tr>
<tr>
<td>5</td>
<td>Represent with two-digit years all dates falling within the sliding window defined by startwin and (optionally) endwin. Represent all other dates with four-digit years. When yearopt=5, startwin and endwin are relative years.</td>
</tr>
<tr>
<td>6</td>
<td>Represent all dates in the current century with two-digit years and all other dates with four-digit years.</td>
</tr>
</tbody>
</table>

To determine the default year option for your locale, invoke the GetFormatItem() NLS class method:

```ObjectScript
WRITE ##class(%SYS.NLS.Format).GetFormatItem("YearOption")
```

If you omit yearopt or specify a yearopt value of -1, $ZDATETIME uses the YearOption property of the current locale, unless one of the following is true: If localeopt=1, the yearopt default is the ODBC year option. If localeopt=0 or is unspecified and dfomat is 18, 19, 20, or 21 (Islamic date formats) the yearopt default is the ODBC year option (4-digit years); the YearOption property value is ignored for Islamic dates.

**startwin**

A numeric value that specifies the start of the sliding window during which dates must be represented with two-digit years. You must supply startwin when you use a yearopt of 3 or 5. startwin is not valid with any other yearopt values.

When yearopt=3, startwin is an absolute date in $HOROLOG date format that indicates the start date of the sliding window.

When yearopt=5, startwin is a numeric value that indicates the start year of the sliding window expressed in the number of years before the current year.
**endwin**

A numeric value that specifies the end of the sliding window during which dates are represented with two-digit years. You may optionally supply `endwin` when `yearopt` is 3 or 5. `endwin` is not valid with any other `yearopt` values.

When `yearopt`=3, `endwin` is an absolute date in **$HOROLOG** date format that indicates the end date of the sliding window.

When `yearopt`=5, `endwin` is a numeric value that indicates the end year of the sliding window expressed as the number of years past the current year.

When `yearopt`=5, the sliding window always begins on January 1st of the year specified in `startwin` and ends on December 31st of the year specified in `endwin`, or of the implied end year (if you omit `endwin`).

If `endwin` is omitted (or specified as -1) the effective sliding window will be 100 years long. The `endwin` value of -1 is a special case that always returns a date value, even when higher and lower `endwin` values return `erropt`. For this reason, it is preferable to omit `endwin` when specifying a 100-year window, and to avoid the use of negative `endwin` values.

If you supply both `startwin` and `endwin`, the sliding window they specify must not have a duration of more than 100 years.

**mindate**

An expression that specifies the lower limit of the range of valid dates (inclusive). Can be specified as a **$HOROLOG** integer date count (for example, 1/1/2013 is represented as 62823) or a **$HOROLOG** string value. You can include or omit the time portion of the **$HOROLOG** date (for example “62823,43200”), but only the date portion of `mindate` is parsed. Specifying an `hdatetime` value earlier than `mindate` generates a `<VALUE OUT OF RANGE>` error.

The following are supported `mindate` values:

- **Positive integer**: Most commonly `mindate` is specified as a positive integer to establish the earliest allowed date as some date after December 31, 1840. For example, a `mindate` of 21550 would establish the earliest allowed date as January 1, 1900. The highest valid value is 2980013 (December 31, 9999).

- **0**: specifies the minimum date as December 31, 1840. This is the `DateMinimum` property default.

- **Negative integer -2 or larger**: specifies a minimum date counting backwards from December 31, 1840. For example, a `mindate` of -14974 would establish the earliest allowed date as January 1, 1800. Negative `mindate` values are only meaningful if the `DateMinimum` property of the current locale has been set to an equal or greater negative number. The lowest valid value is -672045.

- If omitted (or specified as -1), `mindate` defaults to the `DateMinimum` property value for the current locale, unless one of the following is true: If `localeopt`=1, the `mindate` default is 0. If `localeopt` is unspecified and `dformat`=3, the `mindate` default is 0. If `localeopt` is unspecified and `dformat` is 18, 19, 20, or 21 (Islamic date formats) the `mindate` default is 0.

You can get and set the `DateMinimum` property as follows:

```ObjectScript
SET min=#%SYS.NLS.Format.GetFormatItem("DateMinimum")
WRITE "initial DateMinimum value is ",min,!
Permit18thCenturyDates
SET x=#%SYS.NLS.Format.SetFormatItem("DateMinimum",-51498)
SET newmin=#%SYS.NLS.Format.GetFormatItem("DateMinimum")
WRITE "set DateMinimum value is ",newmin,!!
RestrictTo19thCenturyDates
WRITE $ZDATETIME(-13000,1,,,,,,,,,-14974),!!
ResetDateMinimumToDefault
SET oldmin=#%SYS.NLS.Format.SetFormatItem("DateMinimum",min)
WRITE "reset DateMinimum value from ",oldmin," to ",min
```

You may specify `mindate` with or without `maxdate`. Specifying a `mindate` larger than `maxdate` generates an `<ILLEGAL VALUE>` error.
ODBC Date Format (dformat 3)
The application of the DateMinimum property is governed by the localeopt setting. When localeopt=1 (which is the default for dformat=3) the date minimum is 0, regardless of the current locale setting. Therefore, in ODBC format (dformat=3) the following can be used to specify a date prior to December 31, 1840:

- Specify a mindate earlier than the specified date:
  WRITE $ZDATETIME(-30,3,,,,,,,,,-365)

- Specify a DateMinimum property value earlier than the specified date and set localeopt=0:
  DO ##class(%SYS.NLS.Format).SetFormatItem("DateMinimum",-365)
  WRITE $ZDATETIME(-30,3,,,,,,,,,0)

maxdate
An expression that specifies the upper limit of the range of valid dates (inclusive). Can be specified as a $HOROLOG integer date count (for example, 1/1/2100 is represented as 94599) or a $HOROLOG string value. You can include or omit the time portion of the $HOROLOG date (for example “94599,43200”), but only the date portion of maxdate is parsed.

If maxdate is omitted or if specified as -1, the maximum date limit is obtained from the DateMaximum property of the current locale, which defaults to the maximum permissible value for the date portion of $HOROLOG: 2980013 (corresponding to December 31, 9999 CE). However, the application of the DateMaximum property is governed by the localeopt setting. When localeopt=1 (which is the default for dformat=3) the date maximum default is the ODBC value (2980013), regardless of the current locale setting. Islamic date formats also take the ODBC default. The maximum date for Thai date format (dformat=13) is $HOROLOG 2781687 which corresponds to 31/12/9999 BE.

Specifying a hdatetime date larger than maxdate generates a <VALUE OUT OF RANGE> error.

Specifying a maxdate larger than 2980013 generates an <ILLEGAL VALUE> error.

You may specify maxdate with or without mindate. Specifying a maxdate smaller than mindate generates an <ILLEGAL VALUE> error.

erropt
Specifying a value for this parameter suppresses errors associated with invalid or out of range hdatetime values. Instead of generating <ILLEGAL VALUE> or <VALUE OUT OF RANGE> errors, the $ZDATETIME function returns the erropt value.

- Validation: Caché performs canonical numeric conversion on hdatetime. The date and time portions of hdatetime are parsed separately. It parses the first comma encountered as the date/time separator. Additional commas are treated as non-numeric characters.

  Parsing of each portion of an hdatetime string halts at the first non-numeric character. Therefore, an hdatetime such as 64687AD,1234SECS is the same as 64687, 1234. A non-numeric date or time portion (including the null string) evaluates to 0. Thus an empty string hdatetime returns the $HOROLOG initial date: December 31, 1840.

  However, if the date portion value does not evaluate to an integer (contains a non-zero fractional number) it generates an <ILLEGAL VALUE> error.

- Range: The date portion of hdatetime must evaluate to an integer within the mindate/maxdate range. By default, date values greater than 2980013 or less than 0 generate a <VALUE OUT OF RANGE> error. By setting mindate to a negative number, you can extend the range of valid dates before December 31, 1840. However, for dformat 18, 19, 20, or 21 (Hijri Islamic calendar) dates, any date prior to -445031 generates an <ILLEGAL VALUE> error, even if mindate is set to an earlier date.
A time portion of `hdatetime` with a value greater than 86399 generates an `<ILLEGAL VALUE>` error. A negative `hdatetime` time value generates an `<ILLEGAL VALUE>` error.

The `erropt` parameter only suppresses errors generated due to invalid or out of range values of `hdatetime`. Errors generated due to invalid or out of range values of other parameters will always generate errors whether or not `erropt` has been supplied. For example, an `<ILLEGAL VALUE>` error is always generated when `$ZDATETIME` specifies a sliding window where `endwin` is earlier than `startwin`. Similarly, an `<ILLEGAL VALUE>` error is generated when `maxdate` is less than `mindate`.

### localeopt

This Boolean parameter specifies either the user’s current locale definition or the ODBC locale definition as the source for defaults for the locale-specified parameters `dformat`, `tformat`, `monthlist`, `yearopt`, `mindate` and `maxdate`:

- If `localeopt`=0, all of these parameters take the current locale definition defaults.
- If `localeopt`=1, all of these parameters take the ODBC defaults.
- If `localeopt` is not specified, the `dformat` parameter determine the default for these parameters. If `dformat`=3, the ODBC defaults are used. If `dformat` is 18, 19, 20, or 21 the Islamic date and time format defaults are used, regardless of the current locale definition. For all other `dformat` values, the current locale definition defaults are used. Refer to the `dformat` description for further details.

The ODBC standard locale cannot be changed; it is used to format date and time strings that are portable between Caché processes that have made different National Language Support (NLS) choices. The ODBC locale date and time definitions are as follows:

- Date format defaults to 3. Therefore, if `dformat` is undefined or -1, date format 3 is used.
- Date separator defaults to "/". However, date format defaults to 3, which always uses "-" as the date separator.
- Year option defaults to 4 digits.
- Date minimum and maximum are 0 and 2980013 (`$HOROLOG` date count).
- English month names, month abbreviations, weekday names, weekday abbreviations, and the words "Noon" and "Midnight" are used.
- Time format defaults to 1. Time separator is ":". Time precision is 0 (no fractional seconds). AM and PM indicators are "AM" and "PM".

### Examples

The following example displays the current local date and time. It takes the default date and time format for the locale:

```ObjectScript
WRITE $ZDATETIME($HOROLOG)
```

The following example displays the current date and time. `$ZTIMESTAMP` contains the current date and time value as Coordinated Universal Time (UTC) date and time. The `dformat` parameter specifies ODBC date format, the `tformat` parameter specifies a 24-hour clock, and the `precision` parameter specifies 6 digits of fractional second precision:

```ObjectScript
WRITE $ZDATETIME($ZTIMESTAMP,3,1,6)
```

This returns the current time stamp date and time, formatted like: `2005-11-25 18:45:16.960000`.

The following example shows how a local time can be converted to UTC time, and how the date may also change as a result of this conversion. In most time zones, the time conversion in one of the following `$ZDATETIME` operations also changes the date:

```ObjectScript
SET local = $ZDATETIME("60219,82824",3,1)
SET utcwest = $ZDATETIME("60219,82824",3,7)
SET utceast = $ZDATETIME("60219,00024",3,7)
WRITE !,local,!,utcwest,!,utceast
```
Notes

Invalid Values with $ZDATETIME

You receive a <FUNCTION> error in the following conditions:

- If you specify an invalid dformat code (an integer value less than -3 or greater than 17, a zero, or a noninteger value)
- If you specify a invalid value for tformat (an integer value less than -1 or greater than 10, a zero, or a noninteger value)
- If you do not specify a startwin value when yearopt is 3 or 5

You receive a <ILLEGAL VALUE> error under the following conditions:

- If you specify an invalid value for a date or time and do not supply an erropt value
- If the given month number is greater than the number of month values in monthlist
- If maxdate is less than mindate
- If endwin is less than startwin
- If startwin and endwin specify a sliding temporal window whose duration is greater than 100 years

You receive a <VALUE OUT OF RANGE> error under the following conditions:

- If you specify an otherwise valid date which is outside the range defined by the values assumed for maxdate and mindate and do not supply an erropt value

Customizable Date and Time Defaults

Upon Caché startup, the default date and time formats are initialized to the American date and time formats (for example, MM/DD/[YY]YY). To set this and other default formats to the values for your current locale, set the following global variable:

```ObjectScript
SET ^SYS("NLS","Config","LocaleFormat")=1
```

This sets all format defaults for all processes to your current locale values. These defaults persist until this global is changed.

Note: This section describes the user locale definitions applied when localeopt is undefined or set to 0. When localeopt=1, $ZDATETIME uses a predefined ODBC locale.

In the following example, the first $ZDATETIME returns a date and time in the default format for the locale. The input parameters are the $ZTIMESTAMP special variable, with the dformat and tformat taking defaults, and precision set to 2 decimal digits. In most locales, the first $ZDATETIME will return dformat=1 or the American date and time format with a slash date separator and a dot decimal separator for fractional seconds.

In the ChangeVals section, the first SetFormatItem() method changes the locale date format default to dformat=4, or the European date format (DD/MM/[YY]YY), as shown by the second $ZDATETIME. The second SetFormatItem() method changes the locale default for the date separator character (which affects the dformat –1, 1, 4, and 15). In this example, the date separator character is set to a dot (".") as shown by the third $ZDATETIME. The third SetFormatItem() method changes the decimal separator character for this locale to the European standard (",") as shown by the final $ZDATETIME. This program then restores the initial date format values:
$ZDATETIME

$ZDATETIME is similar to $ZDATE except it converts a combined date and time value. $ZDATE only converts a date value. For example:

WRITE $ZDATE($HOROLOG)

returns the current date, formatted like: 03/25/2011.

WRITE $ZDATETIME($HOROLOG)

returns the current date and time, formatted like: 03/25/2011 13:53:57.

$ZDATE does not support $format values 5 through 8.

See Also

- JOB command
- $ZDATE function
- $ZDATEH function
- $ZDATETIMEH function
- $ZTIME function
- $HOROLOG special variable
- $ZTIMESTAMP special variable
- $DATE utility, which is documented in the “Legacy Documentation” chapter in Using InterSystems Documentation
$ZDATETIMEH

Validates a date and time and converts from display format to Caché internal format.

$ZDATETIMEH(datetime,dformat,tformat,monthlist,yearopt,startwin,endwin,mindate,maxdate,erropt,localeopt)
$ZDTH(datetime,dformat,tformat,monthlist,yearopt,startwin,endwin,mindate,maxdate,erropt,localeopt)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>datetime</strong></td>
<td>The date and time input value. A date/time string specified in display format. $ZDATETIMEH converts this date/time string to $HOROLOG format. The <strong>datetime</strong> value can be either an explicit date and time (specified in various formats), an explicit date (specified in various formats) with the time value defaulting to 0, or the string “T” or “t”, representing the current date, with the time value either specified or defaulting to 0. The “T” or “t” string can optionally include a signed integer offset. See <strong>datetime</strong> below.</td>
<td></td>
</tr>
<tr>
<td><strong>dformat</strong></td>
<td>Optional — An integer code specifying the date format for the date portion of <strong>datetime</strong>. If <strong>datetime</strong> is “T”, <strong>dformat</strong> must be 5, 6, 7, 8, 9, or 15. See <strong>dformat</strong> below.</td>
<td></td>
</tr>
<tr>
<td><strong>tformat</strong></td>
<td>Optional — An integer code specifying the time format for the time portion of <strong>datetime</strong>. See <strong>tformat</strong> below.</td>
<td></td>
</tr>
<tr>
<td><strong>monthlist</strong></td>
<td>Optional — A string or the name of a variable that specifies a set of month names. This string must begin with a delimiter character, and its 12 entries must be separated by this delimiter character. See <strong>monthlist</strong> below.</td>
<td></td>
</tr>
<tr>
<td><strong>yearopt</strong></td>
<td>Optional — An integer code that specifies whether to represent years as two- or four-digit values. See <strong>yearopt</strong> below.</td>
<td></td>
</tr>
<tr>
<td><strong>startwin</strong></td>
<td>Optional — The start of the sliding window during which dates must be represented with two-digit years. See <strong>startwin</strong> below.</td>
<td></td>
</tr>
<tr>
<td><strong>endwin</strong></td>
<td>Optional — The end of the sliding window during which dates are represented with two-digit years. See <strong>endwin</strong> below.</td>
<td></td>
</tr>
<tr>
<td><strong>mindate</strong></td>
<td>Optional — The lower limit of the range of valid dates. Specified as a $HOROLOG integer date count, with 0 representing December 31, 1840. Can be specified as a positive or negative integer. See <strong>mindate</strong> below.</td>
<td></td>
</tr>
<tr>
<td><strong>maxdate</strong></td>
<td>Optional — The upper limit of the range of valid dates. Specified as a $HOROLOG integer date count. See <strong>maxdate</strong> below.</td>
<td></td>
</tr>
<tr>
<td><strong>erropt</strong></td>
<td>Optional — An expression to return when <strong>datetime</strong> is invalid. Specifying a value for this parameter suppresses error codes associated with invalid or out of range <strong>datetime</strong> values. Instead of issuing an error message, $ZDATETIMEH returns <strong>erropt</strong>. See <strong>erropt</strong> below.</td>
<td></td>
</tr>
</tbody>
</table>

---

ObjectScript Functions

Caché ObjectScript Reference
Optional — A boolean flag that specifies which locale to use for the \textit{dformat}, \textit{tformat}, \textit{monthlist}, \textit{yearopt}, \textit{mindate} and \textit{maxdate} default values, and other date and time characteristics, such as the DateSeparator character:

\begin{itemize}
  \item \textit{localeopt}=0: the current locale property settings determine these parameter defaults.
  \item \textit{localeopt}=1: the ODBC standard locale determines these parameter defaults.
  \item \textit{localeopt} not specified: the \textit{dformat} value determines these parameter defaults. If \textit{dformat}=3, ODBC defaults are used. Japanese and Islamic date \textit{dformat} values use their own defaults. For all other \textit{dformat} values, current locale property settings are used as defaults. See \textit{localeopt} below.
\end{itemize}

\textit{localeopt} not specified determines the date separator, time separator, and the other characters, strings, and options used to format dates and times. When 0, the current locale determines the date separator, time separator, and the other characters, strings, and options used to format dates and times. When 1, the ODBC locale determines these characters, strings, and options. The default is 0, unless \textit{dformat}=3, in which case the default is 1. See below.

Omitted parameters between specified parameter values are indicated by placeholder commas. Trailing placeholder commas are not required, but are permitted. Blank spaces are permitted between the commas that indicate an omitted parameter.

\section*{Description}
\texttt{$ZDATETIMEH$} validates a specified date and time value and converts it from display format to internal format. The corresponding \texttt{$ZDATETIME$} function converts a date and time from internal format to display format. Internal format is the format used by the \texttt{SHOROLOG} or \texttt{SZTIMESTAMP}. It represents a date and time value as a string of two numeric values separated by a comma.

The exact value returned depends on the parameters you use.

\texttt{$ZDATETIMEH$}(\textit{datetime}) converts a date and time value in the format "MM/DD/[YY]YY hh:mm:ss[.ffff]" to \texttt{SHOROLOG} format.

\begin{tabular}{|l|l|}
\hline
\textbf{Syntax} & \textbf{Meaning} \\
\hline
\textit{MM} & A two-digit month. \\
\hline
\textit{DD} & A two-digit day. \\
\hline
\textit{[YY]YY} & Two or four digits for years from 1900 to 1999. Four digits for years before 1900 or after 1999. \\
\hline
\textit{hh} & The hour in a 24-hour clock format. \\
\hline
\textit{mm} & Minutes. \\
\hline
\textit{ss} & Seconds. \\
\hline
\textit{ffff} & Fractional seconds (zero to nine digits). \\
\hline
\end{tabular}

\texttt{$ZDATETIMEH$}(\textit{datetime},\textit{dformat},\textit{tformat},\textit{monthlist},\textit{yearopt},\textit{mindate},\textit{maxdate},\textit{erropt}) converts a date and time value that was originally specified (through \texttt{$ZDATETIME$}) in date and time to \texttt{SHOROLOG} or \texttt{SZTIMESTAMP} format. The \textit{dformat}, \textit{tformat}, \textit{yearopt}, \textit{startwin} and \textit{endwin} values are identical to the values used by \texttt{$ZDATETIME$}.

When you use a \textit{dformat} of 5, 6, 7, 8, or 9 \texttt{$ZDATETIMEH$} recognizes and converts a date in any of the external American date formats corresponding to \textit{dformat} codes 1, 2, 3, 5, 6, 7, 8, 9. For a complete list of valid American date formats, refer to \texttt{SZDATEH}. When you use a \textit{dformat} of 15 \texttt{$ZDATETIMEH$} recognizes and converts a date in any unambiguous European date format. For a complete list of valid European date formats, refer to \texttt{SZDATEH}.

The \textit{dformat} values of 5, 6, 7, 8, 9, or 15 also accept the current date specified by the letter “T” or “t”, optionally followed by a plus (+) or a minus (-), and the number of days after or before the current date.
**SZDATETIMEH** recognizes and converts a time in any of eight time formats, regardless of which time format you specify in the function call. In addition, **SZDATETIMEH** recognizes the suffixes “AM, PM, NOON, and MIDNIGHT.” You can express these suffixes in uppercase, lowercase, or mixed case. You can also abbreviate these suffixes to any number of letters.

The recognized forms include:

- The default date format, **MM/DD/[YY]YY**
- The format **DDMmm[YY]YY**
- The ODBC format **[YY]YY-MM-DD**
- The format **DD/MM/[YY]YY**
- The format **Mmm D, YYYY**
- The format **Mmm D YYYY**
- The format **Mmm DD YY**
- The format **YYYYMMDD** (numeric format)

**Parameter Values**

**datetime**

The date and time string that you want to convert to **$HOROLOG** format. You can specify any of the following:

- An expression that evaluates to a single string with the date first, followed by a single blank space, followed by the time.
- An expression that evaluates to a string specifying the date only. The time value defaults to midnight (0), unless **tformat** is 7 or 8, in which case the time defaults to the local time zone offset from midnight (0).
- The letter code “T” or “t”, which specifies the current date. This letter can optionally be followed by a plus (+) or a minus (-) sign and an integer specifying an offset, in days, from the current date. You can either follow this date expression by a single blank space and a time expression, or allow the time to default to midnight (0). (If **fformat** is 7 or 8, the time defaults to the local time zone offset from midnight (0).) If you use this current date option, you must specify a **dfomat** of 5, 6, 7, 8, 9, or 15.

Valid date and time values depend on the DateFormat and TimeFormat properties of the current locale and the values specified for the **dfomat** and **tformat** parameters. For details on specifying dates, refer to **SZDATEH**.

By default, the earliest valid **datetime** date is December 31, 1840 (0 in internal **$HOROLOG** representation). Dates are limited to positive integers by default because the **DateMinimum** property defaults to 0. You can specify earlier dates as negative integers, provided the **DateMinimum** property of the current locale is set to a greater or equal negative integer. The lowest valid **DateMinimum** value is -672045, which corresponds to January 1, 0001. Caché uses the proleptic Gregorian calendar, which projects the Gregorian calendar back to “Year 1”, in conformance with the ISO 8601 standard. This is, in part, because the Gregorian calendar was adopted at different times in different countries. For example, much of continental Europe adopted it in 1582; Great Britain and the United States adopted it in 1752. Thus Caché dates prior to your local adoption of the Gregorian calendar may not correspond to historical dates that were recorded based on the local calendar then in effect. For further details on dates prior to 1840, refer to the **mindate** parameter.

**dfomat**

Format for the date. Valid values are:
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MM/DD/[YY]YY (07/01/97 or 03/27/2002) — American numeric format. You must specify the correct DateSeparator character (/ or .) for the current locale.</td>
</tr>
<tr>
<td>2</td>
<td>DD Mmm [YY]YY (01 Jul 97 or 27 Mar 2002)</td>
</tr>
<tr>
<td>3</td>
<td>[YY]YY-MM-DD (1997-07-01 or 2002-03-27) - ODBC format</td>
</tr>
<tr>
<td>4</td>
<td>DD/MM/[YY]YY (01/07/97 or 27/03/2002) — European numeric format. You must specify the correct DateSeparator character (/ or .) for the current locale.</td>
</tr>
<tr>
<td>5</td>
<td>Mmm D, YYYY (Jul 1, 1997 or Mar 27, 2002)</td>
</tr>
<tr>
<td>6</td>
<td>Mmm D YYYY (Jul 1 1997 or Mar 27 2002)</td>
</tr>
<tr>
<td>7</td>
<td>Mmm DD [YY]YY (Jul 01 1997 or Mar 27 2002)</td>
</tr>
<tr>
<td>8</td>
<td>YYYYMMDD (19930701 or 20020327) - Numeric format</td>
</tr>
<tr>
<td>9</td>
<td>Mmmmm D, YYYY (July 1, 1997 or March 27, 2002)</td>
</tr>
<tr>
<td>13</td>
<td>[D][M]/[M]YYYY (17/2549 or 27/11/2549) — Thai date format. Day and month are identical to European usage, except no leading zeros. The year is the Buddhist Era (BE) year, calculated by adding 543 years to the Gregorian year.</td>
</tr>
<tr>
<td>15</td>
<td>DD/MM/[YY]YY or YYYY-MM-DD or any unambiguous European date format with any DateSeparator character, or YYYYMMDD with no date separators. The DateSeparator character may be any non-alphanumeric character, including blank spaces, regardless of the DateSeparator character specified in the current locale. Also accepts monthlist names and “T”. For a complete list of valid European date formats, refer to $ZDATEH.</td>
</tr>
<tr>
<td>16</td>
<td>YYYYc[M]Mc[D]Dc — Japanese date format. Year, month, and day numbers are the same as other date formats; leading zeros are omitted. The Japanese characters for “year”, “month”, and “day” (shown here as c) are inserted after the year, month, and day numbers. These characters are Year=$CHAR(24180), Month=$CHAR(26376), and Day=$CHAR(26085).</td>
</tr>
<tr>
<td>17</td>
<td>YYYYc [M]Mc [D]Dc — Japanese date format. Same as dformat 16, except that a blank space is inserted after the “year” and “month” Japanese characters.</td>
</tr>
<tr>
<td>18</td>
<td>[D]D Mmmmm YYYY — Tabular Hijri (Islamic) date format with full month name. Day leading zeros are omitted; year leading zeros are included. Caché date -445031 (07/19/0622 C.E.) = 1 Muharram 0001.</td>
</tr>
<tr>
<td>19</td>
<td>[D]D [M]YYYY — Tabular Hijri (Islamic) date format with month number. Day and month leading zeros are omitted; year leading zeros are included. Caché date -445031 (07/19/0622 C.E.) = 1 1 0001.</td>
</tr>
<tr>
<td>20</td>
<td>[D]D Mmmmm YYYY — Observed Hijri (Islamic) date format with full month name. Defaults to Tabular Hijri (dformat 18). To override tabular calculation, use the class %Calendar.Hijri to add observations of new moon crescents.</td>
</tr>
<tr>
<td>21</td>
<td>[D]D [M]YYYY — Observed Hijri (Islamic) date format with month number. Defaults to Tabular Hijri (dformat 19). To override tabular calculation, use the class %Calendar.Hijri to add observations of new moon crescents.</td>
</tr>
<tr>
<td>-1</td>
<td>Get effective dformat value from the DateFormat property of the current locale. This is the default behavior if you do not specify dformat.</td>
</tr>
</tbody>
</table>
Meaning

Value | Meaning
---|---
-2 | $ZDATETIMEH takes an integer count of UTC seconds and returns the corresponding local $HOROLOG datetime value. This is the inverse of $ZDATETIME dformat –2. Refer to $ZDATETIME dformat –2 for further details.

The input datetime value is the value returned by the time() library function, as defined in the ISO C Programming Language Standard. For example, on POSIX-compliant systems this value is the count of seconds from January 1, 1970 00:00:00 UTC.

The tformat, monthlist, yearopt, startwin, and endwin parameters are ignored.

-3 | $ZDATETIMEH takes a datetime value specified in $ZTIMESTAMP internal format, converts that value from UTC Universal time to local time, and returns the resulting value in the same internal format. The tformat, monthlist, yearopt, startwin, and endwin parameters are ignored. $ZDATETIME performs the inverse operation. (Currently, this date conversion has the time conversion anomalies described for tformat values 7 and 8. These potentially affect dates prior to 1970, dates after 2038, and local time variant boundary days, such as the beginning date or end date for Daylight Saving Time.)

Where:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YYYY</td>
<td>YYYY is a four-digit year. [YY]YY is a two-digit year if datetime falls within the active window for two-digit dates; otherwise it is a four-digit number.</td>
</tr>
<tr>
<td>MM</td>
<td>Two-digit month.</td>
</tr>
<tr>
<td>D</td>
<td>One-digit day if the day number &lt;10. Otherwise, two digits.</td>
</tr>
<tr>
<td>DD</td>
<td>Two-digit day.</td>
</tr>
<tr>
<td>Mmm</td>
<td>Mmm is a month abbreviation extracted from the MonthAbbr property of the current locale. The default values are: “Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec” Or an alternate month abbreviation (or name of any length) extracted from an optional list specified as the monthlist parameter to $ZDATETIMEH.</td>
</tr>
<tr>
<td>Mmmm</td>
<td>Full name of the month as specified by the MonthName property of the current locale. The default values are: “January February March ... November December” Or an alternate month name extracted from an optional list specified as the monthlist parameter to $ZDATETIMEH.</td>
</tr>
</tbody>
</table>

**dformat Default**

If you omit dformat or set it to -1, the dformat default depends on the localeopt parameter and the NLS DateFormat property:

- If localeopt=1 the dformat default is ODBC format. The tformat, monthlist, yearopt, mindate and maxdate parameter defaults are also set to ODBC format. This is the same as setting dformat=3.

- If localeopt=0 or is unspecified, the dformat default is taken from the NLS DateFormat property. If DateFormat=3, the dformat default is ODBC format. However, DateFormat=3 does not affect the tformat, monthlist, yearopt, mindate and maxdate parameter defaults, which are as specified in the current NLS locale definition.

To determine the default date properties for your locale, invoke the GetFormatItem() NLS class method:
$ZDATETIMEH will use the value of the DateSeparator property of the current locale (either / or .) as the delimiter between months, days, and the year when $dformat=1 or 4.

European date format ($dformat=4, DD/MM/YYYY order) is the default for many (but not all) European languages, including British English, French, German, Italian, Spanish, and Portuguese (which use a “/” DateSeparator character), as well as Czech (csyw), Russian (rusw), Slovak (skyw), Slovenian (svnw), and Ukrainian (ukrw) (which use a “.” DateSeparator character). For further details on default date formats for supported locales, refer to Dates in Using Caché ObjectScript.

dformat Settings

If $dformat is 3 (ODBC format date), ODBC format defaults are also used for the $tformat, monthlist, yearopt, mindate and maxdate parameter defaults. Current locale defaults are ignored.

If $dformat is 16 or 17 (Japanese date formats), the date format is independent of the locale setting. To use Japanese-format dates you must have a Unicode Caché instance. On an 8-bit Caché instance specifying $dformat is 16 or 17 causes a <ERWIDECHAR> error.

If $dformat is 18, 19, 20, or 21 (Islamic date formats) and localeopt is unspecified, parameters default to Islamic defaults, rather than current locale defaults. The monthlist parameter defaults to Arabic month names transliterated with Latin characters. The $tformat, yearopt, mindate and maxdate parameters default to ODBC defaults. The date separator defaults to the Islamic default (a space), not the ODBC default or the current locale DateSeparator property value. If localeopt=0 current locale property defaults are used for these parameters. If localeopt=1 ODBC defaults are used for these parameters.

tformat

A numeric value that specifies the format in which the time value is input. Supported values are as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Get the effective $tformat value from the TimeFormat property of the current locale, which defaults to a value of 1. This is the default behavior if you do not specify $tformat for all $dformat values except 3.</td>
</tr>
<tr>
<td>1</td>
<td>Specify time in the form “hh:mm:ss” (24-hour clock). This is the default when $dformat=3.</td>
</tr>
<tr>
<td>2</td>
<td>Specify time in the form “hh:mm” (24-hour clock)</td>
</tr>
<tr>
<td>3</td>
<td>Specify time in the form “hh:mm:ss[AM/PM]” (12-hour clock)</td>
</tr>
<tr>
<td>4</td>
<td>Specify time in the form “hh:mm[AM/PM]” (12-hour clock)</td>
</tr>
<tr>
<td>5</td>
<td>Specify time in the form “hh:mm:ss+-hh:mm” (24-hour clock). The time is specified as local time. The following optional suffix may be supplied, but is ignored: a plus (+) or minus (–) suffix followed by the offset of local time from Coordinated Universal Time (UTC). A minus sign (-hh:mm) indicates that the local time is earlier (westward) of the Greenwich meridian by the returned offset number of hours and minutes. A plus sign (+hh:mm) indicates that the local time is later (eastward) of the Greenwich meridian by the returned offset number of hours and minutes.</td>
</tr>
<tr>
<td>6</td>
<td>Specify time in the form “hh:mm+-hh:mm” (24-hour clock). The time is specified as local time. The following optional suffix may be supplied, but is ignored: a plus (+) or minus (–) suffix followed by the offset of local time from Coordinated Universal Time (UTC). A minus sign (-hh:mm) indicates that the local time is earlier (westward) of the Greenwich meridian by the returned offset number of hours and minutes. A plus sign (+hh:mm) indicates that the local time is later (eastward) of the Greenwich meridian by the returned offset number of hours and minutes.</td>
</tr>
</tbody>
</table>
Meaning

7 Specify time in the form "hh:mm:ssZ" (24-hour clock). The time must be specified as Coordinated Universal Time (UTC). The optional "Z" suffix may be supplied or omitted, but is ignored. This suffix merely indicates that the time is assumed to be Coordinated Universal Time (UTC), rather than local time.

8 Specify time in the form "hh:mmZ" (24-hour clock). The time must be specified as Coordinated Universal Time (UTC). The optional "Z" suffix may be supplied or omitted, but is ignored. This suffix merely indicates that the time is assumed to be Coordinated Universal Time (UTC), rather than local time.

If the datetime string contains both a date part and a time part, the time part is separated from the date part by either a single space or the capital letter "T". If a time part is present:

- Time formats 1 through 6 assume datetime specifies local time using the same time zone as the result.
- Time formats 7 and 8 assume datetime specifies UTC time; these formats convert both the date and time to system local time.

For time formats 5 through 8 the datetime time value may be followed by a suffix consisting of either the capital letter "Z" or a UTC offset that starts with a "+" or "-". The presence of a suffix does not affect time zone conversion.

Note: Conversions involving dformat values -2 and -3 and tformat values 7 and 8 and the UTC offsets generated by tformat values 5 and 6 have the following platform-dependent anomalies:

- Local time variant boundary behavior may differ on different operating system platforms. When a local time variant change occurs and the local clock shifts backwards ("Fall back" at the end of Daylight Saving Time) the local time hour is repeated. Within this two-hour period, a Caché time conversion operation cannot determine whether it is being applied to the first occurrence of that local time hour, or the second occurrence of the same hour. $ZDATETIME uses whichever assumption is used by the platform-specific runtime library. Therefore, within this temporal window, different operating system platforms may give different time conversion results.
- Caché performs conversions between local time and UTC time using the standard time offset for all dates that are supported by the operating system platform.

If a specified date is earlier than the earliest date supported by the platform, Caché uses the standard time offset for 1902-01-01 (if this date is supported by the platform). If the date 1902-01-01 is not supported by the platform, Caché uses the standard time offset for 1970-01-01. Any local time variant offset (such as Daylight Saving Time) is ignored.

If a specified date is later than the latest date supported by the platform, Caché calculates a corresponding date within the range 2010-01-01 to 2037-12-31 and uses the standard time offset for that corresponding date. This algorithm should provide accurate time offsets for dates up to 2100-02-28, provided there are no future changes to the laws governing date/time observances.

To determine the default time format for your locale, invoke the GetFormatItem() NLS class method:

WRITE ##class(%SYS.NLS.Format).GetFormatItem("TimeFormat")

12-hour Clock (tformat 3 and 4)

In 12-hour clock formats, morning and evening are specified with time suffixes, here shown as AM and PM. To determine the default time suffixes for your locale, invoke the GetFormatItem() NLS class method, as follows:

WRITE ##class(%SYS.NLS.Format).GetFormatItem("AM"),! WRITE ##class(%SYS.NLS.Format).GetFormatItem("PM"),!
For all $dformat$ values except 3, 18, 19, 20, and 21 the AM and PM properties default to the current locale definition. For $dformat = 3$ (ODBC date format) and $dformat = 18, 19, 20, or 21$ (Islamic date formats) the time suffixes are always “AM” and “PM”, regardless of the current locale property values. The AM and PM property defaults are “AM” and “PM” for all locales except the Japanese locale $jpw$. By default, Midnight and Noon are represented as “MIDNIGHT” and “NOON” for all locales except Japanese locales ($jpw, jpu, jpw, zdsw, zdtw, zdwu$), Portuguese ($ptbw$), Russian ($rusw$), and Ukrainian ($ukrw$). However, when $dformat = 3$, $SZDATETIMEH$ always uses the ODBC standard values, regardless of the default settings for your locale.

$monthlist$

An expression that resolves to a string of month names or month name abbreviations, separated by a delimiter character. The names in $monthlist$ replace the default month abbreviation values from the $MonthAbbr$ property or the month name values from the $MonthName$ property of the current locale.

$monthlist$ is valid only if $dformat$ is 2, 5, 6, 7, 9, 15, 18, or 20. If $dformat$ is any other value $SZDATETIMEH$ ignores $monthlist$.

The $monthlist$ string has the following format:

- The first character of the string is a delimiter character (usually a space). The same delimiter must appear before the first month name and between each month name in $monthlist$. You can specify any single-character delimiter; this delimiter must be specified between the month, day, and year portions of the specified $datetime$ value, which is why a space is usually the preferred character.
- The month names string should contain twelve delimited values, corresponding to January through December. It is possible to specify more or less than twelve month names, but if there is no month name corresponding to the month in $datetime$ an <ILLEGAL VALUE> error is generated.

If you omit $monthlist$ or specify a $monthlist$ value of -1, $SZDATETIMEH$ uses the list of month names defined in the $MonthAbbr$ or $MonthName$ property of the current locale, unless one of the following is true: If $localeopt=1$, the $monthlist$ default is the ODBC month list (in English). If $localeopt$ is unspecified and $dformat$ is 18 or 20 (Islamic date formats) the $monthlist$ default is the Islamic month list (Arabic names expressed using Latin letters), ignoring the $MonthAbbr$ or $MonthName$ property value.

To determine the default month names and month abbreviations for your locale, invoke the $GetFormatItem()$ NLS class method:

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("MonthName"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("MonthAbbr"),!
```

$yearopt$

With $dformat$ values 0, 1, 2, 4, or 7, a numeric code that specifies the time window in which to display the year as a two-digit value. $yearopt$ can be:
### YearOption

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Get effective <code>yearopt</code> value from <code>YearOption</code> property of current locale which defaults to 0. This is the default behavior if you do not specify <code>yearopt</code>.</td>
</tr>
<tr>
<td>0</td>
<td>Represent 20th century dates (1900 through 1999) with two-digit years, unless a process-specific sliding window (established via the <code>%DATE</code> utility) is in effect. If such a window is in effect, represent only those dates falling within the sliding window by two-digit years. Represent all dates falling outside the 20th century or outside the process-specific sliding window by four-digit years.</td>
</tr>
<tr>
<td>1</td>
<td>Represent 20th century dates with two-digit years and all other dates with four-digit years.</td>
</tr>
<tr>
<td>2</td>
<td>Represent all dates with two-digit years.</td>
</tr>
<tr>
<td>3</td>
<td>Represent with two-digit years those dates falling within the sliding temporal window defined by <code>startwin</code> and (optionally) <code>endwin</code>. Represent all other dates with four-digit years. When <code>yearopt</code> = 3, <code>startwin</code> and <code>endwin</code> are absolute dates in <code>$HOROLOG</code> format.</td>
</tr>
<tr>
<td>4</td>
<td>Represent all dates with four-digit years.</td>
</tr>
<tr>
<td>5</td>
<td>Represent all dates with two-digit years all dates falling within the sliding window defined by <code>startwin</code> and (optionally) <code>endwin</code>. Represent all other dates with four-digit years. When <code>yearopt</code>=5, <code>startwin</code> and <code>endwin</code> are relative years.</td>
</tr>
<tr>
<td>6</td>
<td>Represent all dates in the current century with two-digit years and all other dates with four-digit years.</td>
</tr>
</tbody>
</table>

If you omit `yearopt` or specify a `yearopt` value of -1, `$ZDATETIMEH` uses the `YearOption` property of the current locale, unless one of the following is true: If `localeopt`=1, the `yearopt` default is the ODBC year option. If `localeopt`=0 or is unspecified and `dformat` is 18, 19, 20, or 21 (Islamic date formats) the `yearopt` default is the ODBC year option (4-digit years); the `YearOption` property value is ignored for Islamic dates.

#### startwin

A numeric value that specifies the start of the sliding window during which dates must be represented with two-digit years. You must supply `startwin` when you use a `yearopt` of 3 or 5. `startwin` is not valid with any other `yearopt` values.

When `yearopt`=3, `startwin` is an absolute date in `$HOROLOG` date format that indicates the start date of the sliding window.

When `yearopt`=5, `startwin` is a numeric value that indicates the start year of the sliding window expressed in the number of years before the current year. The sliding window always begins on the first day of the year (January 1) specified in `startwin`.

#### endwin

A numeric value that specifies the end of the sliding window during which dates are represented with two-digit years. You may optionally supply `endwin` when `yearopt` is 3 or 5. `endwin` is not valid with any other `yearopt` values.

When `yearopt`=3, `endwin` is an absolute date in `$HOROLOG` date format that indicates the end date of the sliding window.

When `yearopt`=5, `endwin` is a numeric value that indicates the end year of the sliding window expressed as the number of years past the current year. The sliding window always ends on December 31st of the year specified in `endwin`. If `endwin` is not specified, it defaults to December 31st of the year 100 years after `startwin`.

If `endwin` is omitted (or specified as -1) the effective sliding window will be 100 years long. The `endwin` value of -1 is a special case that always returns a date value, even when higher and lower `endwin` values return `erropt`. For this reason, it is preferable to omit `endwin` when specifying a 100-year window, and to avoid the use of negative `endwin` values.
If you supply both *startwin* and *endwin*, the sliding window they specify must not have a duration of more than 100 years.

**mindate**

An expression that specifies the lower limit of the range of valid dates (inclusive). Can be specified as a **SHOROLOG** integer date count (for example, 1/1/2013 is represented as 62823) or a **SHOROLOG** string value. You can include or omit the time portion of a **SHOROLOG** date string (for example “62823,43200”), but only the date portion of *mindate* is parsed. Specifying an *datetime* value earlier than *mindate* generates a <VALUE OUT OF RANGE> error.

The following are supported *mindate* values:

- Positive integer: Most commonly *mindate* is specified as a positive integer to establish the earliest allowed date as some date after December 31, 1840. For example, a *mindate* of 21550 would establish the earliest allowed date as January 1, 1900. The highest valid value is 2980013 (December 31, 9999).
- 0: specifies the minimum date as December 31, 1840. This is the DateMinimum property default.
- Negative integer -2 or larger: specifies a minimum date counting backwards from December 31, 1840. For example, a *mindate* of -14974 would establish the earliest allowed date as January 1, 1800. Negative *mindate* values are only meaningful if the DateMinimum property of the current locale has been set to an equal or greater negative number. The lowest valid value is -672045.
- If omitted (or specified as -1), *mindate* defaults to the DateMinimum property value for the current locale, unless one of the following is true: If localeopt=1, the *mindate* default is 0. If localeopt is unspeciﬁed and dformat=3, the *mindate* default is 0. If localeopt is unspeciﬁed and dformat is 18, 19, 20, or 21 (Islamic date formats) the *mindate* default is 0.

You can get and set the DateMinimum property as follows:

```objectscript
SET min=#class(%SYS.NLS.Format).GetFormatItem("DateMinimum")
WRITE "initial DateMinimum value is ",min,!
Permit18thCenturyDates
  SET x=#class(%SYS.NLS.Format).SetFormatItem("DateMinimum",-51498)
  SET newmin=#class(%SYS.NLS.Format).GetFormatItem("DateMinimum")
  WRITE "set DateMinimum value is ",newmin,!!
RestrictTo19thCenturyDates
  WRITE $ZDATETIMEH("05/29/1805 12:00:00","1","","-14974"),!!
ResetDateMinimumToDefault
  SET oldmin=#class(%SYS.NLS.Format).SetFormatItem("DateMinimum",min)
  WRITE "reset DateMinimum value from ",oldmin," to ",min
```

You may specify *mindate* with or without *maxdate*. Specifying a *mindate* larger than *maxdate* generates an <ILLEGAL VALUE> error.

**maxdate**

An expression that speciﬁes the upper limit of the range of valid dates (inclusive). Can be specified as a **SHOROLOG** integer date count (for example, 1/1/2100 is represented as 94599) or a **SHOROLOG** string value. You can include or omit the time portion of the **SHOROLOG** date (for example “94599,43200”), but only the date portion of *maxdate* is parsed.

If *maxdate* is omitted or if speciﬁed as -1, the maximum date limit is obtained from the DateMaximum property of the current locale, which defaults to the maximum permissible value for the date portion of **SHOROLOG**: 2980013 (corresponding to December 31, 9999). However, the application of the DateMaximum property is governed by the localeopt setting. When localeopt=1 (which is the default for dformat=3) the date maximum default is the ODBC value (2980013), regardless of the current locale setting. Islamic date formats also take the ODBC default. The maximum date for Thai date format (dformat=13) is 31/12/9999 BE, which corresponds to **SHOROLOG** 2781687.

Specifying a *datetime* larger than *maxdate* generates a <VALUE OUT OF RANGE> error.

Specifying a *maxdate* larger than 2980013 generates an <ILLEGAL VALUE> error.
You may specify maxdate with or without mindate. Specifying a maxdate smaller than mindate generates an <ILLEGAL VALUE> error.

**erropt**

Specifying a value for this parameter suppresses errors associated with invalid or out of range datetime values. Instead of generating <ILLEGAL VALUE> or <VALUE OUT OF RANGE> errors, the $ZDATETIMEH function returns the erropt value.

Caché performs standard numeric evaluation on datetime, which must evaluate to an integer date within the mindate/maxdate range. Thus, 7, "7", +7, 0007, 7.0, "7 dwarves", and --7 all evaluate to the same date value: 01/07/1841. By default, values greater than 2980013 or less than 0 generate a <VALUE OUT OF RANGE> error. Fractional values generate an <ILLEGAL VALUE> error. Non-numeric strings (including the null string) evaluate to 0, and thus return the $HOROLOG initial date: 12/31/1840.

The erropt parameter only suppresses errors generated due to invalid or out of range values of datetime. Errors generated due to invalid or out of range values of other parameters will always generate errors whether or not erropt has been supplied. For example, an <ILLEGAL VALUE> error is always generated when $ZDATETIMEH specifies a sliding window where endwin is earlier than startwin. Similarly, an <ILLEGAL VALUE> error is generated when maxdate is less than mindate.

**localeopt**

This Boolean parameter specifies either the user’s current locale definition or the ODBC locale definition as the source for defaults for the locale-specified parameters dformat, tformat, monthlist, yearopt, mindate and maxdate:

- If localeopt=0, all of these parameters take the current locale definition defaults.
- If localeopt=1, all of these parameters take the ODBC defaults.
- If localeopt is not specified, the dformat parameter determine the default for these parameters. If dformat=3, the ODBC defaults are used. If dformat is 18, 19, 20, or 21 the Islamic date and time format defaults are used, regardless of the current locale definition. For all other dformat values, the current locale definition defaults are used. Refer to the dformat description for further details.

The ODBC standard locale cannot be changed; it is used to format date and time strings that are portable between Caché processes that have made different National Language Support (NLS) choices. The ODBC locale date and time definitions are as follows:

- Date format defaults to 3. Therefore, if dformat is undefined or -1, date format 3 is used.
- Date separator defaults to "/". However, date format defaults to 3, which always uses "/" as the date separator.
- Year option defaults to 4 digits.
- Date minimum and maximum are 0 and 2980013 ($HOROLOG date count).
- English month names, month abbreviations, weekday names, weekday abbreviations, and the words "Noon" and "Midnight" are used.
- Time format defaults to 1. Time separator is ":". Time precision is 0 (no fractional seconds). AM and PM indicators are "AM" and "PM".

**Notes**

**$ZDATETIMEH and Fractional Seconds**

Unlike $ZDATETIME, $ZDATETIMEH does not allow you to specify a precision for the time. Any fractional seconds in the original $ZDATETIME-formatted time are retained in the value $ZDATETIMEH returns.

Note that $HOROLOG does not return fractional seconds.
**Invalid Values with $ZDATETIMEH**

You receive a <FUNCTION> error in the following conditions:

- If you specify an invalid dformat code (an invalid integer value, or a non-integer value.)
- If you specify an invalid value for tformat (an integer value less than -1 or greater than 8, a zero, or a non-integer value.)
- If you do not specify a startwin value when yearopt is 3 or 5.

You receive an <ILLEGAL VALUE> error under the following conditions:

- If you specify an invalid value for any date/time unit. If specified, the erropt value is returned rather than issuing an <ILLEGAL VALUE>.
- If you specify excess leading zeros for any date/time unit in an ODBC date. For example, you can represent the February 3, 2007 as “2007–2–3” or “2007–02–03”, but will receive an <ILLEGAL VALUE> for “2007–002–03”. If specified, the erropt value is returned rather than issuing an <ILLEGAL VALUE>.
- If the given month number is greater than the number of month values in monthlist.
- If maxdate is less than mindate.
- If endwin is less than startwin.
- If startwin and endwin specify a sliding temporal window whose duration is greater than 100 years.

You receive a <VALUE OUT OF RANGE> error under the following conditions:

- If you specify a date (or an offset to “T”) which is earlier than Dec. 31, 1840 or later than Dec. 31, 9999, and do not supply an erropt value.
- If you specify an otherwise valid date (or an offset to “T”) which is outside the range of mindate and maxdate and do not supply an erropt value.

**The Current Date**

The following examples shows how you can use the “T” or “t” letter code to specify the current date. Note that dformat must be 5, 6, 7, 8, 9, or 15.

The current date with the time defaulting to 0:

```objectscript
WRITE $ZDATETIMEH("T",5)
```

Three days before the current date, with the time defaulting to 0:

```objectscript
WRITE $ZDATETIMEH("T-3",5)
```

Two days after the current date, with a specified time:

```objectscript
WRITE $ZDATETIMEH("T+2 11:45:00",5)
```

**$ZDATETIMEH Compared to $ZDATEH**

$ZDATETIMEH is similar to $ZDATEH except it converts both a date and a time value to the internal $HOROLOG format (even if no time value is specified.) $ZDATEH only converts a date value to $HOROLOG format. For example:

```objectscript
WRITE $ZDATEH("Nov 25, 2002",5)
```
returns 59133.

```objectscript
WRITE $ZDATETIMEH("Nov 25, 2002 10:08:09.539",5)
```
returns 59133.36489.539.

Specifying `$ZDATETIMEH` with no time value:

```
WRITE $ZDATETIMEH("Nov 25, 2002", 5)
```

returns 59133.0.

Specifying `$ZDATETIMEH` with no time value, and a `tformat` of 7 or 8:

```
WRITE $ZDATETIMEH("Nov 25, 2002", 5, 7)
```

returns a value such as: 59133.68400, where the time value is the local time zone offset from midnight. In this case, U.S. Eastern Standard Time is 5 hours offset from UTC, so the time value here represents 19:00 (5 hours offset from midnight).

**See Also**

- JOB command
- `$ZDATE` function
- `$ZDATEH` function
- `$ZDATETIME` function
- `$ZTIME` function
- `$ZTIMEH` function
- `$HOROLOG` special variable
- `$ZTIMESTAMP` special variable
- `%DATE` utility, which is documented in the “Legacy Documentation” chapter in *Using InterSystems Documentation*
- More information on locales in the section on “System Classes for National Language Support” in *Caché Specialized System Tools and Utilities*. 
$ZDCHAR

Converts a $DOUBLE floating point number to an eight-byte string.

$ZDCHAR(n)  $ZDC(n)

Parameter

| n  | An IEEE-format floating point number. It can be specified as a value, a variable, or an expression. |

Description

$ZDCHAR returns an eight-byte (quad) character string corresponding to \( n \). The bytes of the character string are presented in little-endian byte order, with the least significant byte first.

The number \( n \) can be a positive or negative IEEE floating point number. If \( n \) is not numeric, $ZDCHAR returns the empty string. For further details on IEEE floating point numbers, refer to the $DOUBLE function.

Example

The following examples return an eight-byte string corresponding to the IEEE floating point number:

```objectscript
WRITE $ZDCHAR($DOUBLE(1.4)),!
WRITE $ZDCHAR($DOUBLE(1.400000000000001))
```

These two functions return: "ffffffö?" and "kffffö?"

Notes

$ZDCHAR and Other $CHAR Functions

$ZDCHAR converts an IEEE floating point number to a eight byte (64-bit) character string. $ZQCHAR converts an integer to an eight byte (64-bit) character string. To convert an integer to an 8-bit character string use $CHAR. To convert an integer to a 16-bit (wide) character string use $ZWCHAR. To convert an integer to a 32-bit (long) character string use $ZLCHAR.

See Also

- $DOUBLE function
- $ZDASCII function
- $CHAR function
- $ZWCHAR function
- $ZLCHAR function
- $ZQCHAR function
$ZEXP

Exponential function (inverse of natural logarithm).

$ZEXP (n)

Parameter

| n     | A number of any type. A number larger than 335.6 results in a <MAXNUMBER> error. A number smaller than -295.4 returns 0. |

Description

$ZEXP$ is the exponential function $e^n$, where $e$ is the constant 2.718281828. Therefore, to return the value of $e$, you can specify $ZEXP (1)$. $ZEXP$ is the inverse of the natural logarithm function $ZLN$.

Parameter

$n$

Any number. It can be specified as a value, a variable, or an expression. A positive value larger than 335.6 or smaller than -4944763837 results in a <MAXNUMBER> error. A negative value smaller than -295.4 returns 0. A value of zero (0) returns 1. A non-numeric string is evaluated as 0 and therefore returns 1.

Examples

The following example demonstrates that $ZEXP$ is the inverse of $ZLN$:

```
SET x=7
WRITE $ZEXP(x),!
WRITE $ZLN(x),!
WRITE $ZEXP($ZLN(x))
```

The following example returns $ZEXP$ for negative and positive integers and for zero. This example returns the constant $e$ as $ZEXP (1)$:

```
FOR x=-3:1:3 {
   WRITE !,"The exponential of ",x," = ",$ZEXP(x)
}
QUIT
```

returns:

The exponential of -3 = .04978706836786394297
The exponential of -2 = .1353352832366126919
The exponential of -1 = .367879441171422316
The exponential of 0 = 1
The exponential of 1 = 2.718281828459045236
The exponential of 2 = 7.389056098930650228
The exponential of 3 = 20.0855369231876674

The following example uses IEEE floating point numbers ($DOUBLE$ numbers). The first $ZEXP$ returns a numeric value, the second $ZEXP$ returns "INF" (or <MAXNUMBER> depending on the IEEEError() method setting):

```
SET rtn=#class(%SYSTEM.Process).IEEEError(0)
WRITE $ZEXP($DOUBLE(1.0E2)),!
WRITE $ZEXP($DOUBLE(1.0E3))
```

The following example demonstrates that the empty string or a nonnumeric value is treated as 0:

```
WRITE $ZEXP(""),!
WRITE $ZEXP("INF")
```
both return 1.

See Also

- $ZLN function
$ZF

Invokes non-ObjectScript programs or functions from ObjectScript routines.

$ZF("function_name",args)

**Parameters**

<table>
<thead>
<tr>
<th>function_name</th>
<th>The name of the function you want to call.</th>
</tr>
</thead>
<tbody>
<tr>
<td>args</td>
<td>Optional — A set of argument values passed to the function.</td>
</tr>
</tbody>
</table>

**Description**

The various forms of the $ZF function allow you to invoke non-ObjectScript programs (such as shell or operating system commands) or functions from ObjectScript routines. You can define interfaces or links to functions written in other languages into Caché and call them from ObjectScript routines using $ZF.

$ZF can also be used to:

- Spawn a child process to execute a program or command: $ZF(-100).
- Load a Dynamic Link Library (DLL) then execute functions from that library: $ZF(-3), $ZF(-4), $ZF(-5), and $ZF(-6).

These implementations of $ZF take a negative number as the first parameter. They are described in their own reference pages.

**Parameters**

*function name*

The name of the function you want to call enclosed in quotation marks, or a negative number.

*args*

The args parameters are in the form: arg1, arg2, arg3, ...argn. The arguments can consist of such items as descriptions of how arguments are passed and the entry point to the C function you are calling.

**Notes**

**Calling UNIX® System Services with $ZF**

Caché supports error checking functions for use with UNIX® system calls from $ZF. These calls allow you to check for asynchronous events and to set an alarm handler in $ZF. By using these UNIX® functions you can distinguish between real errors, <CTRL-C> interrupts, and calls that should be restarted.

The function declarations are included in cdzf.h and are described in the following table:

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Purpose</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>int sigrtclr();</td>
<td>Clears retry flag.</td>
<td>Should be called once before using sigrchk()</td>
</tr>
<tr>
<td>int dzfalarm();</td>
<td>Establishes new SIGALRM handler.</td>
<td>On entry to $ZF, the previous handler is automatically saved. On exit, it is restored automatically. A user program should not alter the handling of any other signal.</td>
</tr>
</tbody>
</table>
### Declaration

| int sigrtchk();                        | Checks for asynchronous events. | Should be called whenever one of the following system calls fails: open(), close(), read(), write(), ioctl(), pause(), any call that fails when the process receives a signal. It returns a code indicating the action the user should take:
-1 = Not a signal. Check for I/O error. See contents of errno variable.
0 = Other signal. Restart operation from point at which it was interrupted.
1 = SIGINT/. Exit from $ZF with a SIGTERM "return 0." The System traps these signals appropriately.

---

In a typical $ZF function used to control some device, you would code something like this:

```c
IF ((fd = open(DEV_NAME, DEV_MODE)) < 0) {
  Set some flags
  Call zferror
  return 0;
}
```

The `open` system call can fail if the process receives a signal. Usually this situation is not an error and the operation should be restarted. Depending on the signal, however, you might take other actions. So, to take account of all the possibilities, consider using the following C code:

```c
sigrtclr();
WHILE (TRUE) {
  IF (sigrtchk() == 1) { return 1 or 0; }
  IF ((fd = open(DEV_NAME, DEV_MODE)) < 0) {
    switch (sigrtchk()) {
      case -1:
        /* This is probably a real device error */
        Set some flags
        Call zferror
        return 0;
      case 0:
        /* A innocuous signal was received. Restart. */
        continue;
      case 1:
        /* Someone is trying to terminate your job. */
        Do cleanup work
        return 1 or 0;
    }
  } ELSE { break; }
  /* Code to handle the normal situation: */
  /* open() system call succeeded */
}
```

Remember you must not set any signal handler except through dzfalarm().

### Translating Strings between Encoding Systems

Caché supports input-output translation via a $ZF argument type, t (or T), which can be specified in the following formats:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>Specifies the current process I/O translation object.</td>
</tr>
<tr>
<td>t/</td>
<td>Specifies the default process I/O translation table name.</td>
</tr>
<tr>
<td>t/name/</td>
<td>Specifies a particular I/O translation table name.</td>
</tr>
</tbody>
</table>

$ZF conveys the translated string to the external procedure via a counted-byte string placed in the following C structure:
typedef struct zarray {
    unsigned short len;
    unsigned char data[1]; /* 1 is a dummy value */
} *ZARRAYP;

This is also the structure used for the b (or B) argument type.

The following $ZF sample function performs a round trip conversion:

```c
#include cdzf.h
extern int trantest();

ZFBEGIN
ZFENTRY("TRANTEST","t/SJIS/ T/SJIS/",trantest)
ZFEND

int trantest(inbuf,outbuf);

ZARRAYP inbuf; /* Buffer containing string that was converted from internal Caché encoding to SJIS encoding before it was passed to this function */
ZARRAYP outbuf; /* Buffer containing string in SJIS encoding that will be converted back to internal Caché encoding before it is passed back into the Caché environment */
{
    int i;
    /* Copy data one byte at a time from the input argument buffer to the output argument buffer */
    for (i = 0; i < inbuf->len; i++)
        outbuf->data[i] = inbuf->data[i];
    /* Set number of bytes of data in the output argument buffer */
    outbuf->len = inbuf->len;
    return 0; /* Return success */
}
```

**Note:** Conceptually speaking, data flows to and from a $ZF external procedure, as if the external procedure were a device. The output component of an I/O translation is used for data that is passed to an external procedure because the data is “leaving” the Caché environment. The input component of an I/O translation is used for data that is received from an external procedure because the data is “entering” the Caché environment.

If the output component of an I/O translation is undefined and your application attempts to pass anything but the null string using that I/O translation, Caché returns an error, because it does not know how to translate the data.

If the input component of an I/O translation is undefined and an argument of type string associates that I/O translation with a $ZF output argument, Caché returns an error, because an output argument with an undefined translation is purposeless.

### Zero-Terminated and Counted Unicode Strings

The $ZF function supports argument types for zero-terminated Unicode strings and counted Unicode strings. These are supported even in versions of Caché that do not use Unicode characters internally.

The argument types for zero-terminated Unicode strings and counted Unicode strings have the following codes:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>Pointer to a zero-terminated Unicode character string.</td>
</tr>
<tr>
<td>s</td>
<td>Pointer to a counted Unicode character string.</td>
</tr>
</tbody>
</table>

For both argument types, the C data type of the Unicode character is an unsigned short. A pointer to a zero-terminated Unicode string is declared as follows:

```c
unsigned short *p;
```

A pointer to a counted Unicode string is declared as a pointer to the following C structure:
typedef struct zwarray {
    unsigned short len;
    unsigned short data[1]; /* 1 is a dummy value */
} *ZWARRAYP;

For example:

ZWARRAYP *p;

The `len` field contains the length of the Unicode character array.

The `data` field contains the characters in the counted Unicode string. The maximum size of a Unicode string is the maximum `ZF` string size, which is an updateable configuration parameter that defaults to 32767.

Each Unicode character is two bytes long. This is important to consider when declaring Unicode strings as output arguments, because Caché reserves space for the longest string that may be passed back. When using the default string size, the total memory consumption for a single Unicode string argument is calculated as follows:

32767 maximum characters * 2 bytes per character = 65534 total bytes.

This is close to the default maximum memory area allocated for all `ZF` arguments, which is 67584. This maximum `ZF` heap area is also an updateable configuration parameter.

**Error Messages**

When the `ZF` heap area is exhausted, `ZF` issues an `<OUT OF ZF HEAP SPACE>` error. When the `ZF` String Stack is exhausted, `ZF` issues a `<STRINGSTACK>` error. When `ZF` is unable to allocate a buddyblock, it issues a `<STORE>` error.

**Execution from Child Processes and DLLs**

The `ZF` function can take a negative number as its first parameter. These negative numbers specify functions that support spawned child processes and Dynamic-Link Libraries (DLLs). Each of these `ZF` functions is described in a separate reference page.

**See Also**

- `ZF(-100)` function
- `ZF(-1)` function (deprecated)
- `ZF(-2)` function (deprecated)
- `ZF(-3)` function
- `ZF(-4)` function
- `ZF(-5)` function
- `ZF(-6)` function
- Using the Caché Callout Gateway
$ZF(-1)

Executes an operating system command or program as a child process, synchronously. (Deprecated)

$ZF(-1,program)

**Parameters**

| program | Optional — The operating system command or program to be executed as a child process, specified as a quoted string. If you omit program, $ZF(-1) launches the operating system shell. |

**Description**

**Note:** $ZF(-1) is a deprecated function. It is described here for compatibility with existing code only. All new code development should use $ZF(-100).

$ZF(-1) permits a Caché process to invoke a program or a command of the host operating system. It executes the program or command specified in program as a spawned child process from the current console. It executes synchronously; it waits for the process to return. $ZF(-1) returns the child process exit status.

$ZF(-1) returns the following status codes:

- It returns 0 if the child process executed successfully.
- It returns a positive integer based on the exit status error code issued by the operating system shell. This integer exit status code value is determined by the host operating system. For example, for most Windows command syntax errors, $ZF(-1) returns 1.
- It returns -1 if the child process could not be forked.

Because $ZF(-1) waits for a response from the spawned child process, you cannot successfully shut down Cache while the child process is executing.

$ZF(-1) with no specified parameters launches the default operating system shell. For further details, see “Issuing Operating System Commands” in *Using the Callout Gateway*.

If a pathname supplied in program contains a space character, pathname handling is platform-dependent. Windows and UNIX® permit space characters in pathnames, but the entire pathname containing spaces must be enclosed in an additional set of double quote (") characters. This is in accordance with the Windows cmd /c statement. For further details, specify cmd /? at the Windows command prompt.

You can use the NormalizeFilenameWithSpaces() method of the %Library.File class to handle spaces in pathnames as appropriate for the host platform.

$ZF(-1) requires the %System_Callout:U privilege. See “Adding the %System_Callout:USE Privilege” in *Using the Callout Gateway* for details.

If $ZF(-1) is unable to spawn a process, it generates a <FUNCTION> error.

At the Terminal in the Terminal, you can perform operations similar to $ZF(-1) by using an exclamation point (!) or a dollar sign ($) as the first character, followed by the operating system command you wish to execute. The ! or $ command line prefix executes the operating system command, returns results from the invoked process and displays those results at the Terminal. $ZF(-1) does not return operating system command results; it executes the operating system command, then returns the exit status code for the invoked process. For further details, see “Issuing Operating System Commands” in *Using the Callout Gateway*. 
**Auditing**

An OS command audit record is added to the audit log for each $ZF(-1) call. This record includes information such as the following:

Execute O/S command Directory: c:\182u5\mgr\ Command: ls -lt 4002

$ZF(-1), $ZF(-2), and $ZF(-100)

These three functions are in most respects identical. $ZF(-100) is the preferred function for all purposes, replacing both $ZF(-1) and $ZF(-2).

- **$ZF(-1)** executes using the OS shell. It is synchronous; it suspends execution of the current process while awaiting completion of the spawned child process. It receives status information from the spawned process, which it returns as an exit status code (an integer value) when the spawned process completes. $ZF(-1) does not set $ZCHILD.
- **$ZF(-2)** executes using the OS shell. It is asynchronous; it does not suspend execution of the current process. It immediately returns a status value upon spawning the child process. Because it does not await completion of the spawned child process it cannot receive status information from that process. $ZF(-2) sets $ZCHILD if its fifth argument is true.
- **$ZF(-100)** can be synchronous or asynchronous. It can execute using the operating system shell or not using the shell. It always sets $ZCHILD. Both $ZF(-1) and $ZF(-2) with no specified parameters launch the operating system shell; $ZF(-100) requires a program parameter (and the /SHELL flag) to launch the operating system shell.

**Examples**

The following Windows example executes a user-written program, in this case displaying the contents of a .txt file. It uses NormalizeFilenameWithSpaces() to handle a pathname for $ZF(-1). A pathname containing spaces is handled as appropriate for the host platform. A pathname that does not contain spaces is passed through unchanged.

```
SET fname="C:\My Test.txt"
WRITE fname,!
SET x=#class(%Library.File).NormalizeFilenameWithSpaces(fname)
WRITE x,!
WRITE $ZF(-1,x)
```

The following Windows example invokes the Windows operating system SOL command. SOL opens a window that displays the Solitaire game provided with the Windows operating system. Upon closing of the Solitaire interactive window, $ZF(-1) returns the Windows shell exit status of 0, indicating success:

```
SET x=$ZF(-1,"SOL")
WRITE x
```

The following Windows example invokes a non-existent Windows operating system command. $ZF(-1) returns the Windows shell exit status of 1, indicating a syntax error:

```
WRITE $ZF(-1,"SOX")
```

The following Windows example invokes a Windows operating system command, specifying a non-existent network name. $ZF(-1) returns the Windows shell exit error status of 2:

```
WRITE $ZF(-1,"NET USE :k \ \bogusname")
```

**See Also**

- $ZF(-2) function
- $ZF(-100) function
• Issuing Operating System Commands in *Using the Callout Gateway*
$ZF(-2)

Executes an operating system command or program as a child process, asynchronously. (Deprecated)

$ZF(-2,program)

Parameter

| program | Optional — The operating system command or program to be executed as a child process, specified as a quoted string. If you omit program, $ZF(-2) launches the operating system shell. |

Description

Note: $ZF(-2) is a deprecated function. It is described here for compatibility with existing code only. All new code development should use $ZF(-100).

$ZF(-2) permits a Caché process to invoke a program or a command of the host operating system. $ZF(-2) executes the operating system command specified in program as a spawned child process from the current console. It executes asynchronously; it returns immediately after spawning the child process and does not wait for the process to terminate. Input and output devices default to the null device.

$ZF(-2) does not return the child process exit status. Instead, if the child process was created successfully, $ZF(-2) returns 0. $ZF(-2) returns -1 if a child process could not be forked.

Because $ZF(-2) does not wait for a response from the spawned child process, you can shut down Caché while the child process is executing.

$ZF(-2) closes the parent process principal device (specified in $PRINCIPAL) before executing the operating system command. This is done because the child process executes concurrently with the parent. If $ZF(-2) did not close $PRINCIPAL, output from the parent and the child would become intermingled. When using $ZF(-2) you should redirect I/O in the command if you wish to recover output from the child process. For example:

```
SET x=$ZF(-2,"ls -l > mydir.txt")
```

$ZF(-2) with no specified parameters launches the default operating system shell. For further details, see “Issuing Operating System Commands” in Using the Callout Gateway.

If a pathname supplied in program contains a space character, pathname handling is platform-dependent. Windows and UNIX® permit space characters in pathnames, but the entire pathname containing spaces must be enclosed in an additional set of double quote (") characters. This is in accordance with the Windows cmd /c statement. For further details, specify cmd /? at the Windows command prompt.

You can use the NormalizeFilenameWithSpaces() method of the %Library.File class to handle spaces in pathnames as appropriate for the host platform.

$ZF(-2) is a privileged operation, which requires the %System_Callout:USE Privilege. See “Adding the %System_Callout:USE Privilege” in Using the Callout Gateway for details.

Auditing

An OS command audit record is added to the audit log for each $ZF(-2) call. This record includes information such as the following:

```
Execute O/S command Directory: c:\182u5\mgr\ Command: ls -lt 4002 – Detached
```
The Detached keyword indicates the call is \$ZF(-2); a \$ZF(-1) call does not have this keyword.

\$ZF(-2), \$ZF(-1), and \$ZF(-100)

These three functions are in most respects identical. \$ZF(-100) is the preferred function for all purposes, replacing both \$ZF(-1) and \$ZF(-2).

- \$ZF(-2) executes using the OS shell. It is asynchronous; it does not suspend execution of the current process. It immediately returns a status value upon spawning the child process. Because it does not await completion of the spawned child process it cannot receive status information from that process. \$ZF(-2) sets \$ZCHILD if its fifth argument is true.

- \$ZF(-1) executes using the OS shell. It is synchronous; it suspends execution of the current process while awaiting completion of the spawned child process. It receives status information from the spawned process, which it returns as an exit status code (an integer value) when the spawned process completes. \$ZF(-1) does not set \$ZCHILD.

- \$ZF(-100) can be synchronous or asynchronous. It can execute using the operating system shell or not using the shell. It always sets \$ZCHILD. Both \$ZF(-1) and \$ZF(-2) with no specified parameters launch the operating system shell; \$ZF(-100) requires a program parameter (and the /SHELL flag) to launch the operating system shell.

See Also

- \$ZF(-1) function
- \$ZF(-100) function
- \$PRINCIPAL special variable
- Issuing Operating System Commands in Using the Callout Gateway
- Adding the %System_Callout:USE Privilege in Using the Callout Gateway
$ZF(-3)

Loads a Dynamic-Link Library (DLL) and executes a library function.

$ZF(-3,dll_name,func_name,args)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dll_name</td>
<td>The name of the dynamic-link library (DLL) to load, specified as a quoted string. When a DLL is already loaded, dll_name can be specified as a null string (“”).</td>
</tr>
<tr>
<td>func_name</td>
<td>Optional — The name of the function to execute within the DLL, specified as a quoted string.</td>
</tr>
<tr>
<td>args</td>
<td>Optional — A comma-separated list of arguments to pass to the function.</td>
</tr>
</tbody>
</table>

Description

Use $ZF(-3) to load a Dynamic-Link Library (DLL) and execute the specified function from that DLL. $ZF(-3) returns the function’s return value.

$ZF(-3) can be invoked in any of the following ways:

To just load a DLL:

```
SET x=$ZF(-3,"mydll")
```

To load a DLL and execute a function located in that DLL:

```
SET x=$ZF(-3,"mydll","$$myfunc",1)
```

Loading a DLL using $ZF(-3) makes it the current DLL, and automatically unloads the DLL loaded by a previous invocation of $ZF(-3).

To execute a function located in a DLL loaded by a previous $ZF(-3), you can speed execution by specifying the current DLL using the null string, as follows:

```
SET x=$ZF(-3,"","$$myfunc2",1)
```

To explicitly unload the current DLL (loaded by a previous $ZF(-3) call):

```
SET x=$ZF(-3,")")
```

$ZF(-3) can load only one DLL. Loading a DLL unloads the previous DLL. You can also explicitly unload the currently loaded DLL, which would result in no currently loaded DLL. (However, note that $ZF(-5) loads and unloads do not affect loads and unloads for use with $ZF(-5) or $ZF(-6), as described below.)

The DLL name specified can be a full pathname, or a partial pathname. If you specify a partial pathname, Caché canonicalizes it to the current directory. Generally, DLLs are stored in the binary directory (“bindir”). To locate the binary directory, call the BinaryDirectory() method of the %SYSTEM.Util class. For further details, refer to the InterSystems Class Reference.

Notes

**Dynamic-Link Libraries**

A DLL is a binary library that contains routines that can be loaded and called at runtime. When a DLL is loaded, Caché finds a function named GetZFTable() within it. If GetZFTable() is present, it returns a pointer to a table of the functions located in the DLL. Using this table, $ZF(-3) calls the specified function from the DLL.
**Loading Multiple DLLs**

Calls to $ZF(-3) can only load one DLL at a time; loading a DLL unloads the previous DLL. To load multiple DLLs concurrently, execute DLL functions with $ZF(-5) or $ZF(-6). Loading or unloading a DLL using $ZF(-3) has no effect on DLLs loaded for use with $ZF(-5) or $ZF(-6).

**See Also**

- $ZF(-5) function
- $ZF(-6) function
- Using $ZF(-3) for Simple Library Function Calls in *Using the Caché Callout Gateway*. 

$ZF(-4)$

Provides utility functions used with $ZF(-5)$ and $ZF(-6)$.

$ZF(-4,1,$dll\_name$)$
$ZF(-4,n,$dll\_id$,func\_name)$
$ZF(-4,n,$dll\_id$,decr\_flag)$
$ZF(-4,n,$dll\_index$,dll\_name)$
$ZF(-4,n,$dll\_index$,decr\_flag)$

**Parameters**

| **$n$** | A code for the type of operation to perform: 1=load DLL by name. 2=unload DLL by id. 3=look up function in DLL by id. 4=unload DLL by index. 5= create an entry in the system DLL index table. 6=delete an entry in the system DLL index table. 7= create an entry in the process DLL index table. 8=delete an entry in the process DLL index table. |
| **$dll\_name$** | The name of the dynamic-link library (DLL). Used with $n=1$, 5, or 7. |
| **$dll\_id$** | The id value of a loaded dynamic-link library (DLL). Used with $n=2$, or 3. |
| **$dll\_index$** | A user-defined index to a dynamic-link library (DLL) in a DLL index table. Must be a unique, positive, nonzero integer. The numbers 1024 through 2047 are reserved for system use. Used with $n=4$, 5, 6, 7, or 8. |
| **$func\_name$** | The name of the function to look up within the DLL. Used only when $n=3$. |
| **decr\_flag** | Optional — A flag for decrementing the DLL reference count. Used with $n=2$ or 4. |

**Description**

$ZF(-4)$ can be used to establish an ID value for a DLL or for a function within a DLL. These ID values are used by $ZF(-5)$ to execute a function.

$ZF(-4)$ can be used to establish an index to a DLL index table. These index values are used by $ZF(-6)$ to execute a function.

- You can explicitly load shared libraries using $ZF(-4,1)$, which loads a library and returns a handle that can be used to access library functions with $ZF(-5)$.
- You can explicitly load a single shared library using $ZF(-3)$, which loads a single active library and invokes its methods.
- You can implicitly load shared libraries using $ZF(-6)$, after indexing a library with $ZF(-4,5)$ or $ZF(-4,7)$.  

Caché ObjectScript Reference 557
Establishing ID Values

To load a DLL and return its ID, use the following syntax:

\[
dll\_id=\$ZF(-4,1,dll\_name)
\]

To look up a function from a DLL loaded by \$ZF(-4,1), and return an ID for that function, use the following syntax:

\[
func\_id=\$ZF(-4,3,dll\_id,func\_name)
\]

To execute a function located by \$ZF(-4,3), use \$ZF(-5).

To unload a specific DLL loaded by \$ZF(-4,1), use the following syntax:

\[
\$ZF(-4,2,dll\_id)
\]

To unload all DLLs loaded by \$ZF(-4,1), use the following syntax:

\[
\$ZF(-4,2)
\]

Increment and Decrement DLL Loads

When two classes have loaded the same library, the library will be unloaded by the first call to \$ZF(-4,2,dll\_id) or \$ZF(-4,4,dll\_index). This can leave the other class stranded without access to the library. For this reason, Caché supports a reference count on each DLL. Caché maintains a reference count of the number of times a library is loaded with \$ZF(-4,1,dll\_name). Each call to \$ZF(-4,1,dll\_name) increases the reference count.

\$ZF(-4,2) provides an optional decrement flag argument, \textit{decr\_flag}. Each call to \$ZF(-4,2,dll\_id,1) decrements the reference count by 1. A call to \$ZF(-4,2,dll\_id,1) unloads the library if the reference count goes to zero. A call to \$ZF(-4,2,dll\_id) (or \$ZF(-4,2,dll\_id,0)) ignores the reference count and unloads the library immediately.

A call to \$ZF(-4,5) or \$ZF(-4,7) establishes a library index. Subsequent calls to \$ZF(-6) to execute a function implicitly loads the library and increment the reference count. Each call to \$ZF(-4,4,dll\_index,1) decrements this reference count by 1.

The reference count interactions between reference counts established by \textit{dll\_name} and \textit{dll\_index} are as follows:

- Libraries loaded with \$ZF(-4,1,dll\_name) are not unloaded by a call to \$ZF(-4,4,dll\_index,1) unless the reference count is zero.
- Libraries loaded with \$ZF(-4,1,dll\_name) are immediately unloaded by either \$ZF(-4,2,dll\_id) or \$ZF(-4,4,dll\_index) (with no decrement flag argument) with no regard to the reference count.
- Libraries loaded implicitly with \$ZF(-6) are not unloaded by \$ZF(-4,2,dll\_id,1), even if the reference count goes to zero; they can only be unloaded by \$ZF(-4,4,dll\_index,1).
- Libraries loaded implicitly with \$ZF(-6) are immediately unloaded by either \$ZF(-4,2,dll\_id) or \$ZF(-4,4,dll\_index) (with no decrement flag argument) with no regard to the reference count.

\$ZF(-4,2) with no dll\_id argument unloads all libraries immediately, without regard to the reference count, or whether they were loaded with \$ZF(-4,1,dll\_name) or implicitly with \$ZF(-6).
Establishing Index Values

To index a DLL in the system DLL index table, use the following syntax:

$ZF(-4,5,dll_index,dll_name)

To index a DLL in the process DLL index table, use the following syntax:

$ZF(-4,7,dll_index,dll_name)

To look up and execute a function indexed by $ZF(-4,5) or $ZF(-4,7), use $ZF(-6).

To unload an indexed DLL, use the following syntax:

$ZF(-4,4,dll_index)

To delete an index entry in the system DLL index table, use the following syntax:

$ZF(-4,6,dll_index)

To delete an index entry in the process DLL index table, use the following syntax:

$ZF(-4,8,dll_index)

To delete all index entries in the process DLL index table, use the following syntax:

$ZF(-4,8)

For a detailed description of how to use $ZF(-4) and $ZF(-5), refer to “Using $ZF(-5) to Access Libraries by System ID” in Using the Caché Callout Gateway.

For a detailed description of how to use $ZF(-4) and $ZF(-6), refer to “Using $ZF(-6) to Access Libraries by User Index” in Using the Caché Callout Gateway.

See Also

- $ZF(-3) function
- $ZF(-5) function
- $ZF(-6) function
- Using $ZF(-5) to Access Libraries by System ID in Using the Caché Callout Gateway
- Using $ZF(-6) to Access Libraries by User Index in Using the Caché Callout Gateway
$ZF(-5)

Executes a DLL function loaded using $ZF(-4).

$ZF(-5,dll_id,func_id,args)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dll_id</td>
<td>The ID value for the dynamic-link library (DLL), as supplied by $ZF(-4).</td>
</tr>
<tr>
<td>func_id</td>
<td>The ID value of the function within the DLL as supplied by $ZF(-4).</td>
</tr>
<tr>
<td>args</td>
<td>Optional — One or more arguments passed to the called function.</td>
</tr>
</tbody>
</table>

**Description**

To execute a function located in a DLL loaded using $ZF(-4), use the following syntax:

```
return=$ZF(-5,dll_id,func_id,args)
```

**See Also**

- $ZF(-4) function
- Using $ZF(-5) to Access Libraries by System ID in *Using the Caché Callout Gateway*
$ZF(-6)

Executes a DLL function indexed using $ZF(-4).

$ZF(-6, dll_index, func_ID, args)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dll_index</td>
<td>A user-specified index to a DLL filename in the DLL index tables, from $ZF(-4).</td>
</tr>
<tr>
<td>func_ID</td>
<td>Optional — The ID value of the function within the DLL as supplied by $ZF(-4). If omitted, call verifies the validity of DLL_index, loads the image, and returns the image location.</td>
</tr>
<tr>
<td>args</td>
<td>Optional — The argument(s) to pass to the function, if any, specified as a comma-separated list.</td>
</tr>
</tbody>
</table>

**Description**

$ZF(-6) provides a fast Dynamic Link Library (DLL) function interface using a user-defined index for a DLL filename. You establish this user-defined index in $ZF(-4) by assigning an integer (dll_index) to uniquely associate with a dll_name. You can place this entry in either a process DLL index table, or a system DLL index table.

Both $ZF(-5) and $ZF(-6) can be used to execute a function from a DLL, which has been located by $ZF(-4).

For a detailed description of how to use $ZF(-6), refer to “Using $ZF(-6) to Access Libraries by User Index” in Using the Caché Callout Gateway.

**See Also**

- $ZF(-3) function
- $ZF(-4) function
- $ZF(-5) function
- Using $ZF(-6) to Access Libraries by User Index in Using the Caché Callout Gateway
$ZF(-100)

Executes an operating system command or program as a child process.

$ZF(-100,flags,program,args)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flags</td>
<td>A quoted string containing one or more keyword flags. Multiple keyword flags are separated by blank spaces. A keyword flag can take the format /keyword, /keyword=value, or /keyword+=value. Keywords are not case-sensitive. The flags specify how to execute program.</td>
</tr>
<tr>
<td>program</td>
<td>An operating system command or a program to be executed as a child process, specified as a quoted string. You can specify a full path, or just a program name. The operating system uses its rules, such as a PATH environment variable, to search for the specified program.</td>
</tr>
<tr>
<td>args</td>
<td>Optional — A comma-separated list of program options and arguments. You can specify a null argument as &quot;&quot;. You can use a local array and indirection .args or the args... syntax to specify a variable number of arguments.</td>
</tr>
</tbody>
</table>

**Description**

$ZF(-100) permits a Caché process to invoke an executable program or a command of the host operating system. It executes the program or command specified in program as a spawned child process from the current console. $ZF(-100) allows you to invoke a program or command either synchronously or asynchronously, with or without invoking the operating system shell. $ZF(-100) provides similar functionality to $ZF(-1) and $ZF(-2). Its use is preferable to $ZF(-1) or $ZF(-2), which are both deprecated functions.

You can use a local array and indirection to specify a variable number of args, as shown in the following UNIX® example:

```
SET args=2
SET args(1)="-01"
SET args(2)="myfile.c"
SET status = $ZF(-100,"/ASYNC", "gcc",.args)
```

$ZF(-100) sets $ZCHILD to the PID of the started program.

**Keyword Flags**

How $ZF(-100) executes depends on the flags string values:

- /ASYNC: Execute program asynchronously; do not wait for it to complete. The default is to execute synchronously. If /ASYNC is not specified and /STDIN, /STDOUT, or /STDERR is not specified, Caché attempts to use the operating systems’ current descriptors or standard handles for these files.
- /SHELL: Execute program using a shell. The default is to not use a shell.
- /STDOUT=filename: I/O redirection standard data output file. If filename does not exist, the system will create it. If filename exists, /STDOUT=filename will overwrite existing data; /STDOUT+=filename will append to existing data.
- /STDERR=filename: I/O redirection standard error output file. If filename does not exist, the system will create it. If filename exists, /STDERR=filename will overwrite existing error log data; /STDERR+=filename will append...
to existing error log data. If you specify the same file for /STDOUT and /STDERR, both types of data will be written to that file.

• /LOGCMD: log the resulting command line in cconsole.log. Because sometimes it can be hard to get the arguments for complex commands right, this keyword flag allows developers to check if the arguments passed to the command are being correctly formed (especially with regard to quoting). The log facility does not add any quotes or other delimiters. The cconsole.log entry is truncated at 1000 characters.

• /NOQUOTE: inhibit automatic quoting of commands, command arguments, or filenames. By default, $ZF(-100) provides automatic quoting, and escaping of spaces in paths that is appropriate for most user-supplied values. When needed, you can override this default by specifying /NOQUOTE; the user is then responsible for providing appropriate quotes. See Quoting User-Specified Values.

To specify $ZF(-100) with no keyword flags, specify the empty string for this parameter:

```
SET status = $ZF(-100,"", "ls", "-l")
```

**I/O Redirection**

I/O redirection for /STDIN=filename, /STDOUT=filename, and /STDERR=filename follow UNIX® conventions. On both UNIX and Windows systems:

• /STDIN=filename: The file with that filename is linked to the stdin file handle given to the process that executes the specified cmd string.

• /STDOUT=filename: If a file with that filename does not already exist, it is created. For an existing file, /STDOUT=filename truncates the file to zero size; /STDOUT+=filename appends to the existing file data. This file is linked to the stdout handle given to the process that executes the specified cmd string. This creates a new file containing the stdout output of the spawned command.

• /STDERR=filename: If a file with that filename does not already exist, it is created. For an existing file, /STDERR=filename truncates the file to zero size; /STDERR+=filename appends to the existing file data. This file is linked to the stderr handle given to the process that executes the specified cmd string. This creates a new file containing the stderr output of the spawned command.

If /STDIN, /STDOUT, or /STDERR is not specified:

• If /ASYNC is specified, the null device is used in place of the unspecified file(s). A handle that references the null device is given to the process that executes the specified cmd string as the unspecified file’s handle.

• If /ASYNC is not specified, the handle used by the Caché job executing the $ZF(-100) function is copied and is given to the process that executes the specified cmd string as the unspecified file’s handle.

  **Note:** On a Windows system you should never omit both the /ASYNC and /STDIN flags.

If /STDIN, /STDOUT, or /STDERR specifies a file that cannot be created or opened, the null device is used in place of the file.

If /STDOUT=filename and /STDERR=filename (or /STDOUT+=filename and /STDERR+=filename) specify the same filename, the specified file is only opened or created once. The resulting file handle is duplicated and supplied as both the stdout and stderr file handles given to the process that executes the specified cmd string. $ZF(-100) generates an <ILLEGAL VALUE> error if you specify the same file for /STDOUT and /STDERR, and one is specified +=filename and the other is specified =filename.

**Quoting User-Specified Values**

By default, $ZF(-100) provides automatic quoting of a command and the arguments to the command. It automatically handles blank spaces if your executable is in a directory with spaces in the name or a command argument specifies a file.
for output that contains a space. $ZF(-100) supplies delimiting double quote characters as needed. This behavior is shown in the following example:

```
DO $ZF(-100, "LOGCMD", "c:sdelete64.exe", "," nobanner","c:dirl\nested directory\deleteme")
```

This logs the following to messages.log; $ZF(-100) quoted the final argument to escape the space in the file path:

```
06/14/18-14:25:05:988 (3788) 0 $ZF(-100) cmd=c:sdelete64.exe -nobanner "c:dirl\nested directory\deleteme"
06/14/18-14:25:06:020 (3788) 0 $ZF(-100) ret=0
```

If the automatic quoting provided does not correctly escape what you want escaped, you can override it using /NOQUOTE. The /NOQUOTE flag suppresses automatic quoting; you must do your own quoting, as needed. If a specified value contains a / character or a blank space, the value must be quoted using doubled double quotes. This is shown in the following example:

```
DO $ZF(-100, "NOQUOTE LOGCMD", "c:sdelete64.exe", "," nobanner","","c:dirl\"
```

This logs the following to messages.log:

```
06/15/18-09:27:38:619 (3788) 0 $ZF(-100) cmd=c:sdelete64.exe "nobanner" "c:dirl" ret=0
```

The behavior differs on UNIX® and Windows systems:

- On a Windows system, if /SHELL is not specified, a command line is created and passed. In this case, some arguments may need to be quoted.
- On any system, when /SHELL is specified, a command line is created and passed. In this case some arguments may need to be quoted.

Double quotes found within a command or command argument are escaped. On Windows these double quotes are escaped by doubling them as " " (as shown in the above examples). On UNIX they are escaped as ".

### Return Status Codes

$ZF(-100) returns the following status codes:

- 0 if the child process was successfully launched asynchronously (with /ASYNC flag). Status of program execution unknown.
- -1 if the child process could not be forked.
- An integer if launched synchronously (no /ASYNC flag). This integer exit status code value is determined by the application called on the host operating system. Commonly it is a positive integer, but some applications may return a negative integer. For example, for most Windows command syntax errors, $ZF(-100) returns 1.

$ZF(-100) with the /SHELL parameter launches the default operating system shell. For further details, see “Issuing Operating System Commands” in Using the Callout Gateway.

If a pathname supplied in program contains a space character, pathname handling is platform-dependent. Windows and UNIX® permit space characters in pathnames, but the entire pathname containing spaces must be enclosed in an additional set of double quote (") characters. This is in accordance with the Windows cmd /c statement. For further details, specify cmd /? at the Windows command prompt.

You can use the NormalizeFilenameWithSpaces() method of the %Library.File class to handle spaces in pathnames as appropriate for the host platform.

$ZF(-100) requires the %System_Callout:U privilege. See “Adding the %System_Callout:USE Privilege” in Using the Callout Gateway for details.

If $ZF(-100) is unable to spawn a process, it generates a <FUNCTION> error.
Error Handling

ZF(-100) generates a <NOTOPEN> error if:

- The /STDIN=filename, /STDOUT=filename, or /STDERR=filename could not be opened.
- The specified program could not be started.

The error is logged in SYSLOG. The operating system error number and message are available from the %SYSTEM.Process.OSError() method.

Auditing

An OS command audit record is added to the audit log for each ZF(-100) call. This record includes information such as the following:

Command: /Users/myname/cache/jlc/bin/clmanager 4002
Flags: /ASYNC/SHELL

ZF(-100), ZF(-1), and ZF(-2)

These three functions are in most respects identical. They differ in the following ways:

- ZF(-100) can be synchronous or asynchronous. It can execute using the operating system shell or not using the shell. It always sets ZCHILD. Both ZF(-1) and ZF(-2) with no specified parameters launch the operating system shell; ZF(-100) requires a program parameter (and the /SHELL flag) to launch the operating system shell. ZF(-100) is the preferred function for all purposes, replacing both ZF(-1) and ZF(-2).
- ZF(-1) executes using the OS shell. It is synchronous; it suspends execution of the current process while awaiting completion of the spawned child process. It receives status information from the spawned process, which it returns as an exit status code (an integer value) when the spawned process completes. ZF(-1) does not set ZCHILD.
- ZF(-2) executes using the OS shell. It is asynchronous; it does not suspend execution of the current process. It immediately returns a status value upon spawning the child process. Because it does not await completion of the spawned child process it cannot receive status information from that process. ZF(-2) sets ZCHILD if its fifth argument is true.

See Also

- ZF(-1) function
- ZF(-2) function
- ZCHILD special variable
- Issuing Operating System Commands in Using the Callout Gateway
$ZHEX

Converts a hexadecimal string to a decimal number and vice versa.

$ZHEX (num)
$ZH (num)

**Parameter**

| num   | An expression that evaluates to a numeric value be converted, either a quoted string or an integer (signed or unsigned). |

**Description**

$ZHEX converts a hexadecimal string to a decimal integer, or a decimal integer to a hexadecimal string.

If `num` is a string value, $ZHEX interprets it as the hexadecimal representation of a number, and returns that number in decimal. Be sure to place the string value within quotation marks.

If `num` is a numeric value, $ZHEX converts it to a string representation of the number in hexadecimal format. If either the initial or the final numeric value cannot be represented as an 8-byte signed integer, $ZHEX issues a <FUNCTION> error.

You can perform the same hexadecimal/decimal conversions using the **HexToDecimal()** and **DecimalToHex()** methods of the %SYSTEM.Util class:

```objectscript
  WRITE $SYSTEM.Util.DecimalToHex("27")
  WRITE $SYSTEM.Util.HexToDecimal("27"),!
  WRITE $SYSTEM.Util.HexToDecimal("1B")
```

$ZHEX can be used with $CHAR to specify a Unicode character using its hexadecimal character code:

```objectscript
  $CHAR($ZHEX("hexnum"))
```

**Parameter**

`num`

A string value or a numeric value, a variable that contains a string value or a numeric value, or an expression that evaluates to a string value or a numeric value.

A string value is read as a hexadecimal number and converted to a positive decimal integer. $ZHEX recognizes both uppercase and lowercase letters “A” through “F” as hexadecimal digits. It truncates leading zeros. It does not recognize plus and minus signs or decimal points. It stops evaluation of a string when it encounters a non-hexadecimal character. Therefore, the strings “F”, “f”, “00000F”, “F.7”, and “FRED” all evaluate to decimal 15. If the first character encountered in a string is not a hexadecimal character, $ZHEX evaluates the string as zero. Therefore, the strings “0”, “0.9”, “+F”, “-F”, and “H” all evaluate to zero. The null string (""") is an invalid value and issues a <FUNCTION> error.

An integer value is read as a decimal number and converted to hexadecimal. An integer can be positive or negative. $ZHEX recognizes leading plus and minus signs. It truncates leading zeros. It evaluates nested arithmetic operations. However, it does not recognize decimal points. It issues a <FUNCTION> error if it encounters a decimal point character. Therefore, the integers 217, 0000217, +217, +-217 all evaluate to hexadecimal D9. -217, -0000217, and +217 all evaluate to FFFFFFFFFFFFF27 (the twos complement). Other values, such as floating point numbers, trailing signs, and nonnumeric characters result in a <FUNCTION> or <SYNTAX> error.

**Examples**

```objectscript
  WRITE $ZHEX("F")
```
returns 15.

    WRITE $ZHEX(15)

returns F.

    WRITE $ZHEX("1AB8")

returns 6840.

    WRITE $ZHEX(6840)

returns 1AB8.

    WRITE $ZHEX("XXX")

returns 0.

    WRITE $ZHEX(-1)

returns FFFFFFFFEFFFFFEFFFF.

    WRITE $ZHEX((3+(107*2)))

returns D9.

**Notes**

**Forcing a Hexadecimal Interpretation**

To force an integer value to be interpreted as hexadecimal, concatenate any non-hexadecimal character to the end of your `num` parameter. For example:

    WRITE $ZHEX(16_"M")

returns 22.

**See Also**

- ZZDUMP command
- $ASCII function
- $CHAR function
$ZISWIDE

Checks whether a string contains any 16-bit wide characters.

$ZISWIDE(string)

Parameter

| string | A string of one or more characters, enclosed in quotation marks. |

Description

$ZISWIDE is a boolean function used to check whether a string contains any 16-bit wide character values. It returns one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>All characters have ASCII values 255 or less (8-bit characters). A null string (“”) also returns 0.</td>
</tr>
<tr>
<td>1</td>
<td>One or more characters have an ASCII value greater than 255 (wide characters).</td>
</tr>
</tbody>
</table>

Note that in a Unicode version of Caché, all characters are 16 bits in length. $ZISWIDE checks the character values to determine if they are in the ASCII range (0-255), and thus could be represented by 8 bits, or in the wide character range (256-65535) and thus use all 16 bits of the Unicode character.

Example

In the following example, the first two commands test strings that contain all narrow (8-bit) character values and return 0. The third command tests a string containing a wide character value (the second character), and therefore, returns 1:

```
WRITE $ZISWIDE("abcd"),","
WRITE $ZISWIDE($CHAR(71,83,77)),","
WRITE $ZISWIDE($CHAR(71,300,77))
```

Note that this example returns 0, 0, 1 only on Caché instances that were installed with Unicode support. Caché installed with 8-bit support returns 0, 0, 0.

See Also

- $ZPOSITION function
- $ZWASCII function
- $ZWCHAR function
- $ZWIDTH function
$ZLASCII

Converts a four-byte string to a number.

$ZLASCII(string,position)
$ZLA(string,position)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A string that can be specified as a value, a variable, or an expression. It must be a minimum of four bytes in length.</td>
</tr>
<tr>
<td>position</td>
<td>Optional — A starting position in the string. The default is 1.</td>
</tr>
</tbody>
</table>

Description

The value $ZLASCII returns depends on the parameters you use.

- $ZLASCII(string) returns a numeric interpretation of a four-byte string, starting with the first character position of string.
- $ZLASCII(string,position) returns a numeric interpretation of a four-byte string beginning at the starting position specified by position.

Upon successful completion, $ZLASCII always returns a positive integer. $ZLASCII returns -1 if string is of an invalid length, or position is an invalid value.

Notes

$ZLASCII and $ASCII

$ZLASCII is similar to $ASCII except that it operates on four byte (32-bit) words instead of single 8-bit bytes. For two byte (16-bit) words use $ZWASCII; for eight byte (64-bit) words, use $ZQASCII.

$ZLASCII(string,position) is the functional equivalent of:

$ASCII(string,position+3)*256 + $ASCII(string,position+2)*256 + $ASCII(string,position+1)*256 + $ASCII(string,position)

$ZLASCII and $ZLCHAR

The $ZLCHAR function is the logical inverse of the $ZLASCII function. For example:

```
SET x=$ZLASCII("abcd")
WRITE !,x
SET y=$ZLCHAR(x)
WRITE !,y
```

Given “abcd” $ZLASCII returns 1684234849. Given 1684234849 $ZLCHAR returns “abcd”.

See Also

- $ASCII function
- $ZLCHAR function
- $ZWASCII function
- $ZQASCII function
$ZLCHAR

Converts a number to a four-byte string.

$ZLCHAR(n)
$ZLC(n)

Parameter

<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>A positive integer in the range 0 through 4294967295. It can be specified as a value, a variable, or an expression.</td>
</tr>
</tbody>
</table>

Description

$ZLCHAR returns a four-byte (long) character string for $n. The bytes of the character string are presented in little-endian byte order, with the least significant byte first.

If $n is out of range or a negative number, $ZLCHAR returns the null string.

Notes

$ZLASCI and $ZLCHAR

The $ZLASCI function is the logical inverse of $ZLCHAR. For example:

```
SET x=$ZLASCI("abcd")
WRITE l,x
SET y=$ZLCHAR(x)
WRITE l,y
```

Given “abcd” $ZLASCI returns 1684234849. Given 1684234849 $ZLCHAR returns “abcd”.

$ZLCHAR and $CHAR

$ZLCHAR is similar to $CHAR, except that it operates on four byte (32-bit) words instead of single 8-bit bytes. For two byte (16-bit) words use $ZWASCII; for eight byte (64-bit) words, use $ZQASCII.

$ZLCHAR is the functional equivalent of the following form of $CHAR:

```
SET n=$ZLASCI("abcd")
WRITE l,n
WRITE l,$CHAR(n\256,n\256\256,n\(256**2)\#256,n\(256**3))
```

Given “abcd” $ZLASCI returns 1684234849. Given 1684234849, this $CHAR statement returns “abcd”.

See Also

- $ZLASCI function
- $CHAR function
- $ZWCHAR function
- $ZQCHAR function
$ZLN

Returns the natural logarithm of the specified number.

$ZLN(n)

Parameter

| n                  | Any positive nonzero number, which can be specified as a value, a variable, or an expression. |

Description

$ZLN returns the natural logarithm (base e) value of n.

Specifying zero or a negative number results in an <ILLEGAL VALUE> error.

The corresponding natural logarithm power function is $ZEXP.

Examples

The following example writes the natural log of the integers 1 through 10:

```objectscript
FOR x=1:1:10 {
    WRITE !,"The natural log of ",x," = ",$ZLN(x)
}QUIT
```

returns:

The natural log of 1 = 0
The natural log of 2 = .6931471805599453089
The natural log of 3 = 1.09861228886688109691
The natural log of 4 = 1.386294361119890618
The natural log of 5 = 1.6094379124434100375
The natural log of 6 = 1.791759469228055002
The natural log of 7 = 1.945910149055313306
The natural log of 8 = 2.079441541679835929
The natural log of 9 = 2.197224577336219384
The natural log of 10 = 2.302585092994045684

The following example shows the relationship between $ZLN and $ZEXP:

```objectscript
SET x=$ZEXP(1) ; x = 2.718281828459045236
WRITE $ZLN(x)
```

returns 1.

```objectscript
WRITE $ZLN(0)
```

issues an <ILLEGAL VALUE> error.

See Also

- $ZEXP function
- $ZLOG function
- $ZPI special variable
$ZLOG

Returns the base 10 logarithm value of the specified positive numeric expression.

$ZLOG(n)

Parameter

| n     | Any positive, nonzero number, which can be specified as a value, a variable, or an expression. |

Description

$ZLOG$ returns the base 10 logarithm value of $n$.

Specifying zero or a negative number results in an $<$ILLEGAL VALUE$>$ error.

The corresponding natural log (base e) function is $SZLN$.

Examples

The following example writes the base 10 logarithms of the integers 1 through 10:

```objectscript
FOR x=1:1:10 {
    WRITE !,"The log of ",x," = ",$ZLOG(x)
}
QUIT
returns:

The log of 1 = 0
The log of 2 = .30102995663981195
The log of 3 = .477121254719662437
The log of 4 = .60205999132796239
The log of 5 = .698970004336018805
The log of 6 = .778151250383436333
The log of 7 = .845098040034256831
The log of 8 = .903089986991943586
The log of 9 = .954242509439324875
The log of 10 = 1
```

```objectscript
WRITE $ZLOG($ZPI)
```

returns .4971498726941338541.

```objectscript
WRITE $ZLOG(.5)
```

returns -.30102995663981195.

```objectscript
WRITE $ZLOG(0)
```

issues an $<$ILLEGAL VALUE$>$ error.

See Also

- $SZEXP$ function
- $SZLN$ function
- $SZPI$ special variable
$ZNAME

Validates the specified name string as a legal identifier.

$ZNAME(string,type,lang)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The name to evaluate, specified as a quoted string.</td>
</tr>
<tr>
<td>type</td>
<td>Optional — An integer code specifying the type of name validation to perform. Valid values are 0 through 6. The default is 0.</td>
</tr>
<tr>
<td>lang</td>
<td>Optional — An integer code specifying the language mode to use when validating string. Valid values are 0 through 12. The default is to use the current language mode.</td>
</tr>
</tbody>
</table>

Description

$ZNAME returns 1 (true) if the string parameter is a legal identifier. Otherwise, $ZNAME returns 0 (false). The optional type parameter determines what type of name validation to perform on the string. If this parameter is omitted, the validation defaults to local variable naming conventions. The optional lang parameter specifies what language mode conventions to apply to the validation.

Your locale may not permit the use of an identifier that $ZNAME validates as a legal identifier. The valid identifier characters for your locale are defined in the National Language Support (NLS) Identifier locale setting; they are not user-modifiable. For further details on NLS, refer to the section on “System Classes for National Language Support” in Caché Specialized System Tools and Utilities.

$ZNAME only performs character validation; it does not perform string length validation for identifiers.

Parameters

string

A quoted string to validate as a legal identifier name. The characters a valid string can contain depend both on the type of identifier to validate (specified by type), the language mode (lang), and the definition of your locale. The string specifies only the identifier name; it should not include prefix characters, such as the caret (^) prefix and the optional delimited namespace name prefix for a global, or suffix characters, such as an array subscript or parameter parentheses. By default, the following are valid identifier characters in Caché:

- Uppercase letters: A through Z (CHAR(65) through CHAR(90))
- Lowercase letters: a through z (CHAR(97) through CHAR(122))
- Letters with accent marks: (CHAR(192) through CHAR(255), exclusive of CHAR(215) and CHAR(247))
- Unicode installations: Unicode letters in non-Latin character sets, such as Greek or Cyrillic letters. For example, CHAR(256) through CHAR(687) and CHAR(913) through CHAR(1153) exclusive of CHAR(930) and CHAR(1014).
- Digits: 0 through 9 (CHAR(48) through CHAR(57)) subject to positional restrictions for some identifiers
- The percent sign: % (CHAR(37)) subject to positional restrictions for some identifiers

$ZNAME also accepts as valid characters CHAR(170), CHAR(181), and CHAR(186).
Note: The Japanese locale does not support accented Latin letter characters in identifiers. Japanese identifiers may contain (in addition to Japanese characters) the Latin letter characters A-Z and a-z (65–90 and 97–122), and the Greek capital letter characters (913–929 and 931–937).

**type**

An integer code specifying the type of name validation to perform:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
<th>Restricted Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Validate a local variable name.</td>
<td>First character only: %</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subsequent characters only: digits 0–9</td>
</tr>
<tr>
<td>1</td>
<td>Validate a routine name.</td>
<td>First character only: %</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subsequent characters only: digits 0–9 and the period (.) character. A period</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cannot be the first or last character in a routine name.</td>
</tr>
<tr>
<td>2</td>
<td>Validate a label (tag) name.</td>
<td>First character only: %</td>
</tr>
<tr>
<td>3</td>
<td>Validate a global or process-private global  name.</td>
<td>First character only: %</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subsequent characters only: digits 0–9 and the period (.) character. A period</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cannot be the first or last character in a global name.</td>
</tr>
<tr>
<td>4</td>
<td>Validate a fully qualified class name.</td>
<td>First character only: %</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subsequent characters only: digits 0–9 and the period (.) character. A period</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cannot be the first or last character in a routine name.</td>
</tr>
<tr>
<td>5</td>
<td>Validate a method name.</td>
<td>First character only: %</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subsequent characters only: digits 0–9</td>
</tr>
<tr>
<td>6</td>
<td>Validate a property name</td>
<td>First character only: %</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subsequent characters only: digits 0–9</td>
</tr>
</tbody>
</table>

If \( \text{type} = 0 \) (or not specified), an identifier that passes validation may be used for a local variable name, or for any other type of ObjectScript name. This is the most restrictive form of validation. The first character of a valid identifier must be either a percent sign (%) or a valid letter character. The second and subsequent characters of a valid identifier must be either a valid letter character or a digit.

If \( \text{type} = 2 \), an identifier that passes validation may be used for a line label. This is the only type of identifier that allows a digit (0–9) as the first character. Specify only the label name; do not specify a colon prefix (used in triggers) or parameter parentheses following the label name.

If \( \text{type} = 3 \), an identifier that passes validation may be used for global and process-private global names. However, global and process-private global names cannot include wide characters; \$ZNAME considers wide-character letters to be valid.
identifier characters for all name validation types. Therefore, if type=3, an identifier containing wide character letters passes $ZNAME validation, but generates a <WIDE CHAR> error when used as a global name or process-private global name.

If type = 4, an identifier that passes validation may be used for a class name. A class name can contain periods, with the following restrictions: a period may not be immediately followed by a number character or by another period. These restrictions on the use of periods do not apply to type=1 and type=3 validation. No valid identifier of any type may have a period as the first or last character of string.

lang

An integer code specifying the language mode to use for validation. Caché applies the conventions of the specified language mode to the validation without changing the current language mode. (For a list of available current language modes, see the LanguageMode() method of the %SYSTEM.Process class.) The default is for $ZNAME to use the language mode conventions of the current language mode. Because all Caché language modes use the same naming conventions, lang can be omitted and take the default, except to specify lang=11.

You use lang=11 to validate Caché MVBasic local variables. If you specify $ZNAME(string,0,11), $ZNAME follows the MultiValue Basic naming conventions for local variables, which differ from the ObjectScript local variable naming conventions. For further details, see User Variables in the Caché MultiValue Basic Reference.

Examples

The following example shows the $ZNAME function validating the expressions as true (1). Note that the last two examples contain periods, which are permitted in routine names (type=1) and global names (type=3):

```objectscript
WRITE !,$ZNAME("A")
WRITE !,$ZNAME("A1")
WRITE !,$ZNAME("%A1",0)
WRITE !,$ZNAME("%A1",1)
WRITE !,$ZNAME("A.1",1)
WRITE !,$ZNAME("A.1",3)
```

In the following example, the first $ZNAME fails validation (returns 0) because (by default) it validates for a local variable name, and the first character of a local variable name cannot be a digit. The second $ZNAME passes validation (returns 1) because type=2 specifies label validation, and first character of a label name can be a digit.

```objectscript
WRITE "local var: ",$ZNAME("1A")!,
WRITE "label: ",$ZNAME("1A",2)
```

The following example fails validation for all type values. Caché names of all types cannot contain a percent sign unless it is the first character of the name:

```objectscript
FOR i=0:1:6 {
  WRITE "type ",i," is ",$ZNAME("A\%1",i),!
}
```

The following example shows the full set of valid 8-bit identifier characters for local variable names. These valid identifier characters include the letter characters ASCII 192 through ASCII 255, with the exceptions of ASCII 215 and ASCII 247, which are arithmetic symbols:

```objectscript
FOR n=1:1:255  {
  IF $ZNAME("A"_SCHAR(n),0) & $ZNAME($SCHAR(n),0) {
    WRITE !,$ZNAME($SCHAR(n),0)," ASCII code="n," Char.="$SCHAR(n) }
  ELSEIF $ZNAME($SCHAR(n),0) {
    WRITE !,$ZNAME($SCHAR(n),0)," ASCII code="n," 1st Char.="$SCHAR(n) }
  ELSEIF $ZNAME("A"_SCHAR(n),0) {
    WRITE !,$ZNAME("A"_SCHAR(n),0)," ASCII code="n," Subseq. Char.="$SCHAR(n) }
  ELSE { }
}
WRITE !,"All done"
```

The following example passes validation on a Unicode installation of Caché. The Greek letters specified are valid Unicode letters and thus pass $ZNAME validation. However, this name cannot be used for a global or process-private global (type=3), and may not be usable with some locales (such as the Japanese locale):


ObjectScript Functions

IF $SYSTEM.Version.IsUnicode() {
  WRITE $C(913)_$C(961)_$C(947)_$C(959),!
  FOR i=0:1:6 {
    WRITE "type ",i," is ",$ZNAME($C(913)_$C(961)_$C(947)_$C(959)),!
  }
} ELSE {WRITE "This example requires a Unicode installation of Caché"}

SQL Identifiers

SQL identifiers may include punctuation characters (underscore (_), at sign (@), pound sign (#), and dollar sign ($)) that are not valid characters in ObjectScript identifiers. SQL routine names may not include the percent sign (%) at any location other than the first character. For further details, see Identifiers in the Using Caché SQL.

See Also

- ObjectScript Symbols table
- Cache SQL Symbols table
$ZPOSITION

Returns the number of characters in an expression that can fit within a specified field width.

$ZPOSITION(expression,field,pitch)

Parameters

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A string expression.</td>
</tr>
<tr>
<td>field</td>
<td>An integer expression that specifies field width.</td>
</tr>
<tr>
<td>pitch</td>
<td>Optional — A numeric expression that specifies the pitch value to use for full-width characters. The default is 2. Other permissible values are 1, 1.25, and 1.5.</td>
</tr>
</tbody>
</table>

Description

$ZPOSITION returns the number of characters in expression that can fit within the field value. The pitch value determines the width to use for full-width characters. All other characters receive a default width of 1 and are considered to be half-width. Because half-width characters count as 1, field also expresses the number of half-width characters that fit in field.

$ZPOSITION adds the widths of the characters in the expression one at a time until the cumulative width equals the value of field or until there are no more characters in expression. The result is thus the number of characters that will fit within the specified field value including any fractional part of a character that would not completely fit.

Note: $ZPOSITION can be abbreviated as $ZP in DSM-J mode. This abbreviation cannot be used in Caché mode.

Examples

In the following example, assume that the variable string contains two half-width characters followed by a full-width character.

```
WRITE $ZPOSITION(string,3,1.5)
```

returns 2.666666666666666667.

In the above example, the first two characters in string fit in the specified field width with one left over. The third character in string, a full-width character with a width of 1.5 (determined by the pitch argument), would not completely fit, although two thirds (1/1.5) of the character would fit. The fractional part of the result indicates that fact.

In the following example, string is now a string that contains a full-width character followed by two half-width characters. The result returned is 2.5:

```
WRITE $ZPOSITION(string,3,1.5)
```

The results are now different. This is because the first two characters, which have a combined width of 2.5, would completely fit with .5 left over. Even so, only half of the third character (.5/1) would fit.

Finally, if string is a string that contains three half-width characters then all three characters would completely (and exactly) fit, and the result would be 3:

```
WRITE $ZPOSITION(string,3,1.5)
```

Note: Full-width characters are determined by examining the pattern-match table loaded for your Caché process. Any character with the full-width attribute is considered to be a full-width character. The special ZFWCHARZ patcode can be used to check for this attribute (for example, char?1ZFWCHARZ). For more information about the full-width attribute, see the description of the $X/$Y Tab in the section on “System Classes for National Language Support” in Caché Specialized System Tools and Utilities.
See Also

- \$ZWIDTH function
- \$ZZENKAKU function
$ZPOWER

Returns the value of a number raised to a specified power.

$ZPOWER(num,exponent)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>The number to be raised to a power.</td>
</tr>
<tr>
<td>exponent</td>
<td>The exponent.</td>
</tr>
</tbody>
</table>

**Description**

$ZPOWER returns the value of the num parameter raised to the nth power.

This function performs the same operation as the Exponentiation operator (**). For details on valid parameter values and the value returned for specific combinations of parameter values, see Exponentiation Operator in the “Operators and Expressions” chapter of Using Caché ObjectScript.

**Parameters**

**num**

The number to be raised to a power. It can be integer or floating point, negative, positive, or zero. It can be a standard Caché number or an IEEE double-precision binary floating-point number (a $DOUBLE number). It can be specified as a value, a variable, or an expression.

If you specify num as a quoted string, the string is parsed as a number, as described in “Strings as Numbers” in the “Data Types and Values” chapter of Using Caché ObjectScript. The null string (""") and nonnumeric strings evaluate to zero.

**exponent**

The exponent is a number that can be integer or floating point, negative, positive, or zero. It can be a standard Caché number or an IEEE double-precision binary floating-point number (a $DOUBLE number). It can be specified as a numeric or string value, a variable, or an expression.

If you specify exponent as a quoted string, the string is parsed as a number, as described in “Strings as Numbers” in the “Data Types and Values” chapter of Using Caché ObjectScript. The null string (""") and nonnumeric strings evaluate to zero.

The following combinations of num and exponent result in an error:

- If num is negative, exponent must be an integer. Otherwise an <ILLEGAL VALUE> error is generated.
- If num is 0, exponent must be a positive number or zero. Otherwise an <ILLEGAL VALUE> or <DIVIDE> error is generated.
- Large exponent values, such as $ZPOWER(9, 153) may result in an overflow, generating a <MAXNUMBER> error, or may result in an underflow, returning 0. Which result occurs depends on whether num is greater than 1 (or -1), and whether exponent is positive or negative. A <MAXNUMBER> error occurs when an operation exceeds the largest number that Caché supports. For further details, refer to “Extremely Large Numbers” in the “Data Types and Values” chapter of Using Caché ObjectScript.

For further details on valid parameter values and the value returned for specific combinations of parameter values, see Exponentiation Operator in the “Operators and Expressions” chapter of Using Caché ObjectScript.
Examples
The following example raises 2 to the first ten powers:

```
SET x=0
WHILE x < 10 {
    SET rtn=$ZPOWER(2,x)
    WRITE !,"The ",x," power of 2=",rtn
    SET x=x+1 }
```

See Also
- $ZSQR function
- $ZEXP function
- $ZLN function
- $ZLOG function
- Operators in Using Caché ObjectScript
$ZQASCII

Converts an eight-byte string to a number.

$ZQASCII(string,position)
$ZQA(string,position)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A string. It can be a value, a variable, or an expression. It must be a minimum of eight bytes in length.</td>
</tr>
<tr>
<td>position</td>
<td>Optional — A starting position in the string, expressed as a positive integer. The default is 1. Position is counted in single bytes, <em>not</em> eight-byte strings. The position cannot be the last byte in the string, or beyond the end of the string. A numeric position value is parsed as an integer by truncating decimal digits, removing leading zeros and plus signs, etc.</td>
</tr>
</tbody>
</table>

**Description**

The value that $ZQASCII returns depends on the parameters you use.

- $ZQASCII(string) returns a numeric interpretation of an eight-byte string starting at the first character position of string.
- $ZQASCII(string,position) returns a numeric interpretation of an eight-byte string beginning at the starting byte position specified by position.

$ZQASCII can return either a positive or a negative integer.

$ZQASCII issues a <FUNCTION> error if string is of an invalid length, or position is an invalid value.

**Example**

The following example determines the numeric interpretation of the character string "abcdefgh":

```
WRITE $ZQASCII("abcdefgh")
```

It returns 7523094288207667809.

The following examples also return 7523094288207667809:

```
WRITE !,$ZQASCII("abcdefgh",1)
WRITE !,$ZQASCII("abcdefghxx",1)
WRITE !,$ZQASCII("xxabcdefghxx",3)
```

**Notes**

**$ZQASCII and $ASCII**

$ZQASCII is similar to $ASCII except that it operates on eight byte (64-bit) words instead of single 8-bit bytes. For 16-bit words use $ZWASCII; for 32-bit words, use $ZLASCII.

**$ZQASCII and $ZQCHAR**

The $ZQCHAR function is the logical inverse of $ZQASCII. For example:

```
WRITE $ZQASCII("abcdefgh")
```
returns: 7523094288207667809.

    WRITE $ZQCHAR(7523094288207667809)

returns “abcdefgh”.

**See Also**

- `$ASCII` function
- `$ZQCHAR` function
$ZQCHAR

Converts a number to an eight-byte string.

| $ZQCHAR(n)  |
| $ZQC(n)    |

Parameter

| n | An integer in the range -9223372036854775808 through 9223372036854775807. It can be specified as a value, a variable, or an expression. |

Description

$ZQCHAR returns an eight-byte (quad) character string corresponding to the binary representation of n. The bytes of the character string are presented in little-endian byte order, with the least significant byte first.

$ZQCHAR issues a <FUNCTION> error if n is invalid.

Example

The following example returns the eight-byte string for the integer 7523094288207667809:

```
WRITE $ZQCHAR(7523094288207667809)
```

returns: “abcdefgh”

Notes

$ZQCHAR and $CHAR

$ZQCHAR is similar to $CHAR except that it operates on eight byte (64-bit) words instead of single 8-bit bytes. For 16-bit words use $ZWCHAR; for 32-bit words, use $ZLCHAR.

$ZQCHAR and $ZQASCII

$ZQASCII is the logical inverse of the $ZQCHAR function. For example:

```
WRITE $ZQCHAR(7523094288207667809)
```

returns: abcdefgh

```
WRITE $ZQASCII("abcdefgh")
```

returns: 7523094288207667809

See Also

- $ZQASCII function
- $CHAR function
- $ZLCHAR function
- $ZWCHAR function
$ZSEARCH

Returns the full file specification, pathname and filename, of a specified file.

$ZSEARCH(target)  $ZSE(target)

Parameter

| target | A filename, a pathname, or a null string. May contain one or more * or ? wildcard characters. |

Description

$ZSEARCH returns the full file specification (pathname and filename) of a specified target file or directory. The filename may contain wild cards so that $ZSEARCH can return a series of fully qualified pathnames that satisfy the wild carding.

Note: Some operating systems use the slash (/) character as the directory path delimiter. Other operating systems use the backslash (\) character. In this Description, the word "slash" means either slash or backslash, as appropriate.

If the target parameter does not specify a pathname, $ZSEARCH searches the current working directory. $ZSEARCH applies the rules in its matching process in the following order:

1. $ZSEARCH scans the target to see if it is surrounded with percent characters (%). If $ZSEARCH finds such text, it treats the string as an environment variable. $ZSEARCH performs name translation on the string.
2. $ZSEARCH scans the string that results from the previous step to find the final slash. If $ZSEARCH finds a final slash, it uses the string up to, but not including, the final slash as the path or directory to be searched. If $ZSEARCH does not find a final slash, it searches the current working directory, which is determined by the current namespace.
3. If $ZSEARCH found a final slash in the previous step, it uses the portion of the target string following the final slash as the filename search pattern. If $ZSEARCH did not find a final slash in the previous step, it uses the whole string that results from Step 1 as the filename search pattern.

The filename search pattern can be any legal filename string or a filename wildcard expression. The first filename that matches the search pattern is returned as the $ZSEARCH function value. Which is the first matching file is platform-dependent (as described in the Notes section).

If the next invocation of $ZSEARCH specifies the null string as the target, $ZSEARCH continues with the previous target and returns the next filename that matches the search pattern. When there are no more files that match the search pattern, $ZSEARCH returns a null string.

The NormalizeDirectory() method of the %Library.File class can also be used to return the full pathname of a specified file or directory, as shown in the following example:

```
NEW $NAMESPACE
SET $NAMESPACE="SAMPLES"
WRITE ##class(%Library.File).NormalizeDirectory("CACHE.DAT"),!
NEW $NAMESPACE
SET $NAMESPACE="USER"
WRITE ##class(%Library.File).NormalizeDirectory("CACHE.DAT")
```

However, NormalizeDirectory() cannot use wildcards.

**Wildcards**

$ZSEARCH allows the use of the following wildcard expressions within the quoted target string.
Wildcard | Match
--- | ---
* | Matches any string of zero or more characters.
? | Matches a single character. On Windows systems matches one or zero characters at the end of a name element.

These wildcards follow the host platform’s usage rules. On Windows, $ZSEARCH performs a case-independent search, then returns the actual case of the located file or directory. For example, “j*” can match “JOURNAL”, “journal”, or “Journal”; the actual directory name is “Journal”, which is what is returned.

On Windows and UNIX® systems you can also use the following standard pathname symbols: a single dot (.) to specify the current directory, or a double dot (..) to specify its parent directory. These symbols can be used in combination with wildcard characters.

**Parameters**

**target**

The following are the available types of values for the target parameter:

<table>
<thead>
<tr>
<th>Target Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pathname</td>
<td>An expression that evaluates to a string specifying the path to the file or group of files you want to list.</td>
</tr>
<tr>
<td>filename</td>
<td>A filename. The default location is the current dataset.</td>
</tr>
<tr>
<td>null string (&quot;&quot;&quot;)</td>
<td>Returns the next matching file name from the previous $ZSEARCH.</td>
</tr>
</tbody>
</table>

**Examples**

The following Windows examples find all files ending with “.DAT” as a file extension in the SAMPLES namespace.

```
NEW $NAMESPACE
SET $NAMESPACE="Samples"
SET file=$ZSEARCH("*.DAT")
WHILE file'="" {
    WRITE !,file
    SET file=$ZSEARCH(""")
}
WRITE !,"That is all the matching files"
QUIT
```

returns:

```
C:\InterSystems\Cache\mgr\samples\CACHE.DAT
```

The following Windows example finds all files beginning with the letter “c” in the SAMPLES namespace.

```
NEW $NAMESPACE
SET $NAMESPACE="Samples"
SET file=$ZSEARCH("c**")
WHILE file'="" {
    WRITE !,file
    SET file=$ZSEARCH(""")
}
WRITE !,"That is all the matching files"
QUIT
```

returns:

```
C:\InterSystems\Cache\mgr\samples\CACHE.DAT
C:\InterSystems\Cache\mgr\samples\cache.lck
```
Notes

Directory Locking

In order to give accurate results, the process keeps the directory open until $ZSEARCH has returned all files in the directory (that is, until $ZSEARCH returns a null string, or a new $ZSEARCH is started). This may prevent other operations, such as deleting the directory. When you start a $ZSEARCH you should always repeat the $ZSEARCH("") until it returns a null string. An alternative, if you do not want to retrieve all files, is to issue $ZSEARCH with a filename that you know does not exist, such as $ZSEARCH(-1).

Windows Support

For Windows, the target parameter is a standard file specification, which may contain wildcard characters (* and ?).

- The * wildcard can be used to match a dot, but the ? wildcard cannot. Therefore, “MYFILE*” matches MYFILE-FOLDER, MYFILE.DOC, and MYFILEBACKUP.DOC; “MYFILE?DOC” does not match MYFILE.DOC.
- The ? wildcard does not match zero characters within a name element. Therefore, “MY?FILE.DOC” matches MY2FILE.DOC, but does not match MYFILE.DOC.
- The ? wildcard matches zero characters at the end of a name element. Extra trailing ? wildcards are ignored. Therefore, “MYFILE?.DOC” matches both MYFILE2.DOC and MYFILE.DOC.

If you do not specify a directory, the current working directory is used. $ZSEARCH returns the first matching entry in the directory in alphabetical order. It returns the full file specification or fully qualified pathname. The drive letter is always returned as an uppercase letter, regardless of how it was specified.

By default, Windows checks only the first three characters of a filename extension suffix. Therefore, $ZSEARCH("*.doc") would return not only all files with the .doc suffix, but also all files with the .docx suffix. If you wish to limit your search to only .docx files, you must specify the four character suffix: $ZSEARCH("*.docx"). You cannot use trailing ? wildcards to limit your search to suffixes longer than three characters.

UNIX® Support

For UNIX®, the target parameter is a standard UNIX® file specification, which may contain wildcard characters (* and ?). If you do not specify a directory, the current working directory is used.

For UNIX®, $ZSEARCH returns the first active entry in the directory. Since UNIX® does not keep the directory entries in alphabetical order, the returned values are not in alphabetical order. Unlike Windows platforms, the $ZSEARCH function does not return the full file specification or fully qualified pathnames, unless the current working directory is used.
$ZSEC

Returns the trigonometric secant of the specified angle value.

$ZSEC(n)

Parameter

| n   | Angle in radians ranging from 0 to 2 Pi. It can be specified as a value, a variable, or an expression. |

Description

$ZSEC returns the trigonometric secant of \( n \). The result is a signed decimal number. The secant of 0 is 1. The secant of \( \pi \) is -1.

Note: Caché uses the host operating system’s routines to calculate trigonometric functions. For this reason, results obtained from different operating systems may not precisely match.

Parameter

\( n \)

An angle in radians ranging from \( \pi \) to 2 \( \pi \) (inclusive). It can be specified as a value, a variable, or an expression. You can specify the value \( \pi \) by using the \$ZPI special variable. You can specify positive or negative values smaller than \( \pi \) or larger than 2 \( \pi \); Caché resolve these values to the corresponding multiple of \( \pi \). For example, 3 \( \pi \) is equivalent to \( \pi \), and negative \( \pi \) is equivalent to \( \pi \).

A non-numeric string is evaluated as 0, and therefore $ZSEC returns 1.

Example

The following example permits you to compute the secant of a number:

```objectscript
READ "Input a number: ", num
IF $ZABS(num)>(2*$ZPI) { WRITE !,"number is a larger than 2 pi" }
ELSE {
    WRITE !,"the secant is: ",$ZSEC(num)
}
QUIT
```

See Also

- $ZCSC function
- $ZPI special variable
$ZSEEK

Establishes a new offset into the current sequential file.

$ZSEEK(offset,mode)

Parameters

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset</td>
<td>The offset into the current file.</td>
</tr>
<tr>
<td>mode</td>
<td>Optional — An integer value that determines the relative position of the offset. The default is 0.</td>
</tr>
</tbody>
</table>

Description

$ZSEEK establishes a new offset into the current device. The current device must be a sequential file. If the current device is not a sequential file, $ZSEEK issues a <FUNCTION> error.

The mode parameter determines the point from which offset is based. Called without parameters, $ZSEEK returns the current position in the file.

If there is no specifically set current device, $ZSEEK assumes that the device is the principal device.

Parameters

offset

The offset (in characters) from the point established by mode.

mode

The valid values are:

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Offset is relative to the beginning of the file (absolute).</td>
</tr>
<tr>
<td>1</td>
<td>Offset is relative to the current position.</td>
</tr>
<tr>
<td>2</td>
<td>Offset is relative to the end of the file.</td>
</tr>
</tbody>
</table>

If you do not specify a mode value, $ZSEEK assumes a mode value of 0.

See Also

- OPEN command
- CLOSE command
- I/O Devices and Commands in Caché I/O Device Guide
- Sequential File I/O in Caché I/O Device Guide
$ZSIN

Returns the trigonometric sine of the specified angle value.

$ZSIN(n)

Parameter

| n  | Angle in radians ranging from Pi to 2 Pi (inclusive). Other supplied numeric values are converted to a value within this range. |

Description

$ZSIN returns the trigonometric sine of $n. The result is a signed decimal number ranging from 1 to -1 (see note). $ZSIN(0) returns 0. $ZSIN($ZPI/2) returns 1.

Note: $ZSIN (like all trigonometric functions) calculates its values based on pi rounded to the number of available decimal digits. Therefore, the value returned by $ZSIN($ZPI) is .00000000000000462644 and $ZSIN($ZPI*2) is -.0000000000000092529. For this reason you should not perform limit tests comparing these returned values to 0.

Parameter

n

An angle in radians ranging from Pi to 2 Pi (inclusive). It can be specified as a value, a variable, or an expression. You can specify the value Pi by using the $ZPI special variable. You can specify positive or negative values smaller than Pi or larger than 2 Pi; Caché resolve these values to the corresponding multiple of Pi. For example, 3 Pi is equivalent to Pi, and negative Pi is equivalent to Pi.

A non-numeric string is evaluated as 0.

Examples

The following example permits you to compute the sine of a number:

```objectscript
READ "Input a number: ", num
IF $ZABS(num)>(2*$ZPI) { WRITE !,"number is larger than 2 pi" }
ELSE {
    WRITE !,"the sine is: ", $ZSIN(num)
}
QUIT
```

The following example compares the results from Caché fractional numbers ($DECIMAL numbers) and $DOUBLE numbers. In both cases, the sine of pi is a fractional number (not 0), but the sine of pi/2 is set to exactly 1:

```objectscript
WRITE !,"the sine is: ", $ZSIN($ZPI)
WRITE !,"the sine is: ", $ZSIN($DOUBLE($ZPI))
WRITE !,"the sine is: ", $ZSIN($ZPI/2)
WRITE !,"the sine is: ", $ZSIN($DOUBLE($ZPI)/2)
```

In the following example, all $ZSIN functions return zero (0):

```objectscript
WRITE !,"the sine is: ", $ZSIN(0.0)
WRITE !,"the sine is: ", $ZSIN(-0.0)
WRITE !,"the sine is: ", $ZSIN($DECIMAL(0.0))
WRITE !,"the sine is: ", $ZSIN($DOUBLE(0.0))
WRITE !,"the sine is: ", $ZSIN($DECIMAL(-0.0))
WRITE !,"the sine is: ", $ZSIN($DOUBLE(-0.0))
WRITE !,"the sine is: ", $ZSIN($DECIMAL(-0.0))
WRITE !,"the sine is: ", $ZSIN($DOUBLE(-0.0))
```
This is true on all platforms, including AIX.

**See Also**

- $ZCOS function
- $ZARCSIN function
- $ZPI special variable
$ZSQR

Returns the square root of a specified number.

$ZSQR (n)

**Parameter**

| n     | Any positive number, or zero. (The null string and nonnumeric string values are treated as a zero.) Can be specified as a value, a variable, or an expression. |

**Description**

$ZSQR$ returns the square root of $n$. It returns the square root of 1 as 1. It returns the square root of 0 and the square root of a null string ("") as 0. Specifying a negative number invokes an `<ILLEGAL VALUE>` error. You can use the absolute value function $ZABS$ to convert negative numbers to positive numbers.

**Examples**

The following example returns the square root of a user-supplied number.

```objectscript
READ "Input number for square root: ", num
IF num<0 { WRITE "ILLEGAL VALUE: no negative numbers" }
ELSE { WRITE $ZSQR(num) }
QUIT
```

Here are some specific examples:

```objectscript
WRITE $ZSQR(2)
```

returns 1.414213562373095049.

```objectscript
WRITE $ZSQR($ZPI)
```

returns 1.772453850905516027.

**See Also**

- $ZABS$ function
- $ZPOWER$ function
$ZSTRIP

Removes types of characters and individual characters from a specified string.

$ZSTRIP(string, action, remchar, keepchar)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string to be stripped.</td>
</tr>
<tr>
<td>action</td>
<td>What to strip from string. An action consists of an action code followed by a one or more mask codes. The mask code is optional when specifying remchar. An action is specified as a quoted string.</td>
</tr>
<tr>
<td>remchar</td>
<td>Optional — A string of specific character values to remove. If action does not contain a mask code, remchar lists the characters to remove. If action contains a mask code, remchar lists additional characters to remove that are not covered by the action parameter’s mask code.</td>
</tr>
<tr>
<td>keepchar</td>
<td>Optional — A string of specific character values to not remove that are designated for removal by the action parameter’s mask code. A mask code must be specified to specify keepchar.</td>
</tr>
</tbody>
</table>

Description

The $ZSTRIP function removes types of characters and/or individual character values from the specified string. In the action parameter you specify an action code indicating the kind of remove operation to perform, and (optionally) a mask code specifying the types of characters to remove. You can specify individual character values to remove using the remchar parameter. $ZSTRIP can remove both types of characters (such as all lowercase letters) and listed character values (such as the letters “AEIOU”) in the same operation. You can use the optional remchar and keepchar parameters to modify the effects of the action parameter’s mask code by specifying individual character values to remove or to keep.

For further information, refer to the Pattern Matching Operators section of Using Caché ObjectScript. You can also select types of characters, character sequences, and ranges of characters using the Regular Expression functions $LOCATE and SMATCH.

Parameters

action

A string indicating what characters to strip, specified as an action code, optionally followed by one or more mask codes.
### Action Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Strip all characters that match the mask code(s).</td>
</tr>
<tr>
<td>&lt;</td>
<td>Strip leading characters that match the mask code(s).</td>
</tr>
<tr>
<td>&gt;</td>
<td>Strip trailing characters that match the mask code(s).</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Strip leading and trailing characters that match the mask code(s).</td>
</tr>
<tr>
<td>=</td>
<td>Strip repeating characters that match the mask code(s). When encountering a series of repeated characters, this code strips the duplicate characters leaving a single instance. This code only strips duplicate adjacent characters. Thus stripping “a” from “aaaaaabc” yields “abc”, but stripping “a” from “abaca” returns the string “abaca” unchanged. The duplicate character test is case-sensitive.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Strip leading, trailing, and repeating characters that match the mask code(s).</td>
</tr>
</tbody>
</table>

An action code can consist of the * character, or any combination of a single <, >, or = character.

To strip types of characters, the **action** string should consist of an action code followed by one or more mask codes. To strip specific character values, omit the mask code, and specify a **remchar** value. You can specify both a mask code and a **remchar** value. If you specify neither a mask code nor a **remchar** value, **$ZSTRIP** returns string.

### Mask Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Strip everything.</td>
</tr>
<tr>
<td>A</td>
<td>Strip all alphabetic characters.</td>
</tr>
<tr>
<td>P</td>
<td>Strip punctuation characters, including blank spaces.</td>
</tr>
<tr>
<td>C</td>
<td>Strip control characters (0-31, 127-159).</td>
</tr>
<tr>
<td>N</td>
<td>Strip numeric characters. Note that a numeric string is not converted to canonical form before applying <strong>$ZSTRIP</strong>.</td>
</tr>
<tr>
<td>L</td>
<td>Strip lowercase alphabetic characters.</td>
</tr>
<tr>
<td>U</td>
<td>Strip uppercase alphabetic characters.</td>
</tr>
<tr>
<td>W</td>
<td>Strip whitespaces ($C(9)$, $C(32)$, $C(160)$).</td>
</tr>
</tbody>
</table>

Mask codes can be specified as uppercase or lowercase characters.

A mask code character can be preceded by a Unary Not (’) meaning do not remove characters of this type. You must specify at least one mask code without a Unary Not before specifying a Unary Not mask code. All mask codes without a Unary Not must precede the mask codes with a Unary Not.

**remchar**

Specific characters to remove, specified as a quoted string. These **remchar** characters can be specified in any order and duplicates are permitted.

If you do not specify a mask code, **$ZSTRIP** applies the **action** parameter to the **remchar** character(s). If you specify a mask code, **remchar** specifies one or more additional characters to remove. For example, if you specified in the **action** parameter that you want to remove all numeric characters (“*N”), but you also want to remove the letter “E” (used to represent scientific notation), you would add the string “E” as the **remchar** parameter, as shown in the second **$ZSTRIP**: 

```
SET str="first:123 second:12E3"
WRITE $ZSTRIP(str,"*N"),!
WRITE $ZSTRIP(str,"*N","E")
```
**keepchar**

Specific characters not to remove. For example, if you specified that you wanted to remove all white spaces and alphabetic characters (*WA), but preserve uppercase M, you would add the string “M” as the keepchar parameter.

**Examples**

The following example strips out all numeric characters. Because a numeric string is not converted to canonical form, the characters + and E are not stripped out:

```objectscript
SET str="+123E4"
WRITE $ZSTRIP(str,"*N")
```

returns: +E

In the following example, the first $ZSTRIP strips all punctuation characters, the second $ZSTRIP strips all punctuation characters except whitespace characters.

```objectscript
SET str="ABC#$%^ DEF& *GHI****
WRITE $ZSTRIP(str,"*P"),!
WRITE $ZSTRIP(str,"*P'W")
```

The following example strips out all characters, except lowercase letters (L). However, the example uses the remchar parameter to strip the lowercase x while preserving all other lowercase characters:

```objectscript
SET str="xXx-Aa BXXbx Cxc Dd xxEeX`XXx"
WRITE $ZSTRIP(str,"*E'L","x")
```

returns: abcdex

The following example strips out all characters, except lowercase letters (L). In this case, the example does not specify a remchar parameter value (but does specify the delimiting commas), but does specify the keepchar parameter to preserve uppercase A, B, and C:

```objectscript
SET str="X-Aa BXXb456X CxC Dd XXeX`XFFFFfXX"
WRITE $ZSTRIP(str,"*E'L","ABC")
```

returns: AaBbCcdef

The following example does not specify a mask code; it specifies to remove the letters “X” and “x” wherever they occur in the string. All other characters in the string are returned.

```objectscript
SET str="xXx-Aa BXXbx Cxc Dd xxEeX`XXx"
WRITE $ZSTRIP(str,"*P","")
```

returns: + $1,234.56

The following example does not specify a mask code; it specifies to remove the character “x” as a leading or trailing character, and to removed repeating “x”s wherever they occurs within the string:

```objectscript
SET str="*x $1x,x23XX4XX,X56XXxxxxX"
WRITE $ZSTRIP(str,"*","Xx")
```

returns: 00x0111x01lx0000

The following example strips out all numeric, alphabetic, and punctuation characters, except whitespace and lowercase letters. Note that all mask codes without a Unary Not must precede any mask codes with a Unary Not:

```objectscript
SET str="Aa66"*A B&$b Cc987 #Dd Ee"
WRITE $ZSTRIP(str,"*HAP'W'L")
```

returns: a b c d e

The following example strips out leading, trailing, and repeating characters that match the mask code A (all alphabetic characters):
It returns 123DEFfG5555567HI; $ZSTRIP stripped leading characters (ABC) until it encountered a character of a type not included in the mask (1), and stripped trailing characters from the end of the string until it encountered a non-mask character (the blank space). Repeated characters of the mask type were reduced to a single occurrence (DDDEE = DE); note that the repeat test is case-sensitive (FFfff = Ff). Repeated characters that are not of the mask type (55555) are unaffected.

The following example strips out all characters except the hexadecimal digits 0–9 and A–F:

```
SET str="123$ GYJF870B-QD @#$+"
WRITE $ZSTRIP (str,"*E'N",,"ABCDEF")
```

returns: 123F870BD

**See Also**

- $EXTRACT function
- $ZCONVERT function
- $LOCATE and $MATCH functions for Regular Expressions
- Pattern Matching operators in *Using Caché ObjectScript*
$ZTAN

Returns the trigonometric tangent of the specified angle value.

$ZTAN(n)

**Parameter**

| n | An angle in radians ranging from Pi to 2 Pi (inclusive). Other supplied numeric values are converted to a value within this range. |

**Description**

$ZTAN$ returns the trigonometric tangent of $n$. The result is a signed decimal number.

**Note:** $ZTAN$ (like all trigonometric functions) calculates its values based on pi rounded to the number of available decimal digits. Therefore, the value returned by $ZTAN($ZPI$)$ is $-0.000000000000000462644$ and $ZTAN(-$ZPI$)$ is $0.000000000000000462644$. For this reason you should not perform limit tests comparing these returned values to 0. $ZTAN(0)$ is 0.

**Parameter**

$n$

An angle in radians ranging from 0 to 2 Pi. It can be specified as a value, a variable, or an expression.

A non-numeric string is evaluated as 0.

**Examples**

The following example permits you to compute the tangent of a number:

```objectscript
READ "Input a number: ", num
WRITE !,"the tangent is: ", $ZTAN(num)
QUIT
```

The following example compares the results from Caché fractional numbers (**$DECIMAL$** numbers) and **$DOUBLE$** numbers. In both cases, the tangent of 0 is exactly 0, but the tangent of pi is a negative fractional number (not exactly 0):

```objectscript
WRITE !,"the tangent is: ", $ZTAN(0.0)
WRITE !,"the tangent is: ", $ZTAN($DOUBLE(0.0))
WRITE !,"the tangent is: ", $ZTAN($ZPI)
WRITE !,"the tangent is: ", $ZTAN($DOUBLE($ZPI))
WRITE !,"the tangent is: ", $ZTAN(1.0)
WRITE !,"the tangent is: ", $ZTAN($DOUBLE(1.0))
```

**See Also**

- **$ZARCTAN$** function
- **$ZSIN$** function
- **$ZPI$** special variable
$ZTIME

Validates a time and converts it from internal format to the specified display format.

$ZTIME(htime, tformat, precision, erropt, localeopt)
$ZT(htime, tformat, precision, erropt, localeopt)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>htime</td>
<td>The internal system time that can be specified as a numeric value, the name of a variable, or as an expression.</td>
</tr>
<tr>
<td>tformat</td>
<td>Optional — An integer value that specifies the format in which you want to return the time value.</td>
</tr>
<tr>
<td>precision</td>
<td>Optional — A numeric value that specifies the number of decimal places of precision in which you want to express the time. If omitted, fractional seconds are truncated.</td>
</tr>
<tr>
<td>erropt</td>
<td>Optional — The expression returned if the htime parameter is considered invalid.</td>
</tr>
<tr>
<td>localeopt</td>
<td>Optional — A boolean flag that specifies which locale to use. When 0, the current locale determines the time separator, and the other characters, strings, and options used to format times. When 1, the ODBC locale determines these characters, strings, and options. The ODBC locale cannot be changed; it is used to format date and time strings that are portable between Caché processes that have made different National Language Support (NLS) choices. The default is 0.</td>
</tr>
</tbody>
</table>

Omitted parameters between specified parameter values are indicated by placeholder commas. Trailing placeholder commas are not required, but are permitted. Blank spaces are permitted between the commas that indicate an omitted parameter.

**Description**

The $ZTIME function converts an internal system time, htime, specified in the time format from the special variable $HOROLOG or $ZTIMESTAMP, to a printable format. If no optional parameters are used, the time will be returned in the format: “hh:mm:ss”; where “hh” is hours in a 24–hour clock, “mm” is minutes, and “ss” is seconds. Otherwise, the time will be returned in the format specified by the value of the tformat and precision parameters.

**Parameters**

- **htime**
  
  This value represents the number of elapsed seconds since midnight. It is the second component of a $HOROLOG value, which can be extracted by using $PIECE($HOROLOG, "," , 2). htime can be an integer, or a fractional number with the number of fractional digits of precision specified by precision.

  For tformat values –1 through 4, htime valid values must have their integer portion in the range 0 through 86399. (-0 is treated as 0.) Values outside of this range generate an <ILLEGAL VALUE> error. For tformat values 9 and 10, htime valid values can also include negative numbers and numbers greater than 86399.

- **tformat**
  
  Supported values are as follows:
**tformat**  |  **Description**  
---|---  
-1 | Get the effective format value from the TimeFormat property of the current locale, which defaults to a value of 1. This is the default behavior if you do not specify `tformat` and `localeopt` is unspecified or 0.  
1 | Express time in the form "hh:mm:ss" (24-hour clock).  
2 | Express time in the form “hh:mm” (24-hour clock).  
3 | Express time in the form “hh:mm:ss[AM/PM]” (12-hour clock).  
4 | Express time in the form “hh:mm[AM/PM]” (12-hour clock).  
9 | For MultiValue support. Same as `tformat` 1 ("hh:mm:ss" 24-hour clock) for numeric values in the range 0 through 86399. Also accepts negative time values and time values greater than 86399.  
10 | For MultiValue support. Same as `tformat` 2 ("hh:mm" 24-hour clock) for numeric values in the range 0 through 86399. Also accepts negative time values and time values greater than 86399.

To determine the default time format for your locale, invoke the `GetFormatItem()` NLS class method:
```objectscript
WRITE ##class(%SYS.NLS.Format).GetFormatItem("TimeFormat")
```

In 12-hour clock formats, morning and evening are represented by time suffixes, here shown as AM and PM. To determine the default time suffixes for your locale, invoke the following NLS class methods:
```objectscript
WRITE ##class(%SYS.NLS.Format).GetFormatItem("AM"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("PM"),!
```

**tformat 9 and 10**

Time formats 9 and 10 are provided for MultiValue support. They are identical to `tformats` 1 and 2 for all internal time values within the allowed range for ObjectScript: 0 through 86399. Like other time formats, -0 is treated as 0.

Time formats 9 and 10 accept negative numeric values and returns a corresponding negative time value. `tformat` 9 returns a negative `htime` as "-hh:mm:ss" (24-hour clock). `tformat` 10 returns a negative `htime` as "-hh:mm" (24-hour clock). For time format 10, "–59.9" returns “–00:00" and “–60" returns “–00:01".

Time formats 9 and 10 accept numeric values greater than 86399, and return the corresponding time. For time format 10, "86400" returns “24:00”, and “400000" returns “111:06".

Time format 9 returns whole seconds, and optionally returns fractional seconds, for all numeric values. The `precision` argument specifies the number of digits of fractional seconds.

**precision**

The function displays fractional seconds carried out to the number of decimal places specified in the `precision` parameter. For example, if you enter a value of 3 as `precision`, `$ZTIME` displays fractional seconds to three decimal places. If you enter a value of 9, `$ZTIME` displays fractional seconds to nine decimal places. Supported values are as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Gets the precision value from the TimePrecision property of the current locale, which defaults to a value of 0. This is the default behavior if you do not specify <code>precision</code>.</td>
</tr>
<tr>
<td><code>n</code></td>
<td>A value that is greater than or equal to 0 results in the expression of time to <code>n</code> decimal places.</td>
</tr>
<tr>
<td>0</td>
<td>If set to 0, or defaults to a value of 0, fractional seconds are truncated.</td>
</tr>
</tbody>
</table>
To determine the default time precision for your locale, invoke the `GetFormatItem()` NLS class method:

```ObjectScript
WRITE ##class(%SYS.NLS.Format).GetFormatItem("TimePrecision")
```

**erropt**

This parameter suppresses error messages associated with invalid `htime` values. Instead of generating `<ILLEGAL VALUE>` error messages, the function returns the value indicated by `erropt`.

**localeopt**

This parameter selects either the user’s current locale definition (0) or the ODBC locale definition (1) as the source for time options. The ODBC locale cannot be changed; it is used to format date and time strings that are portable between Caché processes that have made different National Language Support (NLS) choices.

The ODBC locale time definitions are as follows:

- Time format defaults to 1. Time separator is ".". Time precision is 0 (no fractional seconds).
- AM and PM indicators are "AM" and "PM". The words "Noon" and "Midnight" are used.

**Examples**

To return the current local time using the special variable `$HOROLOG`, you must use the `$PIECE` function to specify the second piece of `$HOROLOG`. The following returns the time in the 24-hour clock format "13:55:11":

```ObjectScript
WRITE $ZTIME($PIECE($HOROLOG,,2),1)
```

In the examples that follow, `htime` is set to `$PIECE($HOROLOG,,2)` for the current time. These examples show how to use the various forms of `$ZTIME` to return different time formats.

The following example in many cases returns time in the format "13:28:55"; however, this format is dependent on locale:

```ObjectScript
SET htime=$PIECE($HOROLOG,,2)
WRITE $ZTIME(htime)
```

The following example returns time in the format "13:28:55":

```ObjectScript
SET htime=$PIECE($HOROLOG,,2)
WRITE $ZTIME(htime,1)
```

The following example returns time in the format "13:28:55.999":

```ObjectScript
SET htime=$PIECE($HOROLOG,,2)
WRITE $ZTIME(htime,1,3)
```

The following example returns time in the format "13:28:55.999999999":

```ObjectScript
SET htime=$PIECE($HOROLOG,,2)
WRITE $ZTIME(htime,1,9)
```

The following example returns time in the format "01:28:24PM":

```ObjectScript
SET htime=$PIECE($HOROLOG,,2)
WRITE $ZTIME(htime,2)
```

The following example returns time in the format "01:28PM":

```ObjectScript
SET htime=$PIECE($HOROLOG,,2)
WRITE $ZTIME(htime,3)
```

The following example returns time in the format "00:28:24PM":

```ObjectScript
SET htime=$PIECE($HOROLOG,,2)
WRITE $ZTIME(htime,4)
```
The following example returns time in the format “13:45:56.021”, the current UTC time with three decimal places of precision:

```objectscript
SET t=$ZTIME($PIECE($ZTIMESTAMP,"",2),1,3)
WRITE "Current UTC time is ",t
```

**Notes**

**Invalid Parameter Values**

- You receive a <FUNCTION> error if you specify an invalid `tformat` value.
- You receive an <ILLEGAL VALUE> error for all `tformat` except 9 and 10 if you specify a value for `htime` outside the allowed range of 0 to 86399 (inclusive) and do not supply an `erropt` value.

**Decimal Delimiter**

$ZTIME will use the value of the DecimalSeparator property of the current locale as the delimiter between the whole and fractional parts of numbers. The default value of DecimalSeparator is “.”; all documentation examples use this delimiter.

To determine the default decimal separator for your locale, invoke the `GetFormatItem()` NLS class method:

```objectscript
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DecimalSeparator")
```

**Time Delimiter**

By default, Caché uses the value of the TimeSeparator property of the current locale to determine the delimiter character for the time string. By default, the delimiter is “:”; all documentation examples use this delimiter.

To determine the default time separator for your locale, invoke the following NLS class method:

```objectscript
WRITE ##class(%SYS.NLS.Format).GetFormatItem("TimeSeparator")
```

**Time Suffixes**

By default, Caché uses properties in the current locale to determine the names of its time suffixes. For $ZTIME, these properties (and their corresponding default values) are:

- AM (“AM”)
- PM (“PM”)

This documentation will always use these default values for these properties.

To determine the default time suffixes for your locale, invoke the following NLS class methods:

```objectscript
WRITE ##class(%SYS.NLS.Format).GetFormatItem("AM"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("PM")
```

**See Also**

- $ZDATETIME function
- $ZDATETIMEH function
- $ZTIMEH function
- $PIECE function
- $HOROLOG special variable
- $ZTIMESTAMP special variable
• More information on locales in the section on “System Classes for National Language Support” in Caché Specialized System Tools and Utilities.
$ZTIMEH

Validates a time and converts it from display format to Caché internal format.

$ZTIMEH(time, tformat, erropt, localeopt)
$ZTH(time, tformat, erropt, localeopt)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>The time value to be converted.</td>
</tr>
<tr>
<td>tformat</td>
<td>Optional — A numeric value that specifies the time format from which you are converting.</td>
</tr>
<tr>
<td>erropt</td>
<td>Optional — The expression returned if the <code>time</code> parameter is considered invalid.</td>
</tr>
<tr>
<td>localeopt</td>
<td>Optional — A boolean flag that specifies which locale to use. When 0, the current locale determines the time separator, and the other characters, strings, and options used to format times. When 1, the ODBC locale determines these characters, strings, and options. The ODBC locale cannot be changed; it is used to format date and time strings that are portable between Caché processes that have made different National Language Support (NLS) choices. The default is 0.</td>
</tr>
</tbody>
</table>

Omitted parameters between specified parameter values are indicated by placeholder commas. Trailing placeholder commas are not required, but are permitted. Blank spaces are permitted between the commas that indicate an omitted parameter.

**Description**

The `$ZTIMEH` function converts a time value from a format produced by the `$ZTIME` function to the format of the special variables `$HOROLOG` and `$ZTIMESTAMP`. If the optional parameter `tformat` is not specified, the input time must be in the format “hh:mm:ss.fff...”. Otherwise, the same integer format code used to produce the printable time from the `$ZTIME` function must be used for the time to be converted properly.

**Parameters**

**tformat**

Supported values are as follows:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Get the effective format value from the TimeFormat property of the current locale, which defaults to a value of 1. This is the default behavior if you do not specify <code>tformat</code> and <code>localeopt</code> is unspecified or 0.</td>
</tr>
<tr>
<td>1</td>
<td>Input time is in the form “hh:mm:ss” (24-hour clock).</td>
</tr>
<tr>
<td>2</td>
<td>Input time is in the form “hh:mm” (24-hour clock).</td>
</tr>
<tr>
<td>3</td>
<td>Input time is in the form “hh:mm:ss[AM/PM]” (12-hour clock).</td>
</tr>
<tr>
<td>4</td>
<td>Input time is in the form “hh:mm[AM/PM]” (12-hour clock).</td>
</tr>
</tbody>
</table>

To determine the default time format for your locale, invoke the `GetFormatItem()` NLS class method:

```ObjectScript
WRITE #class(%SYS.NLS.Format).GetFormatItem("TimeFormat")
```
**erropt**

This parameter suppresses error messages associated with invalid time values. Instead of generating <ILLEGAL VALUE> error messages, the function returns the value indicated by erropt.

**localeopt**

This parameter selects either the user’s current locale definition (0) or the ODBC locale definition (1) as the source for time options. The ODBC locale cannot be changed; it is used to format date and time strings that are portable between Caché processes that have made different National Language Support (NLS) choices.

The ODBC locale time definitions are as follows:

- Time format defaults to 1. Time separator is ":". Time precision is 0 (no fractional seconds).
- AM and PM indicators are "AM" and "PM". The words "Noon" and "Midnight" are used.

**Examples**

When the input time is “14:43:38”, the following examples both return 53018:

```
SET time=“14:43:38"
WRITE 1,$ZTIMEH(time)
WRITE 1,$ZTIMEH(time,1)
```

When the input time is “14:43:38.974”, the following example returns 53018.974:

```
SET time=“14:43:38.974"
WRITE $ZTIMEH(time,1)
```

**Notes**

**Fractional Seconds**

Unlike the $ZTIME function, $ZTIMEH does not allow you to specify a precision. Any fractional seconds in the original time format returned by $ZTIME are retained in the value returned by $ZTIMEH.

**Invalid Parameter Values**

You receive a <FUNCTION> error if you specify an invalid tformat code (an integer less than -1 or greater than 4, a zero, or a noninteger value).

If you do not supply an erropt value, you receive an <ILLEGAL VALUE> error under the following conditions:

- Specify a time with an hour value outside the allowed range of 0 to 23 (inclusive).
- Specify a time with a minute value outside the allowed range of 0 to 59 (inclusive).
- Specify a time with a second value outside the allowed range of 0 to 59 (inclusive).
- Specify a time value which uses a delimiter other than the value of the TimeSeparator property of the current locale.

**Time Delimiter**

By default, Caché uses the value of the TimeSeparator property of the current locale to determine the delimiter character for the time string. By default, the delimiter is “:”; all documentation examples use this delimiter.

To determine the default time separator for your locale, invoke the GetFormatItem() NLS class method:

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("TimeSeparator")
```
**Time Suffixes**

By default, Caché uses properties in the current locale to determine the names of its time suffixes. For `$ZTIMEH`, these properties (and their corresponding default values) are:

- AM ("AM")
- PM ("PM")
- Midnight ("MIDNIGHT")
- Noon ("NOON")

This documentation will always use these default values for these properties.

To determine the default time suffixes for your locale, invoke the `GetFormatItem()` NLS class method, as follows:

```ObjectScript
WRITE ##class(%SYS.NLS.Format).GetFormatItem("AM"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("PM"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("Midnight"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("Noon")
```

**See Also**

- `$ZDATETIME` function
- `$ZDATETIMEH` function
- `$ZTIME` function
- `$HOROLOG` special variable
- `$ZTIMESTAMP` special variable
- More information on locales in the section on “System Classes for National Language Support” in *Caché Specialized System Tools and Utilities*. 
$ZVERSION(1)

Returns the operating system type.

Parameter

The only supported parameter value is the number 1.

Description

$ZVERSION(1) returns the current operating system type as an integer code. It returns the following values: 2 for Windows, 3 for UNIX®, and 0 if not known.

You can use the isWINDOWS and isUNIX system-supplied macros to return the same information as a boolean value.

You can use the %SYSTEM.Version.GetOS() to return the same information as a string.

You can use the $ZVERSION special variable to return complete Caché version information, including the current operating system type.

Example

The following example returns the current operating system type:

```
#include %occInclude
WRITE "OS type as code: ", $ZVERSION(1),!
WRITE "OS type as Boolean: ", $$$isWINDOWS," UNIX? ", $$$isUNIX,!
WRITE "OS type as string: ", %SYSTEM.Version.GetOS(),!
WRITE "Caché Version: ", $ZVERSION
```

See Also

- $ZVERSION special variable.
$ZWASCII

Converts a two-byte string to a number.

$ZWASCII(string,position)
$ZWA(string,position)

Parameters

| string | A string. It can be a value, a variable, or an expression. It must be a minimum of two bytes in length. |
| position | Optional — A starting position in the string, expressed as a positive integer. The default is 1. Position is counted in single bytes, not two-byte strings. The position cannot be the last byte in the string, or beyond the end of the string. A numeric position value is parsed as an integer by truncating decimal digits, removing leading zeros and plus signs, etc. |

Description

The value that $ZWASCII returns depends on the parameters you use.

- $ZWASCII(string) returns a numeric interpretation of a two-byte string starting at the first character position of string.
- $ZWASCII(string,position) returns a numeric interpretation of a two-byte string beginning at the starting byte position specified by position.

Upon successful completion, $ZWASCII always returns a positive integer. $ZWASCII returns -1 if string is of an invalid length, or position is an invalid value.

Example

The following example determines the numeric interpretation of the character string "ab":

```
WRITE $ZWASCII("ab")
```

It returns 25185.

The following examples also return 25185:

```
WRITE !,$ZWASCII("ab",1)
WRITE !,$ZWASCII("abxx",1)
WRITE !,$ZWASCII("xxabxx",3)
```

In the following examples, string or position are invalid. The $ZWASCII function returns -1 in each case:

```
WRITE !,$ZWASCII("a")
WRITE !,$ZWASCII("aba",3)
WRITE !,$ZWASCII("ababab",99)
WRITE !,$ZWASCII("ababab",0)
WRITE !,$ZWASCII("ababab",-1)
```

Notes

$ZWASCII and $ASCII

$ZWASCII is similar to $ASCII except that it operates on two byte (16-bit) words instead of single 8-bit bytes. For four byte (32-bit) words, use $ZLASCII; For eight byte (64-bit) words, use $ZQASCII.

$ZWASCII(string,position) is the functional equivalent of:

$ASCII(string,position+1)*256+$ASCII(string,position)
$ZWASCII and $ZWCHAR

The $ZWCHAR function is the logical inverse of $ZWASCII. For example:

```objectscript
WRITE $ZWASCII("ab")
```

returns: 25185.

```objectscript
WRITE $ZWCHAR(25185)
```

returns “ab”.

See Also

- $ASCII function
- $ZWCHAR function
$ZWCHAR

Converts a number to a two-byte string.

$ZWCHAR(n)
$ZWC(n)

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>A positive integer in the range 0 through 65535. It can be specified as a value, a variable, or an expression.</td>
</tr>
</tbody>
</table>

Description

$ZWCHAR returns a two-byte (wide) character string corresponding to the binary representation of n. The bytes of the character string are presented in little-endian byte order, with the least significant byte first. It is the functional equivalent of:

WRITE $CHAR(n\#256,n\256)

If n is out of range or a negative number, $ZWCHAR returns the null string.

Example

The following example returns the two-byte string for the integer 25185:

WRITE $ZWCHAR(25185)

returns: ab

Notes

$ZWCHAR and $CHAR

$ZWCHAR is similar to $CHAR except that it operates on two byte (16-bit) words instead of single 8-bit bytes. For four byte (32-bit) words, use $ZLCHAR; For eight byte (64-bit) words, use $ZQCHAR.

$ZWCHAR and $ZWASCII

$ZWASCII is the logical inverse of the $ZWCHAR function. For example:

WRITE $ZWCHAR(25185)

returns: ab

WRITE $ZWASCII("ab")

returns: 25185

See Also

- $ZWASCII function
- $CHAR function
- $ZLCHAR function
- $ZQCHAR function
$ZWIDTH

Returns the total width of the characters in an expression.

$ZWIDTH(expression,pitch)

**Parameters**

<table>
<thead>
<tr>
<th>expression</th>
<th>A string expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>pitch</td>
<td>Optional — The numeric pitch value to use for full-width characters. The default is 2. Other permissible values are 1, 1.25, and 1.5. (These values with any number of trailing zeros are permissible.) All other pitch values result in a &lt;FUNCTION&gt; error.</td>
</tr>
</tbody>
</table>

**Description**

$ZWIDTH returns the total width of the characters in *expression*. The *pitch* value determines the width to use for full-width characters. All other characters are assigned a width of 1 and are considered to be half-width.

$ZWIDTH is only available in Unicode versions of Caché.

**Note:** $ZWIDTH can be abbreviated as $ZW in DSM-J mode. This abbreviation cannot be used in Caché mode.

**Example**

Assume that the variable *STR* contains two half-width characters followed by a full-width character:

```objectscript
WRITE $ZWIDTH(STR,1.5)
```

returns 3.5.

In this example, the two half-width characters total 2. Adding 1.5 (the specified pitch value) for the full-width characters produces a total of 3.5.

**Note**

Full-width characters are determined by examining the pattern-match table loaded for your Caché process. Any character with the full-width attribute is considered to be a full-width character. You can use the special ZFWCHARZ patcode to check for this attribute (char?1ZFWCHARZ). For more information about the full-width attribute, see the description of the $X/$Y Tab in the section on “System Classes for National Language Support” in *Caché Specialized System Tools and Utilities*.

**See Also**

- $ZPOSITION function
- $ZZENKAKU function
$ZWPACK and $ZWBPACK

Packs two 8-bit characters into a single 16-bit character.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ZWPACK(string)</td>
<td>Packs a string of 8-bit characters as a string of 16-bit wide characters in little-endian order. Two 8-bit characters are packed into a single 16-bit character.</td>
</tr>
<tr>
<td>$ZWBPACK(string)</td>
<td>Performs the same task, but the 8-bit characters are stored in 16-bit wide characters in big-endian order.</td>
</tr>
</tbody>
</table>

**Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A string consisting of two or more 8-bit characters. <em>string</em> must be an even number of characters.</td>
</tr>
</tbody>
</table>

**Description**

The $ZWPACK function packs a string of 8-bit characters as a string of 16-bit wide characters in little-endian order. Two 8-bit characters are packed into a single 16-bit character.

$ZWBPACK performs the same task, but the 8-bit characters are stored in 16-bit wide characters in big-endian order.

Packing a string is a way to halve the character count of the string for storage and string manipulation. Unpacking restores the original 8-bit character string for display. These operations should not be used when Unicode characters are permitted in the data.

The input *string* has the following requirements:

- *string* must consist of an even number of characters. The empty string is permitted, and returns the empty string.
  - Specifying an odd number of characters results in a <FUNCTION> error.
- *string* cannot contain any multibyte characters. You can use $ZISWIDE on *string* to check that it does not contain multibyte characters. If you use $ZWPACK or $ZWBPACK on a *string* containing multibyte characters, the system generates a <WIDE CHAR> error.

**Examples**

Note that the following examples require a Unicode installation of Caché.

The following example shows $ZWPACK packing four 8-bit characters into two 16-bit wide characters. Note the little-endian order of the bytes in the wide characters of the packed string: hexadecimal 4241 4443.

```objectscript
SET str=$CHAR(65,66,67,68)
WRITE !,$LENGTH(str)," characters: ",str
WRITE !,"$ZWPACK"
SET wstr=$ZWPACK(str)
WRITE !,$LENGTH(wstr)," packed characters: ",wstr
ZZDUMP wstr
WRITE !,"$ZWUNPACK"
SET nstr=$ZWUNPACK(wstr)
WRITE !,$LENGTH(nstr)," unpacked characters: ",nstr
```

The following example shows $ZWBPACK packing four 8-bit characters into two 16-bit wide characters. Note the big-endian order of the bytes in the wide characters of the packed string: hexadecimal 4142 4344.

```objectscript
SET str=$CHAR(65,66,67,68)
WRITE !,$LENGTH(str)," characters: ",str
WRITE !,"$ZWBPACK"
SET wstr=$ZWBPACK(str)
WRITE !,$LENGTH(wstr)," packed characters: ",wstr
ZZDUMP wstr
WRITE !,"$ZWBUNPACK"
SET nstr=$ZWBUNPACK(wstr)
WRITE !,$LENGTH(nstr)," unpacked characters: ",nstr
```
The following example validates *string* before packing it:

```objectscript
SET str=$CHAR(65,66,67,68)
IF $ZISWIDE(str) {
   WRITE !,str," contains wide characters"
   QUIT }
ELSEIF $LENGTH(str) # 2 {
   WRITE !,str," contains an odd number of characters"
   QUIT }
ELSE {
   WRITE !,str," passes validation"
   WRITE !,$LENGTH(str)," characters: ",str
   WRITE !,$LENGTH(wstr)," packed characters: ",wstr
   ZZDUMP wstr
```

**See Also**

- `$LENGTH` function
- `$ZISWIDE` function
- `$ZWUNPACK` and `$ZWBUNPACK` functions
$ZWUNPACK and $ZWBUNPACK

Unpacks a single 16-bit character to two 8-bit characters.

$ZWUNPACK(string)
$ZWBUNPACK(string)

Parameter

| string | A string consisting of one or more 16-bit characters. |

Description

$ZWUNPACK is a function that takes one or more two-byte wide characters and “unpacks” them, returning the corresponding pairs of single-byte characters in little endian order.

$ZWBUNPACK performs the same task, but the two-byte wide characters are unpacked in big-endian order.

Packing a string is a way to halve the character count of the string for storage and string manipulation. Unpacking restores the original 8-bit character string for display. These operations should not be used when Unicode characters are permitted in the data.

The input string should consist entirely of 16-bit wide characters created using $ZWPACK or $ZWBPACK. The empty string is permitted, and returns the empty string. string should not contain any 16-bit Unicode characters, or any 8-bit characters.

You can use $ZISWIDE on string to check that it contains multibyte characters. However, you must use $ZISWIDE on each character to ensure that the string does not contain a mix of 16-bit and 8-bit characters. $ZISWIDE does not distinguish between Unicode and packed 16-bit characters.

Examples

Note that the following examples require a Unicode installation of Caché.

The following example unpacks a string (“ABCD”) that was packed using $ZWPACK. It unpacks two 16-bit wide characters into four 8-bit characters. Note the little-endian order of the bytes in the wide characters of the packed string: hexadecimal 4241 4443.

```
IF $SYSTEM.Version.IsUnicode() {
SET str=$CHAR(65,66,67,68)
WRITE !,$LENGTH(str)," characters: ",str
WRITE !,"$ZWPACK"
SET wstr=$ZWPACK(str)
WRITE !,$LENGTH(wstr)," packed characters: ",wstr
ZZDUMP wstr
WRITE !,"$ZWUNPACK"
SET nstr=$ZWUNPACK(wstr)
WRITE !,$LENGTH(nstr)," unpacked characters: ",nstr
}
ELSE {WRITE "This example requires a Unicode installation of Caché")
```

The following example performs the same operation as the previous example, but uses big-endian order. Note the big-endian order of the bytes in the wide characters of the packed string: hexadecimal 4142 4344.
The following example shows what happens when you “unpacks” a string of 8-bit characters. Note that the unpacking operation assumes each character to be a 16-bit wide character, and thus supplies the missing eight bits as hexadecimal 00. This use of `$ZWUNPACK` is not recommended.

```
IF $SYSTEM.Version.IsUnicode() {
SET str=$CHAR(65,66,67,68)
WRITE !,$LENGTH(str)," characters: ",str
SET wstr=$ZWBPACK(str)
WRITE !,$LENGTH(wstr)," packed characters: ",wstr
ZZDUMP wstr
WRITE !,"$ZWBUNPACK"
SET nstr=$ZWBUNPACK(wstr)
WRITE !,$LENGTH(nstr)," unpacked characters: ",nstr
}
ELSE {WRITE "This example requires a Unicode installation of Caché"
}
```

The following example shows what happens when you “unpacks” a string of 16-bit Unicode characters; in this case, lowercase Greek letters. This use of `$ZWUNPACK` is not recommended.

```
IF $SYSTEM.Version.IsUnicode() {
SET str=$CHAR(945,946,947)
WRITE !,$LENGTH(str)," characters: ",str
SET nstr=$ZWUNPACK(str)
WRITE !,$LENGTH(nstr)," unpacked characters: ",nstr
ZZDUMP nstr
}
ELSE {WRITE "This example requires a Unicode installation of Caché"
}
```

See Also

- `$LENGTH` function
- `$ZISWIDE` function
- `$ZWPACK` and `$ZWBPACK` functions
$ZZENKAKU

Converts Japanese katakana characters from half-width to full-width.

$ZZENKAKU(expression,flag1,flag2)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A string containing half-width characters. These characters may be katakana characters, Roman alphabet letters, or numbers.</td>
</tr>
<tr>
<td>flag1</td>
<td>Optional — A boolean flag to indicate whether conversion to full-width hiragana (0) or conversion to full-width katakana (1) is required.</td>
</tr>
<tr>
<td>flag2</td>
<td>Optional — A boolean flag to indicate whether voiced sound processing is required (1) or not required (0).</td>
</tr>
</tbody>
</table>

**Description**

$ZZENKAKU converts Japanese katakana characters from half-width (hankaku) to full-width (zenkaku) characters. It also converts strings of Roman alphabet letters (“ABC”) and Arabic numbers (123) from half-width to full-width.

$ZZENKAKU is only available in Unicode versions of Caché.

$ZZENKAKU can, optionally, convert half-width katakana to full-width hiragana. Katakana characters are commonly used for foreign terms and foreign loan words; they can be represented as half-width or full-width characters. Hiragana characters are the more standard way of writing Japanese. Hiragana is always full-width.

If flag1 is 0, $ZZENKAKU converts printable ASCII characters to their full-width counterparts and converts half-width katakana characters to full-width hiragana characters. The default value for flag1 is 0.

If flag1 is 1, $ZZENKAKU converts printable ASCII characters to their full-width counterparts and converts half-width katakana characters to full-width katakana characters.

If flag2 is 1 and a half-width katakana character is followed by a voice sound mark or a semi-voice sound mark, then (if appropriate) $ZZENKAKU combines the half-width katakana character and the sound mark character into a target full-width hiragana or katakana character. The default value for flag2 is 1.

**Note:** $ZZENKAKU can be abbreviated as $ZZ in DSM-J mode. This abbreviation cannot be used in Caché mode.

The Japanese language modes are DSM-J (language mode 6) and DTM-J (language mode 7). To determine or to change the current language mode, and for a list of available language modes, see the `LanguageMode()` method of the `%SYSTEM.Process` class.

You can set the physical cursor to use two physical spaces for a character as system-wide behavior by setting the `PhysicalCursor` property of the `Config.NLS.Locales` class.

The `$WASCII` function (and other `$W` functions) supports surrogate pairs of characters used to encode some Japanese kanji characters. For the `ZFWCHARZ` and `ZHWKATAZ` Japanese language pattern match codes, refer to Pattern Matching in Using Caché ObjectScript.

**Examples**

The following example returns the half-width katakana characters “あ”, “め”, “り”, “カ” (America):

```
ZZDUMP $CHAR(65383,65426,65432,65398)
```

The following example converts these half-width katakana characters to the corresponding full-width katakana characters:
The following examples both convert these half-width katakana characters to the corresponding full-width hiragana characters. Note that `$ZZENKAKU` converts from katakana to hiragana by default:

```objectscript
ZZDUMP $ZZENKAKU(CHAR(65383, 65426, 65432, 65398), 1)
ZZDUMP $ZZENKAKU(CHAR(65383, 65426, 65432, 65398), 0)
ZZDUMP $ZZENKAKU(CHAR(65383, 65426, 65432, 65398))
```

### See Also

- `$ZPOSITION` function
- `$ZWIDTH` function
The $ZUTIL Replacement List
Replacements for ObjectScript $ZUTIL Functions

Class methods and properties to replace $ZUTIL ($ZU) functions that are no longer documented.

Obsolete And Deprecated Functionality

Some $ZUTIL functionality is obsolete or deprecated. Beginning with version 2010.1 of Caché, documentation for legacy networking facilities DDP and DCP has been removed. The associated $ZUTIL functions are classed as deprecated; no replacement class methods or properties have been provided. This is also true for several other $ZUTIL functions because they no longer function as intended. Beginning with version 2010.2 of Caché, $ZUTIL(69,39) is also deprecated. The following $ZUTIL functions are not provided with equivalent replacements; the continued use of these $ZUTIL functions is discouraged:

<table>
<thead>
<tr>
<th>$ZUTIL Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>67, 1, …</td>
<td>Returns the activity state of a specified process, and resets.</td>
</tr>
<tr>
<td>68, 6, …</td>
<td>Enables or disables reliable SET networking mode for the current process.</td>
</tr>
<tr>
<td>68, 27, …</td>
<td>Sets or clears network hardening for the current process. Network hardening was only used by the obsolete DCP and DDP networks. It does not apply to ECP networks.</td>
</tr>
<tr>
<td>68,39, …</td>
<td>Enables or disables caching for the current process.</td>
</tr>
<tr>
<td>68, 55, …</td>
<td>Selects $X/$Y behavior for TCP devices for the current process.</td>
</tr>
<tr>
<td>69, 6, …</td>
<td>Sets reliable SET networking mode system-wide.</td>
</tr>
<tr>
<td>69, 13, …</td>
<td>Sets logging of asynchronous SET/KILL errors.</td>
</tr>
<tr>
<td>69, 14, …</td>
<td>Sets logging of async errors to console.</td>
</tr>
<tr>
<td>69, 19, …</td>
<td>Sets DDP password security system-wide.</td>
</tr>
<tr>
<td>69, 20, …</td>
<td>Transfers global nodes with null subscripts with DSM-DDP.</td>
</tr>
<tr>
<td>69, 27, …</td>
<td>Enables or disables network hardening system-wide.</td>
</tr>
<tr>
<td>69, 28, …</td>
<td>Controls root-level (unsubscribed) global node kills system-wide.</td>
</tr>
<tr>
<td>69, 31, …</td>
<td>Sets network locks handling system-wide following a DCP outage.</td>
</tr>
<tr>
<td>69, 35, …</td>
<td>Sets silent retry for domainspace connection attempts system-wide.</td>
</tr>
<tr>
<td>69, 39, …</td>
<td>Sets caching for future processes system-wide.</td>
</tr>
<tr>
<td>69, 55, …</td>
<td>Selects $X/$Y behavior for TCP devices system-wide.</td>
</tr>
<tr>
<td>69, 67, …</td>
<td>Suppresses or displays the stack and register usage message box system-wide.</td>
</tr>
<tr>
<td>78, 28, …</td>
<td>Returns journal directory block information.</td>
</tr>
<tr>
<td>90, 4, …</td>
<td>Starts up in a specified namespace (UNIX®).</td>
</tr>
<tr>
<td>100</td>
<td>Determines which Windows operating system is running.</td>
</tr>
<tr>
<td>113</td>
<td>Reclaims routine and directory blocks.</td>
</tr>
<tr>
<td>130, …</td>
<td>Sets or returns the domain ID or index.</td>
</tr>
<tr>
<td>133, …</td>
<td>Maintains a set of metric counters.</td>
</tr>
</tbody>
</table>
$ZUTIL Function Replacements

Beginning with Caché version 2010.1, the ObjectScript $ZUTIL functions are no longer documented. These functions remain available for use by applications as in prior releases, but the actions they perform are now also available by other means. The list of $ZUTIL functions, their description, and the new and preferred means of accomplishing the same objective are given here.

The following three tables list the $ZUTIL functions for which there are equivalent replacements.

- The first table lists a variety of operations that do not change configuration.
- The second table lists per-process configuration operations ($ZUTIL(68) operations).
- The third table lists system-wide configuration operations ($ZUTIL(69) operations).

Table E–1: Non-Configuration-Related Functions

<table>
<thead>
<tr>
<th>$ZUTIL Function</th>
<th>Description</th>
<th>Class</th>
<th>Property / Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>4, …</td>
<td>Terminates a Cache process.</td>
<td>%SYSTEM.Process</td>
<td>Terminate()</td>
</tr>
<tr>
<td>5, …</td>
<td>Returns current namespace or switches to another namespace.</td>
<td>&lt;Special variable&gt;</td>
<td>$NAMESPACE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$NAMESPACE is not an exact replacement for $ZUTIL(5); however, $NAMESPACE is almost always preferable, and existing use of $ZUTIL(5) should be phased out. $NAMESPACE provides full support for implied namespaces by establishing default mapping for system routines. $ZUTIL(5) handling of implied namespaces does not establish this default mapping.</td>
</tr>
<tr>
<td>9, …</td>
<td>Broadcasts a message to a specified device.</td>
<td>%Library.Device</td>
<td>Broadcast()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>WriteToConsoleLog()</td>
</tr>
<tr>
<td>12, …</td>
<td>Converts file or directory name to canonical form.</td>
<td>%Library.File</td>
<td>NormalizeDirectory()</td>
</tr>
<tr>
<td>15, …</td>
<td>Converts RMS filename to canonical form.</td>
<td>%Library.File</td>
<td>NormalizeFilename()</td>
</tr>
<tr>
<td>18, …</td>
<td>Sets undefined variable handling for the current process.</td>
<td>%SYSTEM.Process</td>
<td>Undefined()</td>
</tr>
<tr>
<td>20, …</td>
<td>Specifies the namespace(s) that contains the routine dataset.</td>
<td>%SYSTEM.Process</td>
<td>UserRoutinePath()</td>
</tr>
<tr>
<td>$ZUTIL Function</td>
<td>Description</td>
<td>Class</td>
<td>Property / Method</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>21, …</td>
<td>Returns the location of process-private globals.</td>
<td>%SYSTEM.Process</td>
<td>PrivateGlobalLocation()</td>
</tr>
<tr>
<td>21</td>
<td>Deletes all process-private globals.</td>
<td>%SYSTEM.Process</td>
<td>KillPrivateGlobals()</td>
</tr>
<tr>
<td>22, 0, …</td>
<td>Specifies the form feed or backspace control code sequence.</td>
<td>%Library.Device</td>
<td>SetFFBS()</td>
</tr>
<tr>
<td>28, …</td>
<td>Performs collation conversion.</td>
<td>%SYSTEM.Util</td>
<td>Collation()</td>
</tr>
<tr>
<td>39, …</td>
<td>Specifies a search path for percent (%) routines.</td>
<td>%SYSTEM.Process</td>
<td>SysRoutinePath()</td>
</tr>
<tr>
<td>49, …</td>
<td>Obtains database information.</td>
<td>SYS.Database</td>
<td>&lt;various class properties&gt;</td>
</tr>
<tr>
<td>53</td>
<td>Passes TCP device name to child process.</td>
<td>%SYSTEM.InetInfo</td>
<td>TCPName()</td>
</tr>
<tr>
<td>53, …</td>
<td>Returns TCP statistics.</td>
<td>%SYSTEM.InetInfo</td>
<td>TCPStats()</td>
</tr>
<tr>
<td>55, …</td>
<td>Returns or changes the current language mode.</td>
<td>%SYSTEM.Process</td>
<td>LanguageMode()</td>
</tr>
<tr>
<td>56, 2</td>
<td>Locates source file and line of code for last ObjectScript error.</td>
<td>%SYSTEM.Process</td>
<td>ErrorLine()</td>
</tr>
<tr>
<td>56, 6</td>
<td>Returns the operating system error code for a sequential file error.</td>
<td>%SYSTEM.Process</td>
<td>OSErr()</td>
</tr>
<tr>
<td>62, …</td>
<td>Performs a syntax check of command line code.</td>
<td>%Library.Routine</td>
<td>CheckSyntax()</td>
</tr>
<tr>
<td>67, 0, …</td>
<td>Returns the activity state of a specified process.</td>
<td>%SYS.ProcessQuery %SYSTEM.Process</td>
<td>IsGhost</td>
</tr>
<tr>
<td>67, 4, …</td>
<td>Returns the process state.</td>
<td>%SYS.ProcessQuery %SYSTEM.Process</td>
<td>State</td>
</tr>
<tr>
<td>67, 5, …</td>
<td>Returns the routine name of a specified process.</td>
<td>%SYS.ProcessQuery %SYSTEM.Process</td>
<td>Routine</td>
</tr>
<tr>
<td>67, 6, …</td>
<td>Returns the namespace name for a specified process.</td>
<td>%SYS.ProcessQuery %SYSTEM.Process</td>
<td>NameSpace</td>
</tr>
<tr>
<td>$ZUTIL Function</td>
<td>Description</td>
<td>Class</td>
<td>Property / Method</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
<td>-------</td>
<td>-------------------</td>
</tr>
<tr>
<td>67, 7, …</td>
<td>Returns the device name for the specified process.</td>
<td>%SYS.ProcessQuery, %SYSTEM.Process</td>
<td>CurrentDevice, CurrentDevice()</td>
</tr>
<tr>
<td>67, 8, …</td>
<td>Returns the number of lines executed by the specified process.</td>
<td>%SYS.ProcessQuery, %SYSTEM.Process</td>
<td>LinesExecuted, LinesExecuted()</td>
</tr>
<tr>
<td>67, 9, …</td>
<td>Returns the number of global references made by the specified process.</td>
<td>%SYS.ProcessQuery, %SYSTEM.Process</td>
<td>GlobalReferences, GlobalReferences()</td>
</tr>
<tr>
<td>67, 10, …</td>
<td>Returns the job type of the specified process.</td>
<td>%SYS.ProcessQuery, %SYSTEM.Process</td>
<td>JobType, JobType()</td>
</tr>
<tr>
<td>67, 11, …</td>
<td>Returns the username of the owner of the specified process.</td>
<td>%SYS.ProcessQuery, %SYSTEM.Process</td>
<td>UserName, UserName()</td>
</tr>
<tr>
<td>67, 12, …</td>
<td>Returns the name of the system for a client application.</td>
<td>%SYS.ProcessQuery, %SYSTEM.Process</td>
<td>ClientNodeName, ClientNodeName()</td>
</tr>
<tr>
<td>67, 13, …</td>
<td>Returns the name of the executable for a client application.</td>
<td>%SYS.ProcessQuery, %SYSTEM.Process</td>
<td>ClientExecutableName, ClientExecutableName()</td>
</tr>
<tr>
<td>67, 14, …</td>
<td>Formerly returned the operating system running a client application. Now returns the CSP Session ID.</td>
<td>%SYS.ProcessQuery, %SYSTEM.Process</td>
<td>CSPSessionID, CSPSessionID()</td>
</tr>
<tr>
<td>67, 15, …</td>
<td>Returns the IP address of a client application.</td>
<td>%SYS.ProcessQuery, %SYSTEM.Process</td>
<td>ClientIPAddress, ClientIPAddress()</td>
</tr>
<tr>
<td>71, …</td>
<td>Sets date to a fixed value for the current process.</td>
<td>%SYSTEM.Process</td>
<td>FixedDate()</td>
</tr>
<tr>
<td>78, 21</td>
<td>Searches journal file for open transactions.</td>
<td>%SYS.Journal.System</td>
<td>GetImageJournalInfo()</td>
</tr>
<tr>
<td>78, 22, …</td>
<td>Returns journaling information.</td>
<td>%SYS.Journal.System</td>
<td>&lt;various class methods&gt;</td>
</tr>
<tr>
<td>78, 23, …</td>
<td>Deletes a journal file.</td>
<td>%SYS.Journal.File</td>
<td>PurgeOne()</td>
</tr>
<tr>
<td>78, 29</td>
<td>Flashes journal buffer.</td>
<td>%SYS.Journal.System</td>
<td>Sync()</td>
</tr>
<tr>
<td>82, 12, …</td>
<td>Redirects I/O operations.</td>
<td>%Library.Device</td>
<td>ReDirectIO()</td>
</tr>
<tr>
<td>86</td>
<td>Returns configuration file pathname and config name.</td>
<td>%SYS.System</td>
<td>GetCPFFileName()</td>
</tr>
<tr>
<td>$ZUTIL Function</td>
<td>Description</td>
<td>Class</td>
<td>Property / Method</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
<td>-------</td>
<td>-------------------</td>
</tr>
<tr>
<td>90, 10, …</td>
<td>Tests whether a namespace is defined.</td>
<td>%SYS.Namespace</td>
<td>Exists()</td>
</tr>
<tr>
<td>94, …</td>
<td>Broadcasts a message to a specified process.</td>
<td>%SYSTEM.Process</td>
<td>Broadcast()</td>
</tr>
<tr>
<td>96, 3, …</td>
<td>Return error number for user-defined command.</td>
<td>%SYSTEM.Process</td>
<td>ThrowError()</td>
</tr>
<tr>
<td>96, 4, …</td>
<td>Sets $TEST to reflect I/O redirection.</td>
<td>%SYSTEM.Process</td>
<td>IODollarTest()</td>
</tr>
<tr>
<td>96, 5, …</td>
<td>Sets the $DEVICE special variable.</td>
<td>&lt;Special variable&gt;</td>
<td>$DEVICE</td>
</tr>
<tr>
<td>96, 9</td>
<td>Returns the calling routine name.</td>
<td>%SYSTEM.Process</td>
<td>CallingRoutine()</td>
</tr>
<tr>
<td>96, 10</td>
<td>Returns the calling routine database.</td>
<td>%SYSTEM.Process</td>
<td>CallingDatabase()</td>
</tr>
<tr>
<td>96, 14</td>
<td>Returns the current device type.</td>
<td>%Library.Device</td>
<td>GetType()</td>
</tr>
<tr>
<td>110</td>
<td>Returns the name of the system that is running.</td>
<td>%SYS.System</td>
<td>GetNodeName()</td>
</tr>
<tr>
<td>114, …</td>
<td>Determines Ethernet address.</td>
<td>%SYSTEM.INetInfo</td>
<td>EthernetAddress()</td>
</tr>
<tr>
<td>115, 11, …</td>
<td>Specifies whether a value can be inserted into an identity column.</td>
<td>%SYSTEM.SQL</td>
<td>GetIdentityInsert(), SetIdentityInsert()</td>
</tr>
<tr>
<td>128, 1</td>
<td>Returns location of last single step during debugging.</td>
<td>%SYSTEM.Process</td>
<td>StepInfo()</td>
</tr>
<tr>
<td>132</td>
<td>Makes the last device in use the principal I/O device.</td>
<td>%Library.Device</td>
<td>ChangePrincipal()</td>
</tr>
<tr>
<td>140, 1, …</td>
<td>Returns file, directory, and disk information and performs file operations.</td>
<td>%Library.File</td>
<td>&lt;various class methods&gt;</td>
</tr>
<tr>
<td>140, 7, …</td>
<td>Returns a bitmap of file attributes.</td>
<td>%Library.File</td>
<td>Attributes()</td>
</tr>
<tr>
<td>147, …</td>
<td>Handles spaces in pathnames for the host platform.</td>
<td>%Library.File</td>
<td>NormalizeFilenameWithSpaces()</td>
</tr>
<tr>
<td>$ZUTIL Function</td>
<td>Description</td>
<td>Class</td>
<td>Property / Method</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
<td>-------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>158, 0</td>
<td>Returns number of currently installed printers.</td>
<td>%Library.Device</td>
<td>InstalledPrinters()</td>
</tr>
<tr>
<td>158, 1</td>
<td>Lists currently installed printers.</td>
<td>%Library.Device</td>
<td>GetPrinters()</td>
</tr>
<tr>
<td>168, ...</td>
<td>Returns location of current working directory, or sets current working directory.</td>
<td>%SYSTEM.Process</td>
<td>CurrentDirectory()</td>
</tr>
<tr>
<td>186, ...</td>
<td>Sets display in programmer prompt for the current process.</td>
<td>%SYSTEM.Process</td>
<td>TerminalPrompt()</td>
</tr>
<tr>
<td>188</td>
<td>Returns local date and time with fractional seconds system-wide.</td>
<td>&lt;Function&gt;</td>
<td>$NOW</td>
</tr>
<tr>
<td></td>
<td>$NOW is not an exact replacement for $ZUTIL(188). $ZUTIL(188) returns current local time with fractional seconds, adjusted for local time variants such as Daylight Saving Time (DST). $NOW can return current local time with fractional seconds, but does not adjust for local time variants. $HOROLOG returns current local time adjusted for local time variants, but does not support fractional seconds.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>189</td>
<td>Checks if TCP device is disconnected.</td>
<td>%SYSTEM.INetInfo</td>
<td>Connected()</td>
</tr>
<tr>
<td>193, ...</td>
<td>Converts Coordinated Universal Time (UTC) to local date and time (and vice versa).</td>
<td>&lt;Function&gt;</td>
<td>$ZDATETIMEH</td>
</tr>
</tbody>
</table>

Table E–2: Process-Related Configuration Functions

<table>
<thead>
<tr>
<th>$ZUTIL Function</th>
<th>Description</th>
<th>Class</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>68,0</td>
<td>Sets undefined variable handling for the current process.</td>
<td>%SYSTEM.Process</td>
<td>Undefined()</td>
</tr>
<tr>
<td>68, 1, ...</td>
<td>Enables or disables use of null subscripts for the current process.</td>
<td>%SYSTEM.Process</td>
<td>NullSubscripts()</td>
</tr>
<tr>
<td>68, 2, ...</td>
<td>Sets sequential file open mode for the current process.</td>
<td>%SYSTEM.Process</td>
<td>OpenMode()</td>
</tr>
<tr>
<td>$ZUTIL Function</td>
<td>Description</td>
<td>Class</td>
<td>Method</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>68, 3, …</td>
<td>Sets automatic sequential file creation option for the current process.</td>
<td>%SYSTEM.Process</td>
<td>FileMode()</td>
</tr>
<tr>
<td>68, 5, …</td>
<td>Enables or disables processing of argumentless BREAK commands for the current process.</td>
<td>%SYSTEM.Process</td>
<td>BreakMode()</td>
</tr>
<tr>
<td>68, 7, …</td>
<td>Retains or strips extended global reference from globals returned to the current process.</td>
<td>%SYSTEM.Process</td>
<td>RefInKind()</td>
</tr>
<tr>
<td>68, 11, …</td>
<td>Enables or disables read line recall for the current process.</td>
<td>%SYSTEM.Process</td>
<td>LineRecall()</td>
</tr>
<tr>
<td>68, 15, …</td>
<td>Enables or disables I/O device disconnect detection for the current process.</td>
<td>%SYSTEM.Process</td>
<td>DisconnectErr()</td>
</tr>
<tr>
<td>68, 21, …</td>
<td>Sets synchronous commit mode for the current process.</td>
<td>%SYSTEM.Process</td>
<td>SynchCommit()</td>
</tr>
<tr>
<td>68, 22, …</td>
<td>Sets handling of escape sequences when $X is updated for the current process.</td>
<td>%SYSTEM.Process</td>
<td>DX()</td>
</tr>
<tr>
<td>68, 26, …</td>
<td>Sets namespace display in programmer prompt for the current process.</td>
<td>%SYSTEM.Process</td>
<td>TerminalPrompt()</td>
</tr>
<tr>
<td>68, 28, …</td>
<td>Restricts or permits kills of root-level global nodes for the current process.</td>
<td>%SYSTEM.Process</td>
<td>GlobalKillDisabled()</td>
</tr>
<tr>
<td>68, 30, …</td>
<td>Sets error handling behavior for the current process.</td>
<td>%SYSTEM.Process</td>
<td>PopError()</td>
</tr>
<tr>
<td>68, 32, …</td>
<td>Sets date range and invalid date behavior for the current process.</td>
<td>%SYSTEM.Process</td>
<td>ZDateNull()</td>
</tr>
<tr>
<td>68, 34, …</td>
<td>Sets whether asynchronous errors can interrupt the current process.</td>
<td>%SYSTEM.Process</td>
<td>AsynchError()</td>
</tr>
<tr>
<td>$ZUTIL Function</td>
<td>Description</td>
<td>Class</td>
<td>Method</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>68, 40, …</td>
<td>Sets sequential file end-of-file handling for the current process.</td>
<td>%SYSTEM.Process</td>
<td>SetZEOF()</td>
</tr>
<tr>
<td>68, 42, …</td>
<td>Sets $JOB format for the current process.</td>
<td>%SYSTEM.Process</td>
<td>NodeNameInPid()</td>
</tr>
<tr>
<td>68, 43, …</td>
<td>Sets clearing of global vectors for the current process.</td>
<td>%SYSTEM.Process</td>
<td>OldZU5()</td>
</tr>
<tr>
<td>68, 45, …</td>
<td>Sets truncation mode for string-to-number conversions for the current process.</td>
<td>%SYSTEM.Process</td>
<td>TruncateOverflow()</td>
</tr>
<tr>
<td>68, 51, …</td>
<td>Sets whether or not changing namespaces changes operating system directories for the current process.</td>
<td>%SYSTEM.Process</td>
<td>SwitchOSdir()</td>
</tr>
<tr>
<td>68, 60, …</td>
<td>Sets handling of asynchronous Telnet disconnect errors for the current process.</td>
<td>%SYSTEM.Process</td>
<td>AsyncDisconnectErr()</td>
</tr>
<tr>
<td>68, 63, …</td>
<td>Enables or disables the use of “e” as scientific notation symbol for the current process.</td>
<td>%SYSTEM.Process</td>
<td>ScientificNotation()</td>
</tr>
<tr>
<td>68, 66, …</td>
<td>Suppress Telnet NUL at end-of-line for the current process.</td>
<td>%SYSTEM.Process</td>
<td>TelnetNUL()</td>
</tr>
<tr>
<td>68, 67, …</td>
<td>Suppresses or displays the stack and register usage message box for the current process.</td>
<td>%SYSTEM.Process</td>
<td>ExceptionLog()</td>
</tr>
<tr>
<td>68, 70, …</td>
<td>Enables or disables $DOUBLE returning INF and NAN values for the current process.</td>
<td>%SYSTEM.Process</td>
<td>IEEEError()</td>
</tr>
<tr>
<td>68, 71, …</td>
<td>Sets IP address format for the current process.</td>
<td>%SYSTEM.Process</td>
<td>IPv6Format()</td>
</tr>
<tr>
<td>68, 72, …</td>
<td>Sets MVBasic handling of undefined variables for the current process.</td>
<td>%SYSTEM.Process</td>
<td>MVUndefined()</td>
</tr>
<tr>
<td>$ZUTIL Function</td>
<td>Description</td>
<td>Class</td>
<td>Property</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
<td>-------</td>
<td>----------</td>
</tr>
<tr>
<td>69, 0, …</td>
<td>Sets undefined variable default handling system-wide.</td>
<td>Config.Miscellaneous</td>
<td>Undefined</td>
</tr>
<tr>
<td>69, 1, …</td>
<td>Sets null subscript mode default system-wide.</td>
<td>Config.Miscellaneous</td>
<td>NullSubscripts</td>
</tr>
<tr>
<td>69, 2, …</td>
<td>Sets sequential file open mode default system-wide.</td>
<td>Config.Miscellaneous</td>
<td>OpenMode</td>
</tr>
<tr>
<td>69, 3, …</td>
<td>Sets automatic sequential file creation system-wide.</td>
<td>Config.Miscellaneous</td>
<td>FileMode</td>
</tr>
<tr>
<td>69, 5, …</td>
<td>Enables argumentless BREAK processing system-wide.</td>
<td>Config.Miscellaneous</td>
<td>BreakMode</td>
</tr>
<tr>
<td>69, 7, …</td>
<td>Retains or strips extended global reference from globals system-wide.</td>
<td>Config.Miscellaneous</td>
<td>RefInKind</td>
</tr>
<tr>
<td>69, 8, …</td>
<td>Sets ZA and ZD locking modes system-wide.</td>
<td>Config.Miscellaneous</td>
<td>ZaMode</td>
</tr>
<tr>
<td>69, 10, …</td>
<td>Sets system behavior when journal is full.</td>
<td>Config.Journal</td>
<td>FreezeOnError</td>
</tr>
<tr>
<td>69, 11, …</td>
<td>Sets Read Line Recall mode system-wide.</td>
<td>Config.Miscellaneous</td>
<td>LineRecall</td>
</tr>
<tr>
<td>69, 15, …</td>
<td>Sets I/O device disconnect detection system-wide.</td>
<td>Config.Miscellaneous</td>
<td>DisconnectErr</td>
</tr>
<tr>
<td>69, 21, …</td>
<td>Sets synchronous commit mode system-wide.</td>
<td>Config.Miscellaneous</td>
<td>SynchCommit</td>
</tr>
<tr>
<td>69, 22, …</td>
<td>Sets $X update mode for escape sequences system-wide.</td>
<td>Config.Miscellaneous</td>
<td>DX</td>
</tr>
<tr>
<td>69, 26, …</td>
<td>Sets namespace display in programmer prompt system-wide.</td>
<td>Config.Startup</td>
<td>TerminalPrompt</td>
</tr>
<tr>
<td>69, 30, …</td>
<td>Sets error handling behavior system-wide.</td>
<td>Config.Miscellaneous</td>
<td>PopError</td>
</tr>
<tr>
<td>69, 32, …</td>
<td>Sets date range and invalid date behavior system-wide.</td>
<td>Config.Miscellaneous</td>
<td>ZDateNull</td>
</tr>
<tr>
<td>$ZUTIL Function</td>
<td>Description</td>
<td>Class</td>
<td>Property</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
<td>----------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>69, 34, …</td>
<td>Sets interruptability of processes by asynchronous errors system-wide.</td>
<td>Config.Miscellaneous</td>
<td>AsynchError</td>
</tr>
<tr>
<td>69, 37, …</td>
<td>Sets physical cursor mode system-wide.</td>
<td>Config.NLS.Locales</td>
<td>PhysicalCursor</td>
</tr>
<tr>
<td>69, 40, …</td>
<td>Sets end-of-file handling for sequential files system-wide.</td>
<td>Config.Miscellaneous</td>
<td>SetZEOF</td>
</tr>
<tr>
<td>69, 42, …</td>
<td>Sets $JOB format default system-wide.</td>
<td>Config.Miscellaneous</td>
<td>NodeNameInPid</td>
</tr>
<tr>
<td>69, 43, …</td>
<td>Sets clearing of global vectors system-wide.</td>
<td>Config.Miscellaneous</td>
<td>OldZU5</td>
</tr>
<tr>
<td>69, 44, …</td>
<td>Sets use of the Nagle algorithm for Telnet transmissions system-wide.</td>
<td>Config.Miscellaneous</td>
<td>UseNagleAlgorithm</td>
</tr>
<tr>
<td>69, 45, …</td>
<td>Truncates numbers during string-to-number conversion system-wide.</td>
<td>Config.Miscellaneous</td>
<td>TruncateOverflow</td>
</tr>
<tr>
<td>69, 49, …</td>
<td>Sets logging of transaction rollbacks system-wide.</td>
<td>Config.Miscellaneous</td>
<td>LogRollback</td>
</tr>
<tr>
<td>69, 51, …</td>
<td>Sets namespace default directory assignment behavior system-wide.</td>
<td>Config.Miscellaneous</td>
<td>SwitchOSdir</td>
</tr>
<tr>
<td>69, 60, …</td>
<td>Sets handling of asynchronous Telnet disconnect errors system-wide.</td>
<td>Config.Miscellaneous</td>
<td>AsyncDisconnectErr</td>
</tr>
<tr>
<td>69, 63, …</td>
<td>Enables or disables lowercase “e” as scientific notation symbol system-wide.</td>
<td>Config.Miscellaneous</td>
<td>ScientificNotation</td>
</tr>
<tr>
<td>69, 66, …</td>
<td>Suppress Telnet NUL at end-of-line system-wide.</td>
<td>Config.Miscellaneous</td>
<td>TelnetNUL</td>
</tr>
<tr>
<td>69, 68, …</td>
<td>Enables or disables the encryption of journal files system-wide.</td>
<td>Security.System</td>
<td>DBEncJournal</td>
</tr>
<tr>
<td>69, 69, …</td>
<td>Enables or disables the use of long strings system-wide.</td>
<td>Config.Miscellaneous</td>
<td>EnableLongStrings</td>
</tr>
</tbody>
</table>
### The $ZUTIL Replacement List

<table>
<thead>
<tr>
<th>$ZUTIL Function</th>
<th>Description</th>
<th>Class</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>69, 70, …</td>
<td>Enables or disables $DOUBLE returning INF and NAN values system-wide.</td>
<td>Config.Miscellaneous</td>
<td>IEEEError</td>
</tr>
<tr>
<td>69, 71, …</td>
<td>Sets IP address format system-wide.</td>
<td>Config.Startup</td>
<td>IPv6</td>
</tr>
<tr>
<td>69, 72, …</td>
<td>Sets MVBasic handling of undefined variables system-wide.</td>
<td>Config.Miscellaneous</td>
<td>MVDefined</td>
</tr>
</tbody>
</table>

**Note:** The $ZUTIL function descriptions excised from this reference have been removed to their own book. It can be found in the Legacy Documentation archive as *ObjectScript $ZUTIL Reference*, Version 2010.1. There are no plans to update this legacy volume.
ObjectScript Special Variables

Special variables are variables that are maintained by the system. They are also referred to as system variables, but are here referred to as special variables to avoid confusion with structured system variables.

Special variable names begin with a dollar sign ($). They can be distinguished from functions because they are not followed by parentheses and take no parameters. Special variable names are not case-sensitive. Many special variable names can be abbreviated. In the Synopsis for each special variable, the full name syntax is first presented, and below it is shown the abbreviated name (if one exists).

Historically, special variables have held scalar values. The system automatically updates these special variables to reflect various aspects of the operating environment. For example, the $IO special variable contains the ID of the current device. The $JOB special variable contains the ID of the current job.

Although you can set some special variables, most are read-only. With the exception of this read-only constraint, you can treat the special variables just as you would any other variable. For example, you can reference a special variable in an expression and assign its current value to another (user-defined) variable.

Any implementation-specific special variable form is marked with the abbreviation of the platform that supports it (Windows or UNIX®). Any form that is not marked with a platform abbreviation is supported by all platforms.

Special variables are listed in alphabetical order.
$DEVICE

Contains user-specified device status information.

The value of $DEVICE can be used to record device status information. You can use the SET command to place a value in $DEVICE. By convention, this value should describe the outcome of an I/O operation as a 3-piece string, in the form:

standard_error,user_error,explanatory_text

By default, $DEVICE contains the null string.

See Also

- SET command
$ECODE

Contains the current error code string.

$ECODE
$EC

Description

When an error occurs, Caché sets the $ECODE special variable to a comma-surrounded string containing the error code corresponding to the error. For example, when a reference is made to an undefined global variable, Caché sets the $ECODE special variable to the following string:

, M7,

$ECODE can contain ISO 11756-1999 standard M error codes, with the form M#, where # is an integer. For example, M6 and M7 are “undefined local variable” and “undefined global variable,” respectively. (M7 is issued for both globals and process-private globals.) For a complete list, see ISO 11756-1999 standard M programming language error messages in the Caché Error Reference.

$ECODE can also contain error codes that are the same as Caché General System error codes (the error codes returned at the terminal prompt and to the $ZERROR special variable). However, $ECODE prepends a “Z” to these error codes, and removes the angle brackets. Thus the $ECODE error ZSYNTAX is a <SYNTAX> error, ZILLEGAL VALUE is an <ILLEGAL VALUE> error, and ZFUNCTION is a <FUNCTION> error. $ECODE does not retain any additional error info for those error codes that provide it; thus ZPROTECT is a <PROTECT> error; the additional info component is kept in $ZERROR, but not in $ECODE. For more information about Caché error codes, see $ZERROR; for a complete list, see General System Error Messages in the Caché Error Reference.

If an error occurs when $ECODE already contains previous error codes, the existing error stack is cleared when the new error occurs. The new error stack will contain only entries that show the state at the time of the current error. (This is a change from earlier $ECODE behavior, where the old error stack would persist until explicitly cleared.)

If there are multiple error codes, Caché appends the code for each error, in the order received, at the end of the current $ECODE value. Each error in the resulting $ECODE string is delimited by commas, as follows:

, ZSTORE, M6, ZILLEGAL VALUE, ZPROTECT,

In the above case, the most recent error is a <PROTECT> error.

You can also explicitly clear or set $ECODE. $ECODE is always cleared when you terminate the current process.

Clearing $ECODE

You can clear $ECODE by setting it to the empty string (""), as follows:

```
SET $ECODE=""
```

Setting $ECODE to the empty string has the following effects:

- It clears all existing $ECODE values. It has no effect on an existing $ZERROR value.
- It clears the error stack for your job. This means that a subsequent call to the $STACK function returns the current execution stack, rather than the last error stack.
- It affects error processing flow of control for $ETRAP error handlers. See Error Processing in Using Caché ObjectScript for more details.

You cannot NEW the $ECODE special variable. Attempting to do so generates a <SYNTAX> error.
Setting $ECODE

You can force an error by setting $ECODE to an value other than the empty string. Setting $ECODE to any non-null value forces an interpreter error during the execution of an ObjectScript routine. After Caché sets $ECODE to the non-null value that you specify, Caché takes the following steps:

1. Writes the specified value to $ECODE, overwriting any previous values.
2. Generates an <ECODETRAP> error. (This sets $ZERROR to the value <ECODETRAP>.
3. Passes control to any error handlers you have established. Your error handlers can check for the $ECODE string value you chose and take steps to handle the condition appropriately.

$ECODE String Overflow

If the length of the accumulated string in $ECODE exceeds 512 characters, the error code that causes the string overflow clears and replaces the current list of error codes in $ECODE. In this case, the list of errors in $ECODE is the list of errors since the most recent string overflow, beginning with the error that caused the overflow. See Using Caché ObjectScript for more information about the maximum string data length.

Notes

Creating Your Own Error Codes

The format for the $ECODE special variable is a comma-surrounded list of one or more error codes. Error codes starting with the letter U are reserved for the user. All other error codes are reserved for Caché.

User–defined $ECODE values should be distinguishable from the values Caché automatically generates. To ensure this, always prefix your error text with the letter U. Also remember to delineate your error code with commas. For example:

    SET $ECODE=",Upassword expired!,"

Check $ZERROR Rather Than $ECODE for Caché Errors

Your error handlers should check $ZERROR rather than $ECODE for the most recent Caché error.

See Also

- ZTRAP command
- $STACK function
- $ESTACK special variable
- $ZEOF special variable
- $ZERROR special variable
- $ZTRAP special variable
- Error Handling in Using Caché ObjectScript
- System Error Messages in Caché Error Reference
$ESTACK

Contains the number of context frames saved on the call stack from a user-defined point.

Description

$ESTACK contains the number of context frames saved on the call stack for your job from a user-defined point. You specify this point by creating a new copy of $ESTACK using the NEW command.

The $ESTACK special variable is similar to the $STACK special variable. Both contain the number of context frames currently saved on the call stack for your job or process. Caché increments and restores both when changing context. The major difference is that you can reset the $ESTACK count to zero at any point by using the NEW command. You cannot reset the $STACK count.

Context Frames and Call Stacks

When a Caché image is started, before any contexts have been saved on the call stack, the values of both $ESTACK and $STACK are zero. Each time a routine calls another routine with DO, the system saves the context of the currently executing routine on the call stack, increments $ESTACK and $STACK, and starts execution of the called routine in the newly created context. The called routine can, in turn, call another routine, and so on. Each time another routine is called, Caché increments $ESTACK and $STACK and places more saved contexts on the call stack.

Issuing a DO command, an XECUTE command, or a call to a user-defined function establishes a new execution context. Issuing a GOTO command does not.

As DO commands, XECUTE commands, or user-defined function references create new contexts, Caché increments the values of $STACK and $ESTACK. As QUIT commands cause contexts to exit, Caché restores the previous contexts from the call stack and decrements the values of $STACK and $ESTACK.

The $ESTACK and $STACK special variables cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error.

Creating a New $ESTACK

You can create a new copy of $ESTACK in any context by using the NEW command. Caché takes the following actions:

1. Saves the old copy of $ESTACK.
2. Creates a new copy of $ESTACK with a value of zero (0).

In this way, you can establish a particular context as the $ESTACK level 0 context. As you create new contexts with DO, XECUTE, or user-defined functions, Caché increments this $ESTACK value. However, when you exit the context in which the new $ESTACK was created ($ESTACK at level 0), Caché restores the value of the previous copy of $ESTACK.

Examples

The following example shows the effect of a NEW command on $ESTACK. In this example, the MainRoutine displays the initial values of $STACK and $ESTACK (which are the same value). It then calls Sub1. This call increments $STACK and $ESTACK. The NEW command creates a $ESTACK with a value of 0. Sub1 calls Sub2, incrementing $STACK and $ESTACK. Returning to MainRoutine restores the initial values of $STACK and $ESTACK:
The following example demonstrates how the value of $ESTACK is incremented as new contexts are created by issuing DO and XECUTE commands, and decremented as these contexts are exited. It also shows that a GOTO command does not create a new context or increment $ESTACK:

```
Main
    NEW $ESTACK
    WRITE !,"Initial Main: $ESTACK=",$ESTACK, "$ESTACK=",$ESTACK
    DO Sub1
    WRITE !,"Return Main: $ESTACK=",$ESTACK
    GOTO Sub2
    Sub1
    WRITE !,"Sub1Call: $ESTACK=",$ESTACK, "$ESTACK=",$ESTACK
    NEW $ESTACK
    WRITE !,"Sub1NEW: $ESTACK=",$ESTACK, "$ESTACK=",$ESTACK
    DO Sub2
    QUIT
    Sub2
    WRITE !,"Sub2Call: $ESTACK=",$ESTACK, "$ESTACK=",$ESTACK
    GOTO Sub1Return
```

```
The following example demonstrates how the value of $ESTACK is incremented as new contexts are created by issuing DO and XECUTE commands, and decremented as these contexts are exited. It also shows that a GOTO command does not create a new context or increment $ESTACK:

Main
    NEW $ESTACK
    WRITE !,"Initial Main: $ESTACK=",$ESTACK, "$ESTACK=",$ESTACK
    DO Sub1
    WRITE !,"Return Main: $ESTACK=",$ESTACK
    XECUTE "WRITE !,"Sub1 XECUTE: $ESTACK=",$ESTACK"
    WRITE !,"Sub1 post-XECUTE: $ESTACK=",$ESTACK
    GOTO Sub2
    Sub1
    WRITE !,"Sub1 via DO: $ESTACK=",$ESTACK
    XECUTE "WRITE !,"Sub1 XECUTE: $ESTACK=",$ESTACK"
    WRITE !,"Sub1 via GOTO: $ESTACK=",$ESTACK
    GOTO Sub2
    Sub2
    WRITE !,"Sub2 via GOTO: $ESTACK=",$ESTACK
    GOTO Sub1Return
```

```
Notes

**Context Levels from the Terminal Prompt**

A routine invoked from a program starts at a different context level than a routine you invoke with the DO command from the Terminal prompt. When you enter a DO command at the Terminal prompt, the system creates a new context for the routine called.

The routine you call can compensate by establishing a $ESTACK level 0 context and then use $ESTACK for all context-level references.

Consider the following routine:

```
START
    ; Establish a $ESTACK Level 0 Context
    NEW $ESTACK
    ; Display the $STACK context level
    WRITE !,"$STACK level in routine START is ",$STACK
    ; Display the $ESTACK context level and exit
    WRITE !,"$ESTACK level in routine START is ",$ESTACK
    QUIT
```

When you run START from a program, you see the following display:

```
$STACK level in routine START is 0
$ESTACK level in routine START is 0
```

When you run START by issuing DO ^START at the Terminal prompt, you see the following display:

```
$STACK level in routine START is 1
$ESTACK level in routine START is 0
```
$ESTACK and Error Processing

$ESTACK is particularly useful during error processing when error handlers must unwind the call stack to a specific context level. See Error Handling in Using Caché ObjectScript for more information about error processing.

See Also

- $STACK function
- $STACK special variable
- Error Handling in Using Caché ObjectScript
$ETRAP

Contains a string of ObjectScript commands to be executed when an error occurs.

**Description**

$ETRAP contains a string that specifies one or more ObjectScript commands that are executed when an error occurs.

**Note:** $ETRAP is the least desirable of the available ObjectScript error handling facilities. Its use is discouraged. (See “Caché Error Handling Facilities” below.)

You use the SET command to give $ETRAP the value of a string that contains one or more ObjectScript commands. Then, when an error occurs, Caché executes the commands you entered into $ETRAP. For example, suppose you set $ETRAP to a string that contains a GOTO command to transfer control to an error-handling routine:

```ObjectScript
SET $ETRAP="GOTO LOGERR^ERRROU"
```

Caché then executes this command in $ETRAP immediately following any ObjectScript command that generates an error condition. Caché executes the SETRAP command at the same context level in which the error condition occurs. Caché saves the $ROLES value in effect when $ETRAP is set; when the $ETRAP code is executed, Caché sets $ROLES to that saved value. This prevents the $ETRAP error handler from using elevated privileges that were granted to the routine after establishing the error handler.

When setting $ETRAP to execute an error handler (for example, with a GOTO command) you can specify the error handler as `label` (a label in the current routine), `^routine` (the beginning of a specified external routine), or `label^routine` (a specified label in a specified external routine).

$ETRAP supports `label+offset` in some contexts (but not in procedures). This optional `+offset` is an integer specifying the number of lines to offset from `label`. InterSystems recommends that you avoid the use of a line offset when specifying an error handler location.

The maximum length of a SET $ETRAP string is 32,741 characters. Setting $ETRAP to a longer string may result in a <MAXSTRING> error.

**$ETRAP Commands Compared with XECUTE Commands**

The commands in a $ETRAP string are not executed in a new context level, unlike the commands in an XECUTE string. In addition, the $ETRAP command string is always terminated by an implicit QUIT command. The implicit QUIT command quits with a null-string argument when the $ETRAP error-handling commands are invoked in a user-defined function context where an argumented QUIT command is required.

**Setting $ETRAP Values in Different Context Levels**

By default, Caché carries the value of the $ETRAP special variable forward into new DO, XECUTE, and user-defined function contexts. However, you can create a new copy of $ETRAP in a context by issuing the NEW command, as follows:

```ObjectScript
NEW $ETRAP
```

Whenever you issue a NEW for $ETRAP, Caché performs the following actions:

1. Saves the copy of $ETRAP that was in use at that point.
2. Creates a new copy of $ETRAP.
3. Assigns the new copy of $ETRAP the same value as the old, saved copy of $ETRAP.
You then use the `SET` command to assign a different value to the new copy of `SETRAP`. In this way, you can establish new `SETRAP` error-handling commands for the current context.

You can also clear `SETRAP` by setting it to the null string. Caché then executes no `SETRAP` commands at the context level in the event of an error.

When a `QUIT` command causes the current context to be exited, Caché restores the old, saved value of `SETRAP`.

### Example

The following example demonstrates how the value of `SETRAP` is carried forward into new contexts and how you can invoke `SETRAP` error-handling commands again in each context after an error occurs. The `SETRAP` commands in this example make no attempt to dismiss the error. Rather, control by default is passed back to `SETRAP` error-handling commands at each previous context level.

The sample code is as follows:

```objectscript
ETR
NEW $ETRAP
SET $ETRAP="WRITE !,""$ETRAP invoked at Context Level ",$STACK"
; Initiate an XECUTE context that initiates a DO context
XECUTE "DO A"
QUIT
; Initiate a user-defined function context
A
SET A=$$B
QUIT
; A User-defined function that generates an error
B()
QUIT 1
```

A sample session using this code might run as follows:

```objectscript
>DO ^ETR
$ETRAP invoked at context level 4
$ETRAP invoked at context level 3
$ETRAP invoked at context level 2
$ETRAP invoked at context level 1
</COMMAND>
```

### Notes

**Use NEW Before Setting `SETRAP` to a New Value**

If you assign a new value to `SETRAP` in a context without first creating a new copy of `SETRAP` with the `NEW` command, Caché establishes that new value as the value of `SETRAP` not only for the current context but also for all previous contexts. Therefore, InterSystems **strongly recommends** that you use the `NEW SETRAP` command to create a new copy of `SETRAP` before you set `SETRAP` with a new value.

**`SETRAP` Value is a Line of ObjectScript Code**

Because the string value of `SETRAP` is executable ObjectScript commands, the length of the string cannot be longer than the maximum length of an ObjectScript routine line. See Using Caché ObjectScript for more information.

**Caché Error Handling Facilities**

The `SETRAP` special variable is one of several ObjectScript language facilities that enable you to control the handling and logging of errors that occur in your applications.

- The preferred Caché features for error handling are the block-structured `TRY` and `CATCH` commands.
- The `$ZTRAP` special variable is preferable to `SETRAP`.
- `SETRAP` will continue to be a supported feature of Caché. However, use of `SETRAP` in new code should generally be avoided in preference to the other error handling facilities.
See Error Handling in Using Caché ObjectScript for more information about error handling.

**$ETRAP and $ZTRAP**

When you set an error handler using $ZTRAP, this handler takes precedence over any existing $ETRAP error handler. Caché implicitly performs a NEW $ETRAP command and sets $ETRAP to the null string ("").

**$ETRAP and TRY / CATCH**

The TRY and CATCH commands perform error handling within an execution level. When an exception occurs within a TRY block, Caché normally executes the CATCH block of exception handler code that immediately follows the TRY block.

**Note:** Use of $ETRAP within a program structured with TRY blocks is strongly discouraged.

You cannot set $ETRAP within a TRY block. Attempting to do so generates a compilation error. You can set $ETRAP prior to the TRY block, or within the CATCH block.

If $ETRAP was previously set and an exception occurs in a TRY block, Caché may take $ETRAP rather than CATCH unless you forestall this possibility. If both $ETRAP and CATCH are present when an exception occurs, Caché executes the error code (CATCH or $ETRAP) that applies to the current execution level. Because $ETRAP is intrinsically not associated with an execution level, Caché assumes that it is associated with the current execution level unless you specify otherwise. You must NEW $ETRAP before setting $ETRAP to establish a level marker for $ETRAP, so that Caché will correctly take CATCH as the current level exception handler, rather than $ETRAP. Otherwise, a system error (including a system error thrown by the THROW command) may take the $ETRAP exception handler.

An exception that occurs within a CATCH block is handled by the current error trap handler.

**See Also**

- NEW command
- SET command
- THROW command
- TRY command
- $ECODE special variable
- $ZEOF special variable
- $ZTRAP special variable
- Error Handling in Using Caché ObjectScript
$HALT

Contains a halt trap routine call.

Description

$HALT contains the name of the current halt trap routine. A halt trap routine is called by your application when a HALT command is encountered. This halt trap routine may perform clean up or logging processing before issuing a HALT command, or it may substitute other processing rather than halting program execution.

You set $HALT to a halt trap routine using the SET command. The halt trap routine is specified by a quoted string with the following format:

```
SET $HALT=location
```

Here location can be specified as label (a label in the current routine or procedure), ^routine (the beginning of a specified external routine), or label^routine (a specified label in a specified external routine).

$HALT supports label+offset in some contexts (but not in procedures). This optional +offset is an integer specifying the number of lines to offset from label. InterSystems recommends that you avoid the use of a line offset when specifying location.

You cannot specify an +offset when calling a procedure or a CACHESYS % routine. If you attempt to do so, Caché issues a <NOLINE> error.

$HALT defines a halt trap routine for the current context. If there is already a halt trap defined for the current context, the new one replaces it. If you specify a nonexistent routine name, a HALT command ignores that $HALT and unwinds the stack to locate a valid $HALT at a previous context level.

To remove the halt trap for the current context, set $HALT to a null string. Attempting to remove a halt trap by using the NEW or KILL commands results in a <SYNTAX> error.

Halt Trap Execution

When you issue a HALT command, Caché checks the current context for $HALT. If no $HALT is defined for the current context (or it is set to a nonexistent routine name or the null string), Caché unwinds the stack to the previous context and looks for $HALT there. This process continues until either a defined $HALT is located or the stack is completely unwound. Caché uses the value of $HALT to transfer execution to the specified halt trap routine. The halt trap routine executes in the context at which $HALT was defined. No error code is set or error message issued.

If no valid $HALT is set in the current context or previous contexts, issuing a HALT command completely unwinds the stack and performs an actual program halt.

Commonly, a halt trap routine performs some cleanup or reporting processing, and then issues a HALT command. Note that with $HALT defined, the original HALT command invokes the halt trap, but does not perform an actual program halt. For an actual halt to occur, the halt trap routine must contain a second HALT command.

A HALT command issued by a halt trap routine is not trapped by that halt trap, but it may be trapped by a halt trap established at a lower context level. Thus a cascading series of halt traps may be invoked by a single HALT command.

Similar processing is performed by the error trap command ZTRAP, and the associated SZTRAP and SETRAP special variables.

$HALT and ^%ZSTOP

If you have $HALT set and also have code defined for ^%ZSTOP when a HALT is issued, the $HALT is executed first. $HALT can prevent the termination of the process, if its halt trap routine does not contain a HALT command.
A ^%ZSTOP routine is executed when the process is actually terminating. For further details on ^%ZSTOP, see the section on “Using the Caché ^%ZSTART and ^%ZSTOP Routines” in Caché Specialized System Tools and Utilities.

Examples

The following example uses $HALT to establish a halt trap:

```
SET $HALT="MyTrap^CleanupRoutine"
WRITE !,"the halt trap is: ",$HALT
```

Note that it is the programmer’s responsibility to make sure that the specified routine exists.

The following example shows how the halt trap routine executes in the context at which $HALT was defined. In this example, $HALT is defined at $ESTACK level 0, HALT is issued at $ESTACK level 1, and the halt trap routine executes at $ESTACK level 0.

Main
```
NEW $ESTACK
SET $HALT="OnHalt"
WRITE !,"Main $ESTACK= ",$ESTACK," $HALT= ",$HALT // 0
DO SubA
WRITE !,"Returned from SubA" // not executed
QUIT
```

SubA
```
WRITE !,"SubA $ESTACK= ",$ESTACK," $HALT= ",$HALT // 1
HALT
WRITE !,"this should never display"
QUIT
```

OnHalt
```
WRITE !,"OnHalt $ESTACK= ",$ESTACK // 0
HALT
QUIT
```

The following example is identical to the previous example, except that $HALT is defined at $ESTACK level 1. A HALT command is issued at $ESTACK level 1, and the halt trap routine executes at $ESTACK level 1. The HALT issued by the halt trap routine unwinds the stack, and, failing to find a $HALT defined at the previous context level, it halts program execution. Thus, the WRITE command following the DO command is not executed.

Main
```
NEW $ESTACK
WRITE !,"Main $ESTACK= ",$ESTACK," $HALT= ",$HALT // 0
DO SubA
WRITE !,"Returned from SubA" // not executed
QUIT
```

SubA
```
WRITE !,"SubA $ESTACK= ",$ESTACK," $HALT= ",$HALT // 1
HALT
WRITE !,"this should never display"
QUIT
```

OnHalt
```
WRITE !,"OnHalt $ESTACK= ",$ESTACK // 0
HALT
QUIT
```

The following example shows how a cascading series of halt traps can be invoked. Halt trap Halt0 is defined at $ESTACK level 0, and halt trap Halt1 is defined at $ESTACK level 1. The HALT command is issued at $ESTACK level 2. Caché unwinds the stack to invoke the halt trap Halt1 at $ESTACK level 1. This halt trap issues a HALT command; Caché unwinds the stack to invoke the halt trap Halt0 at $ESTACK level 0. This halt trap issues a HALT command that halts program execution.

Main
```
NEW $ESTACK
SET $HALT="Halt0"
WRITE !,"Main $ESTACK= ",$ESTACK," $HALT= ",$HALT // 0
DO SubA
WRITE !,"Returned from SubA" // not executed
QUIT
```

SubA
```
SET $HALT="Halt1"
WRITE !,"SubA $ESTACK= ",$ESTACK," $HALT= ",$HALT // 1
DO SubB
```

SubB
```
WRITE !,"SubB $ESTACK= ",$ESTACK," $HALT= ",$HALT // 2
```

Caché ObjectScript Reference
WRITE !,"Returned from SubA"  // not executed
QUIT

SubB
WRITE !,"SubB $ESTACK= "$ESTACK," $HALT= "$HALT  // 2
HALT
WRITE !,"this should never display"
QUIT

Halt0
WRITE !,"Halt0 $ESTACK= "$ESTACK  // 0
WRITE !,"Bye-bye!"
HALT
QUIT

Halt1
WRITE !,"Halt1 $ESTACK= "$ESTACK  // 1
HALT
QUIT

See Also

•  HALT command
$HOROLOG

Contains the local date and time for the current process.

Description

$HOROLOG contains the date and time for the current process. It can contain the following values:

- The current local date and time.
- The current local date and time, adjusted for a different time zone offset.
- A user-specified non-incrementing date. Time continues to be the current local time.

$HOROLOG contains a character string that consists of two integer values, separated by a comma. These two integers represent the current local date and time in Caché storage format. These integers are counters, not user-readable dates and times. $HOROLOG returns the current date and time in the following format:

```
dddddd,sssss
```

The first integer, dddddd, is the current date expressed as a count of the number of days since December 31, 1840, where day 1 is January 1, 1841. Because Caché represents dates using a counter from an arbitrary starting point, Caché is unaffected by the Year 2000 boundary. The maximum value for this date integer is 2980013, which corresponds to December 31, 9999.

The second integer, sssss, is the current time, expressed as a count of the number of seconds since midnight of the current day. The system increments the time field from 0 to 86399 seconds. When it reaches 86399 at midnight, the system resets the time field to 0 and increments the date field by 1. $HOROLOG truncates fractional seconds; it represents time in whole seconds only.

You can obtain the same current date and time information by invoking the Horolog() method, as follows:

```
WRITE $SYSTEM.SYS.Horolog()
```

Refer to %SYSTEM.SYS in the InterSystems Class Reference for further details.

Separating Date and Time

To get just the date portion or just the time portion of $HOROLOG, you can use the $PIECE function, specifying the comma as the delimiter character:

```
SET dateint=$PIECE($HOROLOG","1)
SET timeint=$PIECE($HOROLOG","2)
WRITE ",Date and time: ",$HOROLOG
WRITE ",Date only: ",dateint
WRITE ",Time only: ",timeint
```

To get just the date portion of a $HOROLOG value, you can also use the following programming trick:

```
SET dateint=+$HOROLOG
WRITE ",Date and time: ",$HOROLOG
WRITE ",Date only: ",dateint
```

The plus sign (+) causes Caché to parse the $HOROLOG string as a number. When Caché encounters a nonnumeric character (the comma), it truncates the rest of the string and returns the numeric portion. This is the date integer portion of the string.
Date and Time Functions Compared

The various ways to return the current date and time are compared, as follows:

- **$HOROLOG** contains the local, variant-adjusted date and time in Caché storage format. The local time zone is determined from the current value of the **$ZTIMEZONE** special variable, and then adjusted for local time variants, such as Daylight Saving Time. It returns whole seconds only; fractions of a second are truncated.

- **$NOW** returns the local date and time for the current process. **$NOW** returns the date and time in Caché storage format. It includes fractional seconds; the number of fractional digits is the maximum precision supported by the current operating system.
  - **$NOW()** determines the local time zone from the value of the **$ZTIMEZONE** special variable. The local time is not adjusted for local time variants, such as Daylight Saving Time. It therefore may not correspond to local clock time.
  - **$NOW(tzmins)** returns the time and date that correspond to the specified **tzmins** time zone parameter. The value of **$ZTIMEZONE** is ignored.

- **$ZTIMESTAMP** contains the UTC (Coordinated Universal Time) date and time, with fractional seconds, in Caché storage format. Fractional seconds are expressed in three digits of precision (on Windows systems), or six digits of precision (on UNIX® systems).

Date and Time Conversions

You can use the **$ZDATE** function to convert the date portion of **$HOROLOG** into external, user-readable form. You can use the **$ZTIME** function to convert the time portion of **$HOROLOG** into external user-readable form. You can use the **$ZDATETIME** function to convert both the date and time. When using **$HOROLOG**, setting the *precision* for time values in these functions always returns zeros as fractional seconds.

You can use the **$ZDATEH** function to convert a user-readable date into the date portion of **$HOROLOG**. You can use the **$ZTIMEH** function to convert a user-readable time into the time portion of **$HOROLOG**. You can use the **$ZDATETIMEH** function to convert both the date and time to a **$HOROLOG** value.

Setting the Date and Time

**$HOROLOG** can be set to a user-specified date for the current process using the **FixedDate()** method of the **%SYSTEM.Process** class. **$HOROLOG** cannot be modified using the **SET** command. Attempting to do so results in a *<SYNTAX>* error.

```objectscript
DO ##class(%SYSTEM.Process).FixedDate(12345)  // set $HOROLOG date
WRITE ',,$ZDATETIME($HOROLOG,1,1,9)," $HOROLOG changed date"
WRITE ',,$ZDATETIME($NOW(),1,1,9)," $NOW() no date change"
WRITE ',,$ZDATETIME($ZDATETIMEH($ZTIMESTAMP,-3),1,1,9)," $ZTS UTC-to-local",
   " no date change"
DO ##class(%SYSTEM.Process).FixedDate(0)    // restore $HOROLOG
WRITE ',,$ZDATETIME($HOROLOG,1,1,9)," $HOROLOG current date"
```

Note that **FixedDate()** changes the **$HOROLOG** value, but not the **$NOW** or **$ZTIMESTAMP** value.

Time Zone

By default, **$HOROLOG** contains the date and time for the local time zone. This time zone default is supplied by the operating system, which Caché uses to set the **$ZTIMEZONE** default.

Changing **$ZTIMEZONE** affects the value of **$HOROLOG** for the current process. It changes the time portion of **$HOROLOG**, and this change of time can also change the date portion of **$HOROLOG**. **$ZTIMEZONE** is a fixed offset of time zones from the Greenwich meridian; it does not adjust for local seasonal time variants, such as Daylight Saving Time.
Daylight Saving Time

$HOROLOG adjusts for seasonal time variants based on the algorithm supplied by the underlying operating system. After applying the $ZTIMEZONE value, Caché uses the operating system local time to adjust $HOROLOG (if needed) for seasonal time variants, such as Daylight Saving Time.

You can determine if Daylight Saving Time is in effect for the current date, or for a specified date and time using the IsDST() method. The following example returns the Daylight Saving Time (DST) status for the current date and time.

Because this status could change while the program is running, this example checks it twice:

```
CHECKDST
SET x=$SYSTEM.Util.IsDST()
SET local=$ZDATETIME($HOROLOG)
SET x2=$SYSTEM.Util.IsDST()
GOTO:x'=x2 CHECKDST
ELSEIF x=0 {WRITE local," DST not in effect"}
ELSE {WRITE local," DST setting cannot be determined"}
```

The application of seasonal time variants may differ based on (at least) three considerations:

- Operating system: Within a time zone, $HOROLOG for a given date may differ on different computers. This is because different operating systems use different algorithms to apply time variants. Because policies governing the beginning and end dates for Daylight Saving Time (and other time variants) have changed, older operating systems may not reflect current practice, and/or calculations using older $HOROLOG values may be adjusted using the current beginning and end dates, rather than the ones in force at that time.

- Government policies have changed over time: There have been numerous changes to seasonal time variants since their first adoption in 1916 (much of Europe) and 1918 (United States). Daylight Saving Time has been adopted, rejected, and re-adopted by governmental policies in many places. The seasonal start and end dates for Daylight Saving Time have also changed numerous times. In the United States, recent changes of national policy have occurred in 1966, 1974–75, 1987, and 2007. Adoption of, or exemption from, national policies have also occurred due to local legislative actions. For example, the state of Arizona does not observe Daylight Saving Time.

- Geography: Daylight Saving Time is summer time; the local clock shifts forwards (“Spring ahead”) at the start of DST and shifts backwards (“Fall back”) at the end of DST. Thus the calendar start and end dates for Daylight Saving Time within the same time zone are commonly reversed in the northern hemisphere and the southern hemisphere. Equatorial nations and most of Asia and Africa do not observe Daylight Saving Time.

Local Time Variant Thresholds

$HOROLOG calculates the number of seconds from midnight by consulting the system clock. Therefore, if the system clock is automatically reset when crossing a local time variant threshold, such as the beginning or end of Daylight Saving Time, the time value of $HOROLOG also shifts abruptly ahead or back by the appropriate number of seconds. For this reason, comparisons of two $HOROLOG time values may yield unanticipated results if the period between the two values includes a local time variant threshold.

$NOW does not adjust for local time variants. Its use may be preferable when comparing date and time values if the period between the two values includes a local time variant threshold.

Dates Before 1840

$HOROLOG cannot be directly used to represent dates outside of the range of years 1840 through 9999. However, you can represent historic dates far beyond this range using the Caché SQL Julian date feature. Julian dates can represent a date as an unsigned integer, counting from 4711 BC (BCE). Julian dates do not have a time-of-day component.

You can convert a Caché $HOROLOG date to a Caché Julian date using the TO_CHAR SQL function, or the TOCHAR() method of the %SYSTEM.SQL class. You can convert a Caché Julian date to a Caché $HOROLOG date using the TO_DATE SQL function, or the TODATE() method of the %SYSTEM.SQL class.
The following example takes the current $HOROLOG date and converts it to a Julian date. The + before $HOROLOG forces Caché to treat it as a number, and thus truncate at the comma, eliminating the time integer:

```
WRITE !,"Horolog date = ",+$H
SET x=$SYSTEM.SQL.TOCHAR(+$HOROLOG,"J")
WRITE !,"Julian date = ",x
```

The following example takes a Julian date and converts it to a Caché $HOROLOG date:

```
SET x=$SYSTEM.SQL.TODATE(2455030,"J")
WRITE !,"$HOROLOG date = ",x," = ", $ZDATE(x,1)
```

Note that Julian date values smaller than 1721100 cannot be converted; an <ILLEGAL VALUE> error is generated.

For further information on Julian dates, refer to TO_DATE and TO_CHAR in the Caché SQL Reference.

For information on the starting date of $HOROLOG, see the section “Historical Note.”

**Examples**

The following example displays the current contents of $HOROLOG.

```
WRITE $HOROLOG
```

This returns a value formatted like this: 62210,49170

The following example uses $ZDATE to convert the date field in $HOROLOG to a date format.

```
WRITE $ZDATE($PIECE($HOROLOG,",",1))
```

returns a value formatted like this: 04/29/2011

The following example converts the time portion of $HOROLOG to a time in the form of hours:minutes:seconds on a 12-hour (a.m. or p.m.) clock.

```
CLOCKTIME
NEW
SET Time=$PIECE($HOROLOG,"",2)
SET Sec=Time#60
SET Min=Totmin#60
SET Milhour=Totmin\60
IF Milhour=12 { SET Hour=12,Meridian=" pm" } 
ELSEIF Milhour>12 { SET Hour=Milhour-12,Meridian=" pm" } 
ELSE { SET Hour=Milhour,Meridian=" am" } 
WRITE !,Hour," :",Min," :",Sec,Meridian
QUIT
```

**Historical Note**

In the “Just Ask!” column of the September 1993 issue of “M Computing”, a publication of the M Technology Association, Silver Spring, MD 20903, James M. Poitras explained the choice of starting date for $HOROLOG:

"Starting in early 1969, our group created the Chemistry Lab application at Massachusetts General Hospital (MGH), which was the first package in the MGH MUMPS with Global Data Storage and many of the features of the language today..."

"When we started programming, there were no utility programs of any type. We had to write them all: time, date, verify database, global tally, print routine, etc. I ended up writing initial versions of most of these.

"When I decided on specifications for the date routine, I remembered reading of the oldest (one of the oldest?) U.S. citizen, a Civil War veteran, who was 121 years old at the time. Since I wanted to be able to represent dates in a Julian-type form so that age could be easily calculated and to be able to represent any birth date in the numeric range selected, I decided that a starting date in the early 1840s would be 'safe'. Since my algorithm worked most logically when every fourth year..."
was a leap year, the first year was taken as 1841. The zero point
was then December 30, 1840...

"That's the origin of December 31, 1840 or January 1, 1841. I
wasn't party to the MDC (M Development Committee) negotiations,
but I did explain the logic of my choice to members of the
Committee."

See Also

- $NOW function
- $ZDATE function
- $ZDATEH function
- $ZDATETIME function
- $ZDATETIMEH function
- $ZTIME function
- $ZTIMEH function
- $ZTIMESTAMP special variable
- $ZTIMEZONE special variable
$IO

Contains the ID of the current input/output device.

Description

$IO contains the device ID of the current device to which all input/output operations are directed. If the input and output devices are different, $IO contains the ID of the current input device.

Caché sets the value of $IO to the principal input/output device at login. $PRINCIPAL contains the ID of the principal device. You issue a USE command to change the current device. Only the USE and CLOSE commands, a BREAK command, or a return to the programmer prompt can change this value.

You can return the device type of the current device by using the GetType() method of the %Library.Device class.

On UNIX® systems, $IO contains the actual device name.

On Windows systems, $IO contains a Caché-generated unique identifier for the principal device. For terminal devices (TRM or TNT), this consists of a pseudo-device name enclosed in vertical bars, a colon and another vertical bar, followed by the device’s process ID (pid) number. For non-terminal devices, the pseudo-device name is enclosed in vertical bars and followed by a unique numeric identifier.

For a Terminal: |TRM|:|pid
For a Telnet terminal: |TNT|nodename:portnumber|pid
For a file descriptor: |FD|file_descriptor_number
(File descriptors are used with CALLIN/CALLOUT remote access.)
For a TCP device: |TCP|unique_device_identifier
For a named pipe: |NPIPE|unique_device_identifier
For the default printer: |PRN|
For a printer other than the default: |PRN|physical_device_name

If the principal device is a null device (which is the default for a background process), $IO contains the null device name with "|pid" appended, thus allowing you to use $IO for a unique subscript. The null device name contained in $IO depends on the operating system.

• For Windows systems, $IO contains //./nul:pid
• For UNIX® systems, $IO contains /dev/null:pid

If the input device is redirected via a pipe or file, $IO contains “00”.

The default device number for a device is configurable. Go to the Management Portal, select [System] > [Configuration] > [Device Settings] > [Devices]. For the desired device, click “Edit” to display and modify its Physical Device Name: option. If you do this, $IO will contain the assigned device number, rather than the actual operating system device name.

This special variable cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error.

See Also

• USE command
• $PRINCIPAL special variable
ObjectScript Special Variables

- I/O Devices and Commands in *Caché I/O Device Guide*
- Terminal I/O in *Caché I/O Device Guide*
Contains the ID of the current process.

Description

JOB contains the ID number of the current process. This ID number is the host operating system’s actual Process ID (PID). This ID number is unique for each process.

The format of the string returned to JOB is determined for the current process by the setting of the NodeNameInPid() method of the %SYSTEM.Process class. The system-wide default behavior can be established by setting the NodeNameInPid property of the Config.Miscellaneous class. By default, JOB returns only the PID, but you can set these functions to have JOB return both the PID and the node name. For example: 11284:MYCOMPUTER.

This special variable cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error.

To establish the PID as the terminal prompt, use the TerminalPrompt(5) method of the %SYSTEM.Process class.

Other Information About the Current Process

You can obtain the same current process ID number by invoking the ProcessId() method, as follows:

WRITE $SYSTEM.SYS.ProcessID()

Refer to the %SYSTEM.SYS class in the InterSystems Class Reference for further details.

You can use JOB to obtain the job number for the current process as follows:

SET JobObj-##CLASS(%SYS.ProcessQuery).%OpenId($JOB)
WRITE JobObj.JobNumber

Refer to the %SYS.ProcessQuery class in the InterSystems Class Reference for further details.

You can obtain status information about the current process from the $ZJOB special variable.

You can obtain the PID of the child process or the parent process of the current process from the $ZCHILD and $ZPARENT special variables.

You can obtain the PIDs of the current jobs in the job table from the ^$JOB structured system variable.

See Also

- JOB command
$KEY

Contains the terminator character from the most recent READ.

$KEY

Description

$KEY contains the character or character sequence that terminated the last READ command on the current device. $KEY and $ZB are very similar in function; see below for a detailed comparison.

- If the last read terminated because of a terminator character (such as the <RETURN> key), $KEY contains the terminator character.
- If the last read terminated because of a timeout or a fixed-length read length limit, $KEY contains the null string. No terminator character was encountered.
- If the last read was a single-character read (READ *), and a character was entered, $KEY contains the actual input character.

$KEY and $ZB are very similar, though not identical. See below for a comparison.

You can use the SET command to specify a value for $KEY. You can use the ZZDUMP command to display the value of $KEY.

During a terminal session, the end of every command line is recorded in $KEY as a carriage return (hexadecimal 0D). In addition, the $KEY special variable is initialized to carriage return by the process that initializes the terminal session. Therefore, to display the value of $KEY set by the READ command or a SET command during a terminal session, you must copy the $KEY value to a local variable within the same line of code.

Examples

In the following example, a variable-length READ command either receives data from the terminal or times out after 10 seconds. If the user inputs the data before the timeout, $KEY contains the user-input carriage return (hex 0D) that terminated the data input. If, however, the READ timed out, $KEY contains the null string, indicating that no terminator character was received.

```
READ "Ready or Not: ",x:10
ZZDUMP $KEY
```

In the following example, a fixed-length READ command either receives data from the terminal or times out after 10 seconds. If the user inputs the specified number of characters (in this case, one character), the user does not have to press <RETURN> to conclude the READ operation. The user can respond to the read prompt by pressing <RETURN> rather than entering the specified number of characters.

If the read operation timed out, both $KEY and $ZB contain the null string. If the user inputs a one-character middle initial, $KEY contains the null string, because the fixed-length READ operation concluded without a terminator character. If the user pressed <RETURN> rather than entering a middle initial, $KEY contains the user-input carriage return.

```
READ "Middle initial: ",z#1:10
IF $ASCII($ZB)=-1 {
   WRITE !,"The read timed out" }
ELSEIF $ASCII($KEY)=-1 {
   WRITE !,"A character was entered" }
ELSEIF $ASCII($KEY)=13 {
   WRITE !,"A line return was entered" }
ELSE {
   WRITE !,"Unexpected result" }
```
Notes

$KEY and $ZB Compared

Both $KEY and $ZB contain the character that terminates a READ operation. These two special variables are similar, but not identical. Here are the principal differences:

- $KEY can be set using the SET command. $ZB cannot be SET.
- Following a successful fixed-length READ, $ZB contains the final character input (for example, when the 5-digit postal code “02138” is input as a fixed-length READ, $ZB contains “8”). Following a successful fixed-length READ, $KEY contains the null string (“”).
- $KEY does not support block-based read and write operations, such as magnetic tape I/O.

$KEY on the Command Line

When issuing commands interactively from the Terminal command line, you press <RETURN> to issue each command line. The $KEY and $ZB special variables record this command line terminator character. Therefore, when using $KEY or $ZB to return the termination status of a read operation, you must set a variable as part of the same command line.

For example, if you issue the command:

```
>READ x:10
```

from the command line, then check $KEY, it will not contain the results of the read operation; it will contain the <RETURN> character that executed the command line. To return the results of the read operation, set a local variable with $KEY in the same command line, as follows:

```
>READ x:10 SET rkey=$KEY
```

This preserves the value of $KEY set by the read operation. To display this read operation value, issue either of the following command line statements:

```
>WRITE $ASCII(rkey)
```

; returns -1 for null string (time out)

; returns ASCII decimal value for terminator character

```
>ZZDUMP rkey
```

; returns blank line for null string (time out)

; returns hexadecimal value for terminator character

See Also

- READ command
- SET command
- ZZDUMP command
- $ZB special variable
$NAMESPACE

Contains the namespace for the current stack level.

```
$NAMESPACE
SET $NAMESPACE=namespace
NEW $NAMESPACE
```

**Description**

$NAMESPACE contains the name of the current namespace for the current stack level. You can use $NAMESPACE to:

- Return the name of the current namespace.
- Change the current namespace with SET.
- Establish a new temporary namespace context with NEW and SET.

**Return the Current Namespace Name**

The $NAMESPACE special variable contains the current namespace name.

You can also obtain the name of the current namespace by invoking the NameSpace() method of %SYSTEM.SYS class, as follows:

```
WRITE $SYSTEM.SYS.NameSpace()
```

You can obtain the full pathname of the current namespace by using the NormalizeDirectory() method of %Library.File class, as follows:

```
WRITE $NAMESPACE,
WRITE ##class(%Library.File).NormalizeDirectory(""
```

You can test whether a namespace is defined by using the Exists() method of the %SYS.Namespace class, as follows:

```
WRITE ##class(%SYS.Namespace).Exists("USER"),!  ; an existing namespace
WRITE ##class(%SYS.Namespace).Exists("LOSER")   ; a non-existent namespace
```

These methods are described in the InterSystems Class Reference.

**SET $NAMESPACE**

You can set $NAMESPACE to an existing namespace using the SET command. In SET $NAMESPACE=namespace, specify namespace as a string literal or a variable or expression that evaluates to a quoted string; namespace is not case-sensitive. However, Caché always displays explicit namespace names in all uppercase letters, and implied namespace names in all lowercase letters. A namespace name can contain Unicode letter characters; Caché converts accented lowercase letters to their corresponding accented uppercase letters.

The namespace name can be an explicit namespace name ("USER") or an implied namespace ("^^c:\InterSystems\Cache\mgr\user"). For further details on implied namespaces, refer to the ZNSPACE command.

If the specified namespace does not exist, SET $NAMESPACE generates a <NAMESPACE> error. If you do not have access privileges to a namespace, the system generates a <PROTECT> error, followed by the database path. For example, the %Developer role does not have access privileges to the %SYS namespace. If you have this role and attempt to access this namespace, Caché issues the following error (on a Windows system): <PROTECT>

```
*c:\intersystems\cache\mgr\.
```

When you wish to temporarily change the current namespace, perform some operation, then revert to the prior namespace, use SET $NAMESPACE, rather than the SET $ZNSPACE or the ZNSPACE command. This is because $NAMESPACE permits you to NEW $NAMESPACE.
**NEW $NAMESPACE**

By setting $NAMESPACE you can change the current namespace. This is the preferred way to change a namespace in a method or other routine. By using NEW $NAMESPACE and SET $NAMESPACE you establish a namespace context that automatically reverts to the prior namespace when the method concludes or an unexpected error occurs:

```objectscript
TRY {
    WRITE "before the method: ", $NAMESPACE, !
    DO MyNSMethod("DocBook")
    WRITE "after the method: ", $NAMESPACE
    RETURN
}
```

NextMethod(ns)
NEW $NAMESPACE
IF ##class(%SYS.Namespace).Exists(ns) {
    SET $NAMESPACE=ns }
ELSE {SET $NAMESPACE="User" }
WRITE "namespace changed in method: ", $NAMESPACE, !
SET num=5/$RANDOM(2)
QUIT
NextMethod()
WRITE "This should not write", !
}
CATCH exp {
    WRITE "namespace after error in method: ", $NAMESPACE, !
    IF 1=exp.%IsA("%Exception.SystemException") {
        WRITE "System exception: ", $ZCVT(exp.Name,"O","HTML"), !
    }
}
```

Quitting a routine or branching to an error trap reverts to this stacked namespace. If you create an object instance in the changed namespace, Caché closes the object in that namespace before reverting to the stacked namespace. This is shown in the following Terminal example:

```terminal
USER>NEW $NAMESPACE
USER 1S1>NEW myoref
SAMPLES 2N1>SET $NAMESPACE="SAMPLES"
SAMPLES 2N1>SET myoref=##class(%SQL.Statement).%New()
SAMPLES 2N1>QUIT
/* Cache closes myoref in the SAMPLES namespace
    Then reverts to the USER namespace */
USER>
```

In more complex stacked namespace situations, it is the programmer’s responsibility to explicitly close objects in the proper namespace.

**Examples**

The following example calls a routine that executes in a different namespace than the calling program. It uses NEW $NAMESPACE to stack the current namespace. It then uses SET $NAMESPACE to change the namespace for the duration of Test. The QUIT reverts to the stacked namespace:

```objectscript
WRITE "before: ", $NAMESPACE, !
DO Test
WRITE "after: ", $NAMESPACE, !
QUIT
Test
NEW $NAMESPACE
SET $NAMESPACE="USER"
WRITE "testing: ", $NAMESPACE, !
; routine code
QUIT
```

There is no need to handle an error to switch back to the old namespace; Caché restores the old namespace when you leave the current stack level.

The following example differs from the previous example by omitting NEW $NAMESPACE. Note that upon QUIT the namespace does not revert:
WRITE "before: ",$NAMESPACE,!
DO Test
WRITE "after: ",$NAMESPACE,!
QUIT
Test
NEW
SET $NAMESPACE="USER"
WRITE "testing: ",$NAMESPACE,!
; routine code
QUIT

Calling a separate routine when temporarily changing the current namespace is the preferred programming practice. In situations when calling a separate routine is not practical, you can use the legacy DO command dot syntax. The following example temporarily changes the namespace within a large subroutine by using this DO command syntax to create a stack frame:

WRITE "before: ",$NAMESPACE,!
DO
  . NEW $NAMESPACE
  . SET $NAMESPACE="USER"
  . WRITE "testing: ",$NAMESPACE,!
  . ; routine code
  WRITE "after: ",$NAMESPACE,!

See Also

• NEW command
• SET command
• ZNSPACE command
• $ZNSPACE special variable
• Configuring Namespaces in Caché System Administration Guide
$PRINCIPAL

Contains the ID of the principal I/O device.

Description
$PRINCIPAL contains the ID of the principal I/O device for the current process. $PRINCIPAL operates like $IO. Refer to $IO for details of specific device types and system platforms.

If the principal device is closed, $PRINCIPAL does not change. If the principal input and output devices differ, $PRINCIPAL reflects the ID of the principal input device.

This special variable cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error.

Examples
This example uses $PRINCIPAL to test for a principal device.

    IF $PIECE($PRINCIPAL,"|",4) { WRITE "Principal device is: ",$PRINCIPAL } ELSE { WRITE "Undefined" }

This example uses and writes to the principal device.

    USE $PRINCIPAL
    WRITE "output to $PRINCIPAL"

Notes

$PRINCIPAL and USE 0

$PRINCIPAL is functionally equivalent to the widely used, but nonstandard, USE 0. Use $PRINCIPAL instead of USE 0 because it is standard, and because it makes your code more flexible.

See Also

• USE command
• $IO special variable
Contains a flag indicating what kind of QUIT is required to exit the current context.

Description

$QUIT contains a value that indicates whether an argumented QUIT command is required to exit from the current context. If an argumented QUIT is required to exit from the current context, $QUIT contains a one (1). If an argumented QUIT is not required to exit from the current context, $QUIT contains a zero (0).

In a context created by issuing a DO or XECUTE command, an argumented QUIT is not required to exit. In a context created by a user-defined function, an argumented QUIT is required to exit.

This special variable cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error.

Example

The following example demonstrates $QUIT values in a DO context, in an XECUTE context, and in a user-defined function context.

The sample code is as follows:

QUI
DO
  . WRITE !,"$QUIT in a DO context = ",$QUIT
  . QUIT
XECUTE "WRITE !,"$QUIT in an XECUTE context = ",$QUIT"
  SET A=$$A
  QUIT
  A()
  WRITE !,"$QUIT in a User-defined function context = ",$QUIT
  QUIT 1

A sample session using this code might run as follows:

USER> DO ^QUI
$QUIT in a DO context = 0
$QUIT in an XECUTE context = 0
$QUIT in a User-defined function context = 1

Notes

$QUIT and Error Processing

The $QUIT special variable is particularly useful during error processing when the same error handler can be invoked at context levels that require an argumented QUIT and at context levels that require an argumentless QUIT.

See the Error Handling in Using ObjectScript for more information about error processing.

See Also

- DO command
- QUIT command
- XECUTE command
$ROLES

Contains the roles assigned to the current process.

Description

$ROLES contains the list of roles assigned to the current process. This list of roles consists of a comma-separated string that can contain both User Roles and Added Roles.

A role is assigned to a user either by using the SQL GRANT statement, or by using the Management Portal System Administration, Security, Users option. Select a user name to edit its definition, then select the Roles tab to assign that user to a role. A role can be defined using the SQL CREATE ROLE statement and deleted using the SQL DROP ROLE statement. A role must be defined before it can be assigned to a user. A role can be revoked from a user using the SQL REVOKE statement.

When a process is created using the JOB command, it inherits the same $ROLES and $USERNAME values as its parent process.

When a process performs I/O redirection, this redirection is performed using the user’s login $ROLES value, not the current $ROLES value.

Roles Granted to Roles Not Listed

Granting a role to another role is a concept only available through Caché SQL. Roles granted to roles are used in SQL to determine the user's role list for checking SQL privileges. They cannot be accessed by ObjectScript. You cannot grant a role to another role through Caché System Security. Therefore, the $ROLES special variable list does not contain any roles that SQL operations have granted to the current roles. For further details, refer to the GRANT command in the Caché SQL Reference.

SET $ROLES

You can use the SET command to change the Added Roles part of the list contained in $ROLES. Setting $ROLES only alters a process' Added Roles. It cannot alter its User Roles. To set $ROLES to a different list of Added Roles is a restricted system capability. However, such restrictions do not apply to setting $ROLES to a null string, which deletes the list of Added Roles.

A role must be defined before it can be added. You can define a role using the SQL CREATE ROLE command. CREATE ROLE does not give any privileges to a role. To assign privileges to a role use either the SQL GRANT statement, or the Management Portal System Administration, Security, Roles interface.

You must issue a NEW $ROLES statement before escalating the process roles using SET $ROLES.

NEW $ROLES

NEW $ROLES stacks the current values of both $ROLES and $USERNAME. You can use the NEW command on $ROLES without security restrictions.

Issue a NEW $ROLES and then SET $ROLES to supply Added Roles. You can then create an object instance that uses these Added Roles. If you quit this routine, Caché closes the object with the Added Roles before reverting to the stacked $ROLES value.

Examples

The following example returns the list of roles for the current process.

WRITE $ROLES
The following example first creates the roles Vendor, Sales, and Contractor. It then displays the comma-separated list of default roles (which contain both User Roles and Added Roles). The first `SET $ROLES` replaces the list of Added Roles with the two roles Sales and Contractor. The second `SET $ROLES` concatenates the Vendor role to the list of Added Roles. The final `SET $ROLES` sets the Added Roles list to the null string, removing all Added Roles. The User Roles remain unchanged throughout:

```
CreateRoles
  &sql(CREATE ROLE Vendor)
  &sql(CREATE ROLE Sales)
  &sql(CREATE ROLE Contractor)
  IF SQLCODE=0 {
    WRITE !,"Created new roles"
    DO SetRoles }
  ELSEIF SQLCODE=-118 {
    WRITE !,"Role already exists"
    DO SetRoles }
  ELSE { WRITE !,"CREATE ROLE failed, SQLCODE="$,SQLCODE }
SetRoles()
  WRITE !,"Initial: ",$ROLES
  NEW $ROLES
  SET $ROLES="Sales,Contractor"
  WRITE !,"Replaced: ",$ROLES
  NEW $ROLES
  SET $ROLES=$ROLES_","Vendor"
  WRITE !,"Concatenated: ",$ROLES
  SET $ROLES=""
  WRITE !,"Nulled: ",$ROLES
```

See Also

- ObjectScript: `SET` command `NEW` command `$USERNAME` special variable
- Caché SQL: `CREATE ROLE DROP ROLE GRANT REVOKE %CHECKPRIV`
### $STACK

Contains the number of context frames saved on the call stack.

<table>
<thead>
<tr>
<th>$STACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ST</td>
</tr>
</tbody>
</table>

### Description

$STACK contains the number for context frames currently saved on the call stack for your process. You can also look at $STACK as the zero-based context level number of the currently executing context. Therefore, when a Caché job is started, before any contexts have been saved on the call stack, the value of $STACK is zero (0).

Each time a routine calls another routine with a DO command, the context of the currently executing routine is saved on the call stack and execution starts in the newly created context of the called routine. The called routine can, in turn, call another routine and so on. Each additional call causes another saved context to be placed on the call stack.

An XECUTE command and a user-defined function reference also establish a new execution context. A GOTO command does not.

As new contexts are created by DO commands, XECUTE commands, or user-defined function references, the value of $STACK is incremented. As contexts are exited with the QUIT command, previous context are restored from the call stack and the value of $STACK is decremented.

This special variable cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error. $ESTACK is identical to $STACK, except that you can establish a $ESTACK level of 0 (zero) at any point by issuing a NEW $ESTACK command. You cannot NEW the $STACK special variable.

### Error Handling

When an error occurs, all context information is immediately saved on your process error stack. This changes the value of $STACK. The context information is then accessible using the $STACK function until the value of $ECODE is cleared by an error handler. In other words, while the value of $ECODE is non-null, the $STACK function returns information about a context saved on the error stack rather than an active context at the same specified context level.

### Context Levels from the Terminal Prompt

A routine that is invoked from a program starts at a different context level than a routine invoked from the Terminal prompt with a DO command. The DO command typed at the Terminal prompt causes a new context to be created. The following example shows the routine START invoked from a routine or from the Terminal prompt:

Consider the following routine:

```
START
    ; Display the context level and exit
    WRITE !,"Context level in routine START is ",$STACK
QUIT
```

When you run START from a program, you see the following display:

```
Context level in routine START is 0
```

When you run START by issuing DO ^START at the Terminal prompt, you see the following display:

```
Context level in routine START is 1
```

### Examples

The following example demonstrates how the value of $STACK is incremented as new contexts are create and decremented as contexts are exited.
The sample code is as follows:

STA
  WRITE !,"Context level in routine STA = ",$STACK
  DO A
  WRITE !,"Context level after routine A = ",$STACK
  QUIT
A
  WRITE !,"Context level in routine A = ",$STACK
  DO B
  WRITE !,"Context level after routine B = ",$STACK
  QUIT
B
  WRITE !,"Context level in routine B = ",$STACK
  XECUTE "WRITE !,"Context level in XECUTE = ",$STACK"
  WRITE !,"Context level after XECUTE = ",$STACK
  QUIT

A sample session using this code might run as follows:

USER>DO ^STA
  Context level in routine STA = 1
  Context level in routine A = 2
  Context level in routine B = 3
  Context level after XECUTE = 4
  Context level after routine B = 2
  Context level after routine A = 1

See Also

- $STACK function
- $ESTACK special variable
- Error Handling in Using Caché ObjectScript
$STORAGE

Contains the number of bytes available for local variable storage.

<table>
<thead>
<tr>
<th>$STORAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S</td>
</tr>
</tbody>
</table>

**Description**

$STORAGE returns the number of bytes available for local variable storage in the current process partition. The initial value of $STORAGE is established by the value of $ZSTORAGE, the maximum amount of memory available to the process. The larger the $ZSTORAGE value (in kilobytes), the larger the $STORAGE value (in bytes). However, this relationship between $ZSTORAGE and $STORAGE is not a simple 1:1 ratio.

The $STORAGE value is affected by the following operations:

- $STORAGE decreases as local variables are defined in the local variable space, for example, by using the SET command. The decrease in $STORAGE corresponds to the amount of space required to store the value of the local variable; the size of the name of the local variable has no effect on $STORAGE, but the number of subscript levels does affect $STORAGE. The $STORAGE value increases as local variables are removed, for example, by using the KILL command.

- $STORAGE decreases when you issue a NEW command. NEW establishes a new execution level; space set aside for local variables (whether or not used) at the previous execution level is not available at the new execution level. The initial NEW decreases $STORAGE by approximately 15000; each subsequent NEW decreases $STORAGE by 12288. The $STORAGE value increases when you issue a QUIT command to exit an execution level.

- $STORAGE decreases when you define a flow-of-control statement, such as IF or FOR, or a block structure such as TRY and CATCH. Storage is allocated to compile these structures, not to execute them. Therefore, a FOR statement consumes the same amount of storage regardless whether it loops or how many times it loops; each IF, ELSEIF, and ELSE clause consumes a set amount of storage, regardless of how many branches are executed. The space is allocated from the process that compiled the code. Note that a FOR loop commonly defines a local variable as a counter.

The $STORAGE value is not affected by setting process-private variables, global variables, or special variables. The $STORAGE value is not affected by changing namespaces. The $STORAGE value is not affected by enabling long strings because long string storage is not allocated in the process partition. For further details, refer to Long Strings in the “Data Types and Values” chapter of Using Caché ObjectScript.

The $STORAGE special variable cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error.

**Low Memory and <STORE> Errors**

The $STORAGE value may be a positive or negative number. A value of zero does not indicate no available storage, but indicates that storage is in extremely short supply. If $STORAGE decreases to less than zero, at some point a <STORE> error occurs. For example, if $STORAGE decreases to -7000, allocating storage for another local variable might fail due to a <STORE> error, indicating insufficient available storage space to store a local variable value, or to establish a new execution level.

The first <STORE> error occurs when $STORAGE is some value less than zero; the exact negative $STORAGE value threshold depends upon context. This <STORE> error indicates that you must get additional storage, either by increasing $ZSTORAGE, or by freeing some allocated storage through KILL or QUIT operations. When this first <STORE> error occurs, the system automatically makes 1Mb of additional memory available to the process to enable error processing and recovery. Caché does not change $ZSTORAGE; it allows $STORAGE to go further into negative number values.

When this first <STORE> error occurs, Caché internally designates the process as being in a low memory state. While in this low memory state the process may continue to allocate memory and the value of $STORAGE may continue to decrease.
into lower negative numbers. While in this low memory state the process may free some allocated memory, causing the value of $STORAGE to rise. Thus, the value of $STORAGE may rise or fall within a range of values without issuing additional <STORE> errors. Also, after the first <STORE> error you may see a small rise in $STORAGE caused by Caché freeing some internal memory.

This first <STORE> error provides some memory cushion that allows your process to call diagnostics, perform saves to disk, exit gracefully, free memory, and continue.

A process remains in a low memory state until either of the following occurs:

- The process makes available sufficient memory. Your process can do this by increasing the $ZSTORAGE allocation, and/or by freeing allocated storage through KILL or QUIT operations. When the value of $STORAGE exceeds 256K (or 25% of $ZSTORAGE, whichever is smaller), Caché removes the process from low memory state. At that point the process can again issue a <STORE> error if the available memory decreases into negative numbers.

- The process consumes the additional memory. When the value of $STORAGE reaches -1048576, a second <STORE> error occurs. If your process arrives at this point, no more memory is available to the process and further process operations become unpredictable. It is likely the process will immediately terminate.

You can determine the reason for a <STORE> error by calling the $SYSTEM.Process.MemoryAutoExpandStatus() method.

**Examples**

The following example shows how $STORAGE becomes smaller when $ZSTORAGE is set to a smaller value. Note that the relationship (ratio) between these two values is variable:

```objectscript
SET $ZS=262144
FOR i=1:1:10 {
  WRITE "$ZS=",$ZS," $S=",$S," ratio=","$NORMALIZE($S/$ZS,3),!
  IF $ZS>30000 {SET $ZS=$ZS-30000 }
}
```

The following example shows how $STORAGE decreases as local variables are assigned, and increases when local variables are killed:

```objectscript
WRITE "$STORAGE=",$S," initial value",!
FOR i=1:1:30 {SET a(i)="abcdefghijklmnopqrstuvwxyz"
  WRITE "$STORAGE=",$S,! }
KILL a
WRITE !,"$STORAGE=",$S," after KILL",!
```

The following example shows how the number of subscript levels of an assigned local variable affect $STORAGE:

```objectscript
WRITE "No subscripts:",!
SET before=$S
SET a="abcdefghijklmnopqrstuvwxyz"
WRITE " memory allocated ",before-$S,!
KILL a
WRITE "One subscript level:",!
SET before=$S
SET a(1)="abcdefghijklmnopqrstuvwxyz"
WRITE " memory allocated ",before-$S,!
KILL a(1)
WRITE "Nine subscript levels:",!
SET before=$S
SET a(1,2,3,4,5,6,7,8,9)="abcdefghijklmnopqrstuvwxyz"
WRITE " memory allocated ",before-$S,!
KILL a(1,2,3,4,5,6,7,8,9)
```

The following example shows how $STORAGE decreases (becomes unavailable at that level) as NEW establishes a new execution level:

```objectscript
WRITE "increasing levels:",!
FOR i=1:1:10 {WRITE "$STORAGE=",$S,! NEW }
```
The following example shows how $STORAGE decreases as local variables are assigned until it enters low memory state, issuing a <STORE> error. The <STORE> error is caught by a CATCH block that invokes the StoreErrorReason() method to determine what caused the error. Note that entering the CATCH block consumes a significant amount of storage. Once in the CATCH block, this example allocates one more variable.

```
TRY {
    WRITE !,"TRY block",!
    SET init=$ZSTORAGE
    SET $ZSTORAGE=456
    WRITE "$Initial $STORAGE=",$STORAGE,!
    FOR i=1:1:1000 {
        SET pre=$STORAGE
        SET var(i)="1234567890abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
        IF $STORAGE<0 {WRITE "var(",i,") negative memory=",$STORAGE,!
        ELSEIF pre<$STORAGE {WRITE "var(",i,") new allocation $S=",$STORAGE,!
        ELSE {WRITE "var(",i,") $S=",$STORAGE,!
            }
        }
    }
    CATCH myexp {
        WRITE !,"CATCH block exception handler",!!
        WRITE "Name: ",$ZCVT(myexp.Name,"O","HTML"),!
        IF myexp.Name="<STORE>" {WRITE "store error reason=",
            $SYSTEM.Process.StoreErrorReason(),!
        WRITE "$S=",$STORAGE,!
        SET j=i
        SET var(j)="1234567890abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
        WRITE "var(",j,") added one more variable $S=",$STORAGE,!
        SET $ZSTORAGE=init
        RETURN
    }
}
```

**See Also**

- $ZSTORAGE special variable
- Local variables in the “Variables” chapter of *Using Caché ObjectScript*
## $SYSTEM

Contains system information about system objects.

### $SYSTEM Special Variable

$SYSTEM as a special variable contains the local system name and the name of the current instance of Caché, separated by a colon (:). The name of the machine follows the case conventions of the local operating system and the name of the instance is in uppercase. If Caché is part of the name of the instance, there is no accent on the final letter. For example:

```
MyComputer:CACHE2
```

You can also determine your local system name using the `LocalHostName()` method:

```ObjectScript
WRITE $SYSTEM,!
WRITE $SYSTEM.INetInfo.LocalHostName()
```

The abbreviation $SY can only be used for $SYSTEM as a special variable.

### $SYSTEM Class

$SYSTEM as a class provides access to a variety of system objects. You can invoke a method that returns information, or a method that performs some operation such as upgrading or loading and returns status information. Caché supports several classes of system objects, including the following:

- Version: for version numbers of Caché and its components
- SYS: for the system itself
- OBJ: for Objects
- SQL: for SQL queries
- CSP: for Caché Server Pages

Note that object class names and method names are case-sensitive. Specifying the wrong case for these names results in a `<CLASS DOES NOT EXIST>` or `<METHOD DOES NOT EXIST>` error. If you do not specify parentheses with the method name, it issues a `<SYNTAX>` error.

For further information on using dot syntax with $SYSTEM to access these objects, refer to the chapter “Working with Registered Objects” of Using Caché Objects. For further information on using %SYSTEM.OBJ, refer to Flags and Qualifiers. $SYSTEM can access the System API classes in the %SYSTEM class package, described in the InterSystems Class Reference documentation.

### Flags and Qualifiers

These are arguments which can be used to control the import of external sources into Caché, compile existing applications, and export these to external destinations. In the class documentation for %SYSTEM.OBJ, these are often supplied as the value of the parameter, `qspec`. The available settings for each can be displayed by the commands:

```ObjectScript
DO $SYSTEM.OBJ.ShowFlags()
```
and

DO $SYSTEM.OBJ.ShowQualifiers()

**Flags**

Of the two, the flags are the earlier. They were modeled on UNIX® command-line parameters and thus are one- or two-character sequences. The existing flags and their meanings (excluding deprecated flags) are:

**Table F–1: Existing Flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>Includes subclasses and classes that reference the current class in SQL usage.</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>Compiles the class definitions after loading.</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>Display. Flag set by default.</td>
<td>X</td>
</tr>
<tr>
<td>e</td>
<td>Deletes the extent definition that describes the global storage used by the extent, and deletes the data.</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>Shows hidden classes.</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>Validates XML export format against schema on Load. Flag set by default.</td>
<td>X</td>
</tr>
<tr>
<td>k</td>
<td>Keep source. When this flag is set, source code of generated routines will be kept.</td>
<td></td>
</tr>
<tr>
<td>l</td>
<td>Locks classes while compiling. Flag set by default.</td>
<td>X</td>
</tr>
<tr>
<td>p</td>
<td>Includes classes whose names begin with the “%” character.</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>Recursive. Compiles all the classes that are dependency predecessors.</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>System. Processes system messages or application messages.</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>Includes classes that are related to the current class; classes that either reference the current class in SQL usage, or are referenced by the current class in SQL usage.</td>
<td></td>
</tr>
<tr>
<td>o1, o2, o3, o4</td>
<td>Optimization specifiers. Deprecated and ignored by the class compiler.</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** Flags may be turned off by preceding them with a dash (-).

**Qualifiers**

During the development of Caché version 5.1, it became clear that flags would be insufficient to provide for the emerging needs of import, export and compilation. Rather than replace the flags mechanism, a newer, more extensible set of controls was implemented: qualifiers. To preserve backward compatibility, the flag mechanism remains fully supported. In addition, a qualifier exists whose meaning is the same as each existing flag, and the two may be used in the same specifier.

Since there are so many more qualifiers, they are organized into groups according to the function they control as shown in the following tables. Deprecated qualifiers are not shown.

You can also set these qualifiers for the current namespace (the default) or system-wide using the **SetQualifiers()** method.
<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>/autoinclude</td>
<td>Automatically includes any classes that are not up to date that are required to compile this class.</td>
<td>1</td>
</tr>
<tr>
<td>/checkschema</td>
<td>Validates imported XML files against the schema definition.</td>
<td>1</td>
</tr>
<tr>
<td>/checksysutd</td>
<td>Checks system classes for up-to-dateness.</td>
<td>0</td>
</tr>
<tr>
<td>/checkuptodate</td>
<td>Skips classes or expanded classes that are up-to-date.</td>
<td>expandedonly</td>
</tr>
<tr>
<td>/compile</td>
<td>Causes classes loaded to be compiled as well.</td>
<td>0</td>
</tr>
<tr>
<td>/cspcompileclass</td>
<td>Causes classes created by CSP or CSR load to be compiled.</td>
<td>1</td>
</tr>
<tr>
<td>/cspdeployclass</td>
<td>When CSP page loaded deploys the class generated.</td>
<td>0</td>
</tr>
<tr>
<td>/csphidden</td>
<td>Classes generated from CSP and CSR compilation are marked as hidden.</td>
<td>1</td>
</tr>
<tr>
<td>/defaultowner</td>
<td>When loading classes, if the classowner keyword is not defined, insert the user name specified in this string into the class definition as the class owner. If the value of this string is $USERNAME, insert the current user name into the class definition as the class owner.</td>
<td>—</td>
</tr>
<tr>
<td>/defines</td>
<td>Comma separated list of macros to define and, optionally, their values.</td>
<td>—</td>
</tr>
<tr>
<td>/deleteextent</td>
<td>Deletes the extent definition that describes the global storage used by the extent, and deletes the data.</td>
<td>0</td>
</tr>
<tr>
<td>/diffexport</td>
<td>Does not include any time or platform information in export so the files can be run through diff/merge tools.</td>
<td>0</td>
</tr>
<tr>
<td>/display</td>
<td>Alias qualifier for /displaylog and /displayerror.</td>
<td>—</td>
</tr>
<tr>
<td>/displayerror</td>
<td>Displays error information.</td>
<td>1</td>
</tr>
<tr>
<td>/displaylog</td>
<td>Displays log information.</td>
<td>1</td>
</tr>
<tr>
<td>/expand</td>
<td>Alias qualifier for /predecessorclasses, /subclasses and /relatedclasses.</td>
<td>—</td>
</tr>
<tr>
<td>/exportgenerated</td>
<td>When exporting classes also exports generated classes where the class generating them is also included.</td>
<td>0</td>
</tr>
<tr>
<td>/exportselectivity</td>
<td>Exports the selectivity values stored in the storage definition for this class.</td>
<td>1</td>
</tr>
<tr>
<td>/filterin</td>
<td>Alias qualifier for /application, /system and /percent.</td>
<td>—</td>
</tr>
<tr>
<td>/generated</td>
<td>Determines when expanding patterns or lists of classes in a package whether to include generated items (routines, classes, etc.).</td>
<td>1</td>
</tr>
<tr>
<td>/generatemap</td>
<td>Generates the map file.</td>
<td>1</td>
</tr>
<tr>
<td>Flag</td>
<td>Meaning</td>
<td>Default</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>/importselectivity</td>
<td>0: Do not import selectivity values from the XML file. 1: Import the selectivity values stored in the storage definition when importing XML file. 2: Keep the existing class selectivity values, but if the existing class does not have selectivity specified for something that is present in the XML file then use the selectivity value from the XML file.</td>
<td>2</td>
</tr>
<tr>
<td>/includesubpackages</td>
<td>Includes sub-packages.</td>
<td>1</td>
</tr>
<tr>
<td>/journal</td>
<td>Journaling enabled while performing a class compile. If the process performing the compile has specifically disabled journaling, /journal defaults to 0, rather than the system-wide default of 1.</td>
<td>1</td>
</tr>
<tr>
<td>/keepsource</td>
<td>Keeps the source code of generated routines.</td>
<td>0</td>
</tr>
<tr>
<td>/lock</td>
<td>Uses LOCK command while compiling classes.</td>
<td>1</td>
</tr>
<tr>
<td>/mapped</td>
<td>Includes classes mapped from another database. If you specifically ask to compile a class from another database (CompileList() method), the class will be compiled regardless of the /mapped setting. /mapped only applies when the code is searching for classes, for example, using the CompileAll() method.</td>
<td>0</td>
</tr>
<tr>
<td>/mergeglobal</td>
<td>If importing a global from XML file merges the global with existing data.</td>
<td>0</td>
</tr>
<tr>
<td>/multicompiled</td>
<td>Enables multiple users’ jobs to compile classes.</td>
<td>1</td>
</tr>
<tr>
<td>/percent</td>
<td>Includes percent classes.</td>
<td>0</td>
</tr>
<tr>
<td>/predecessorclasses</td>
<td>Recursively includes dependency predecessor classes.</td>
<td>0</td>
</tr>
<tr>
<td>/relatedclasses</td>
<td>Recursively includes related classes.</td>
<td>0</td>
</tr>
<tr>
<td>/retainstorage</td>
<td>When a class is compiled, the compiler generates a storage definition. By default, if the storage definition is updated the class definition is updated with the updated storage definition. If a new version of the class is loaded from an external source, that updated storage definition is overwritten by whatever is defined in the new version of the class definition. If the new version of the class does not include a storage definition then the existing storage definition is removed. Setting /retainstorage saves the existing storage definition temporarily and restores it after the new version of a class is loaded. If the new version of the class also defines the storage definition, the existing storage definition is overwritten and not retained. If the new version of the class does not define the storage definition, the previous version of the storage definition is restored.</td>
<td>0</td>
</tr>
<tr>
<td>/subclasses</td>
<td>Recursively includes sub-classes.</td>
<td>0</td>
</tr>
<tr>
<td>/system</td>
<td>Processes system messages or application messages.</td>
<td>0</td>
</tr>
</tbody>
</table>
### Table F–3: Export Qualifiers

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>/checksysutd</td>
<td>Checks system classes for up-to-dateness.</td>
<td>0</td>
</tr>
<tr>
<td>/checkuptodate</td>
<td>Checks if classes are up-to-date when projecting.</td>
<td>expandedonly</td>
</tr>
<tr>
<td>/createdirs</td>
<td>Creates directories if they do not exist.</td>
<td>0</td>
</tr>
<tr>
<td>/cspdeployclass</td>
<td>When CSP page loaded deploys the class generated.</td>
<td>0</td>
</tr>
<tr>
<td>/csphidden</td>
<td>Classes generated from CSP and CSR compilation are marked as hidden.</td>
<td>1</td>
</tr>
<tr>
<td>/diffexport</td>
<td>Does not include any time or platform information in export so the files can be run through diff/merge tools.</td>
<td>0</td>
</tr>
<tr>
<td>/display</td>
<td>Alias qualifier for /displaylog and /displayerror.</td>
<td>—</td>
</tr>
<tr>
<td>/displayerror</td>
<td>Displays error information.</td>
<td>1</td>
</tr>
<tr>
<td>/displaylog</td>
<td>Displays log information.</td>
<td>1</td>
</tr>
<tr>
<td>/documatichost</td>
<td>Host that is used in JavaDoc generation.</td>
<td>—</td>
</tr>
<tr>
<td>/documaticnamespace</td>
<td>Namespace that is used in JavaDoc generation.</td>
<td>—</td>
</tr>
<tr>
<td>/documaticport</td>
<td>Port that is used in JavaDoc generation.</td>
<td>—</td>
</tr>
<tr>
<td>/exportgenerated</td>
<td>When exporting classes also exports generated classes where the class generating them is also included.</td>
<td>0</td>
</tr>
<tr>
<td>/exportselectivity</td>
<td>Exports the selectivity values stored in the storage definition for this class.</td>
<td>1</td>
</tr>
<tr>
<td>/exportversion</td>
<td>Specifies the target Caché version to load this export as a three-part release version, for example 2012.2.2. Cache uses the /exportversion value to handle changes in the export format across versions of Caché by removing class keywords that were not implemented in the earlier Caché version. Specifying /exportversion does not guarantee compatibility of code between the exporting and importing systems.</td>
<td>The current version of Caché</td>
</tr>
<tr>
<td>/generatemap</td>
<td>Generates the map file.</td>
<td>1</td>
</tr>
<tr>
<td>/generationtype</td>
<td>Generation mode.</td>
<td>—</td>
</tr>
<tr>
<td>/genserialuid</td>
<td>Generates serialVersionUID.</td>
<td>1</td>
</tr>
<tr>
<td>/importselectivity</td>
<td>0: do not import selectivity values from the XML file; 1: import the selectivity values stored in the storage definition when importing XML file; 2: keep any existing selectivity values but if a property does not have an existing value then use the selectivity from the XML file.</td>
<td>2</td>
</tr>
<tr>
<td>/includesubpackages</td>
<td>Includes sub-packages.</td>
<td>1</td>
</tr>
<tr>
<td>/javadoc</td>
<td>Does not create javadoc.</td>
<td>1</td>
</tr>
<tr>
<td>/make</td>
<td>Only generates dependency or class if timestamp of last compilation is greater than timestamp of last generation.</td>
<td>0</td>
</tr>
</tbody>
</table>
### Table F–4: ShowClassAndObject Qualifiers

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>/mapped</td>
<td>Includes classes mapped from another database.</td>
<td>0</td>
</tr>
<tr>
<td>/mergeglobal</td>
<td>If importing a global from XML file merges the global with existing data.</td>
<td>0</td>
</tr>
<tr>
<td>/newcollections</td>
<td>Uses native Java collections.</td>
<td>1</td>
</tr>
<tr>
<td>/percent</td>
<td>Includes percent classes.</td>
<td>0</td>
</tr>
<tr>
<td>/pojo</td>
<td>POJO generation mode.</td>
<td>0</td>
</tr>
<tr>
<td>/predecessorclasses</td>
<td>Recursively includes dependency predecessor classes.</td>
<td>0</td>
</tr>
<tr>
<td>/primitivedatatypes</td>
<td>Uses Java primitives for %Integer, %Boolean, %BigInt, %Decimal.</td>
<td>0</td>
</tr>
<tr>
<td>/projectabstractstream</td>
<td>Projects classes that contain methods whose arguments are abstract streams or whose return type is an abstract stream.</td>
<td>0</td>
</tr>
<tr>
<td>/projectbyrefmethodstopojo</td>
<td>Projects byref methods to pojo implementation.</td>
<td>0</td>
</tr>
<tr>
<td>/recursive</td>
<td>Exports classes recursively.</td>
<td>1</td>
</tr>
<tr>
<td>/relatedclasses</td>
<td>Recursively includes related classes.</td>
<td>0</td>
</tr>
<tr>
<td>/skipstorage</td>
<td>Does not export the class storage information.</td>
<td>0</td>
</tr>
<tr>
<td>/subclasses</td>
<td>Recursively includes sub-classes.</td>
<td>0</td>
</tr>
<tr>
<td>/system</td>
<td>Processes system messages or application messages.</td>
<td>0</td>
</tr>
<tr>
<td>/unconditionallyproject</td>
<td>Projects regardless of problems that may prevent code from compiling or working correctly.</td>
<td>0</td>
</tr>
<tr>
<td>/usedeepestbase</td>
<td>Uses deepest base in which method or property is defined for method or property definition. If P is defined in A,B, and C and A extends B extends C, then C is a deeper base for P.</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table F–5: UnitTest

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>/autoload</td>
<td>Specifies the directory to be auto-loaded.</td>
<td>—</td>
</tr>
</tbody>
</table>
### Flag Qualifier Mapping

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>/cleanup</td>
<td>Cleans up globals upon completion of unit test. By default, globals are</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>not cleaned up. Even when set, DeepSee globals are not cleaned up.</td>
<td></td>
</tr>
<tr>
<td>/debug</td>
<td>Causes the Asserts to BREAK if they fail.</td>
<td>0</td>
</tr>
<tr>
<td>/delete</td>
<td>Determines if loaded classes should be deleted.</td>
<td>1</td>
</tr>
<tr>
<td>/display</td>
<td>Alias qualifier for /displaylog and /displayerror.</td>
<td>—</td>
</tr>
<tr>
<td>/displayerror</td>
<td>Displays error information.</td>
<td>1</td>
</tr>
<tr>
<td>/displaylog</td>
<td>Displays log information.</td>
<td>1</td>
</tr>
<tr>
<td>/findleakedvariables</td>
<td>When enabled, the public variables currently set in the process are recorded before a test is run, then compared with those set after the test is completed. Other than a predetermined set of known context and output variables, such as SQLCODE, any newly defined variables are reported, with their values, as a test failure.</td>
<td>0</td>
</tr>
<tr>
<td>/load</td>
<td>Determines if classes should be loaded; if not, then onlyclassnames are obtained from the directories.</td>
<td>1</td>
</tr>
<tr>
<td>/loadudl</td>
<td>Loads UDL files produced by Atelier. When set, loads .cls, .mac, .int, and .inc files. /loadudl and /loadxml can be used to limit what types of files are loaded; by default, all files are loaded. UDL files are always loaded as UTF8 so that Unicode characters are loaded correctly.</td>
<td>1</td>
</tr>
<tr>
<td>/loadxml</td>
<td>Loads XML files produced by Studio. When set, loads .xml files. /loadudl and /loadxml can be used to limit what types of files are loaded; by default, all files are loaded.</td>
<td>1</td>
</tr>
<tr>
<td>/recursive</td>
<td>Determines if tests in subdirectories should run recursively.</td>
<td>1</td>
</tr>
<tr>
<td>/run</td>
<td>Determines if tests should run.</td>
<td>1</td>
</tr>
</tbody>
</table>

These qualifiers are given in the qspec as they appear, for example, “/compile/displayerror/subclasses”. No spaces are allowed between qualifiers.

**Note:** Qualifiers may be negated by preceding the qualifier with “no” as in “/nodisplaylog”. Alternatively, the value of the qualifier can be specified explicitly as in “/displaylog=0”.

### Qualifiers For Flags

The following table gives the existing flag and the equivalent qualifier. Some flags map into multiple qualifiers, and also have different meanings when used for differing purposes.

**Table F–6: Flag Qualifier Mapping**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Group</th>
<th>Qualifier</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>Compiler</td>
<td>/subclasses</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>Compiler</td>
<td>/compile</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>Compiler</td>
<td>/displayerror</td>
<td>1</td>
</tr>
</tbody>
</table>
### Order Of Processing

The qspec is processed from left to right. The setting for a given flag or qualifier overrides the current setting whether it came from the environment defaults, or from an occurrence earlier in the qspec.

When both flags and qualifiers appear, the flags must be placed before (to the left of) the qualifiers. This means that qualifier settings always override any flag settings.

### Examples

The following is an example of using $SYSTEM to invoke a method that returns system information:

```objectscript
WRITE $SYSTEM.OBJ.Version()
```

returns a string such as the following:

```
Cache Objects Version 2017.2.0.307
```
Note that this objects version is not formatted the same as the system version number contained in the **$ZVERSION** special variable.

You can list all of the methods for the OBJ class as follows. (By changing the class name, you can use this method to get a list for any system class):

```objectscript
DO $SYSTEM.OBJ.Help()
```

To list information about just one method in a class, specify the method name in the Help argument list, as shown in the following example:

```objectscript
DO $SYSTEM.OBJ.Help("Load")
```

The following are a few more examples of **$SYSTEM** that invoke methods:

```objectscript
DO $SYSTEM.OBJ.Upgrade()
WRITE !,"* * * * * * * * * * * *
DO $SYSTEM.CSP.DisplayConfig()
WRITE !,"* * * * * * * * * * * *
WRITE !,$SYSTEM.Version.GetPlatform()
WRITE !,"* * * * * * * * * * * *
WRITE !,$SYSTEM.SYS.TimeStamp()
```

The following example calls the same methods as the previous example, using the **##class(%SYSTEM)** syntax form:

```objectscript
DO ##class(%SYSTEM.OBJ).Upgrade()
DO ##class(%SYSTEM.CSP).DisplayConfig()
WRITE !,##class(%SYSTEM.Version).GetPlatform()
WRITE !,##class(%SYSTEM.SYS).TimeStamp()
```

The previous two examples requires that UnknownUser have assigned the **%DB_CACHESYS** role.

**See Also**

- **$ISOBJECT** function
- **$ZVERSION** special variable
- Working with Registered Objects in *Using Caché Objects*
$TEST

Contains the truth value resulting from the last command using the timeout option.

Description

$TEST contains the truth value (1 or 0) resulting from the last command with a timeout. $TEST is set by the following commands, regardless of whether they are entered from the Terminal prompt or encountered in routine code:

- A timed JOB sets $TEST to 1 if the attempt to start the new job succeeds before the timeout expires. If the timeout expires, $TEST is set to 0.
- A timed LOCK sets $TEST to 1 if the lock attempt succeeds before the timeout expires. If the timeout expires, $TEST is set to 0.
- A timed OPEN sets $TEST to 1 if the open attempt succeeds before the timeout expires. If the timeout expires, $TEST is set to 0.
- A timed READ sets $TEST to 1 if the read completes before the timeout expires. If the timeout expires, $TEST is set to 0.

Issuing these commands without a timeout does not set $TEST.

Note: $TEST is also set by the legacy version of the IF command. It is neither set nor checked by the current block-structured IF command. When the test expression of a legacy IF command is evaluated, $TEST is set equal to the resulting truth value. In other words, if the IF expression tests true, $TEST is set to 1. If it tests false, $TEST is set to 0 (zero).

Setting $TEST

You can use the SET command to set $TEST to a boolean value. A value of 1, or any non-zero numeric value, sets $TEST=1. A value of 0, or a non-numeric string value, sets $TEST=0.

$TEST can be set by any command or function that can return a logical condition.

Maintaining $TEST

A successful JOB, LOCK, OPEN, or READ command that did not specify a timeout does not change the existing value of $TEST.

The DO command maintains the value of $TEST when calling a procedure, but not when calling a subroutine. For details, refer to the DO command.

The ZBREAK command maintains the value of $TEST when calling execute_code. For details, refer to the ZBREAK command.

Example

The following code performs a timed read and uses $TEST to test for completion of the read.

```caché-objects
READ !,"Type a letter: ",a#1:10
IF $TEST { DO Success(a) } ELSE { DO TimedOut }
Success(val)
WRITE !,"Received data: ",val
TimedOut()
WRITE !,"Timed out"
```
Notes

Operations That Do Not Set $TEST

JOB, LOCK, OPEN, and READ commands without a timeout have no effect on $TEST. Postconditional expressions also have no effect on $TEST.

The block-oriented IF command (which defines a block of code by enclosing it in curly braces) does not use $TEST in any way. The following invocations of the legacy IF command also do not use $TEST: legacy IF without an argument and the ELSE command have no effect on $TEST.

Unsuccessful Timed Operations

Caché does not produce an error message after an unsuccessful timed operation. Your application must check $TEST and then produce an appropriate message.

See Also

- JOB command
- LOCK command
- OPEN command
- READ command
- IF (legacy version) command
$THIS

Contains the current class context.

**Description**

$THIS contains the current class context. The class context for an instance method is the current object reference (OREF). The class context for a class method is the current classname as a string value. For example, if you issue the command DO ..method() or SET ..property = value from within a class method, the .. context is resolved using the current value of $THIS. When making a reference within an object instance, the relative dot syntax (..) is preferred.

$THIS is commonly used when you are within an object instance and you call a function that is on another object. In this circumstance, you can use $THIS to pass the current class context to that function, so that it can return a value to your current object instance.

When $THIS does not contain a valid object reference, Caché returns a <NO CURRENT OBJECT> error.

$THIS can be used in the contexts such as the following:

```
SET x = ##class(otherclassname).method($THIS)
DO ##class(superclass)$THIS.method(args)
```

This special variable cannot be set to a value using the SET command. Attempting to do so results in a <FUNCTION> error.

For further details, refer to “$this Syntax” in Object-specific ObjectScript Features chapter of Using Caché Objects. For information on OREFs, see “OREF Basics” in Using Caché Objects.

**See Also**

- $CLASSNAME function
$THROWOBJ

Contains the OREF from an unsuccessful THROW.

Description

$THROWOBJ contains the object reference (OREF) thrown by the most recent unsuccessful THROW operation. Caché writes an OREF to $THROWOBJ when it issues a <THROW> error. Commonly, this occurs when attempting to issue a THROW when not inside a TRY or CATCH block.

A successful THROW operation resets $THROWOBJ to the empty string.

For information on TRY, THROW, and CATCH, see “The TRY-CATCH Mechanism” in the “Error Processing” chapter of Using Caché ObjectScript.

For information on OREFs, see “OREF Basics” in Using Caché Objects.

Setting $THROWOBJ

You can also explicitly reset $THROWOBJ as follows:

```objectscript
SET $THROWOBJ=""
```

$THROWOBJ cannot be set to any value other than the empty string using the SET command. Attempting to do so results in a <ILLEGAL VALUE> error.

See Also

- THROW command
- Error Processing in Using Caché ObjectScript
$TLEVEL

Contains the current nesting level for transaction processing.

Description

$TLEVEL contains the current transaction level, the number of nested open transactions. The number of TSTART commands issued determines the transaction level.

- Each TSTART increments $TLEVEL by 1.
- Each TCOMMIT decrements $TLEVEL by 1.
- Each TROLLBACK 1 decrements $TLEVEL by 1.
- A TROLLBACK resets $TLEVEL to 0.

A $TLEVEL of 0 cannot be decremented. Issuing a TROLLBACK (or TROLLBACK 1) when $TLEVEL=0 performs no operation. Issuing a TCOMMIT when $TLEVEL=0 results in a <COMMAND> error.

The maximum number of transaction levels is 255. Attempting to exceed 255 transaction levels generates a <TRANSACTION LEVEL> error.

This special variable cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error.

SQL and $TLEVEL

$TLEVEL is also set by SQL transaction statements as follows:

- An initial START TRANSACTION sets $TLEVEL to 1. Additional START TRANSACTION statements have no effect on $TLEVEL.
- Each SAVEPOINT statement increments $TLEVEL by 1.
- A ROLLBACK TO SAVEPOINT pointname statement decrements $TLEVEL. The amount of decrement depends on the savepoint specified.
- A COMMIT resets $TLEVEL to 0.
- A ROLLBACK resets $TLEVEL to 0.

Despite their shared use of $TLEVEL, ObjectScript transaction processing differs from, and is incompatible with, SQL transaction processing. An application should not attempt to mix the two types of transaction processing statements within the same transaction.

Transaction Level and the Terminal Prompt

By default, if $TLEVEL is greater than 0 at the conclusion of a command line or program executed from the Terminal prompt, the current transaction level is displayed as a Terminal prompt prefix.

- When $TLEVEL=0, the Terminal prompt displays the namespace name (by default). For example, USER>
- When $TLEVEL>0, the Terminal prompt displays the TLu: prefix before the namespace name, n being an integer 1 through 255. For example, TL4:USER>.

This Terminal prompt display is configurable, as described in ZNSPACE.
The SQL Shell prompt does not display the current transaction level. Upon exiting the SQL Shell the current $TLEVEL value is displayed at the Terminal prompt. This can including transaction levels established before entering the SQL Shell and transaction level changes that occurred while in the SQL Shell.

The MV command (with no argument) enters the interactive MultiValue Shell and immediately rolls back any open transactions. Upon exiting the MultiValue Shell the current $TLEVEL value is 0. The MV command with a MultiValue command line argument does not affect the current transaction level.

### Examples

The following example shows that each TSTART increments $TLEVEL and each TCOMMIT decrements $TLEVEL:

```
WRITE !,"transaction level ",TLEVEL // 0
TSTART
WRITE !,"transaction level ",TLEVEL // 1
TSTART
WRITE !,"transaction level ",TLEVEL // 2
TCOMMIT
WRITE !,"transaction level ",TLEVEL // 1
TCOMMIT
WRITE !,"transaction level ",TLEVEL // 0
```

The following example shows that repeated invocations of TSTART increment $TLEVEL, and TROLLBACK 1 decrements $TLEVEL.

```
WRITE !,"transaction level ",TLEVEL // 0
TSTART
WRITE !,"transaction level ",TLEVEL // 1
TSTART
WRITE !,"transaction level ",TLEVEL // 2
TROLLBACK 1
WRITE !,"transaction level ",TLEVEL // 1
```

The following example shows that repeated invocations of TSTART increment $TLEVEL, and TROLLBACK resets $TLEVEL to 0.

```
WRITE !,"transaction level ",TLEVEL // 0
TSTART
TSTART
TSTART
WRITE !,"transaction level ",TLEVEL // 3
TROLLBACK
WRITE !,"transaction level ",TLEVEL // 0
```

The following example shows that if $TLEVEL is 0, TROLLBACK commands have no effect:

```
WRITE !,"transaction level ",TLEVEL // 0
TROLLBACK
WRITE !,"transaction level ",TLEVEL // 0
TROLLBACK 1
WRITE !,"transaction level ",TLEVEL // 0
TROLLBACK
WRITE !,"transaction level ",TLEVEL // 0
```

### See Also

- TCOMMIT command
- TROLLBACK command
- TSTART command
- Using ObjectScript for Transaction Processing in Using Caché ObjectScript
$USERNAME

Contains the username for the current process.

### Description

$USERNAME contains the username for the current process. This can be in one of two forms:

- The name of the current user; for example: Mary. This value is returned if multiple security domains are not allowed.
- The name and system address of the current user; for example: Mary@jupiter. This value is returned if multiple security domains are allowed.

To allow multiple security domains, go to the Management Portal, select **System Administration**, **Security**, **System Security**, **System-wide Security Parameters**. Select the **Allow multiple security domains** check box. Changes to this setting apply to new invoked processes; changing it does not affect the value returned by the current process.

You cannot use the **SET** command or the **NEW** command to modify this value. However, **NEW $ROLES** also stacks the current $USERNAME value.

Commonly, the $USERNAME value is the username specified at connection time. However, if unauthenticated access is permitted, a user terminal or an ODBC client may connect to Caché without specifying a username. In this case, $USERNAME contains the string “UnknownUser”.

When a process is created using the **JOB** command, it inherits the same $USERNAME and $ROLES values as its parent process.

A username can be created using the SQL **CREATE USER** statement and deleted using the SQL **DROP USER** statement. A user password can be changed using the SQL **ALTER USER** statement. A user can have roles assigned to it, either by using the SQL **GRANT** statement, or by using system utilities to add a role to the user. You can access the list of roles assigned to the current process with the $ROLES special variable. A role can be revoked from a user using the SQL **REVOKE** statement.

$USERNAME is used in Caché SQL as the USER, CURRENT_USER, and SESSION_USER default field values.

You can return the username for the current process, or for a specified process, by invoking the $SYSTEM.Process.UserName() method.

### Examples

The following example returns the username for the current process.

```
WRITE $USERNAME
```

The following example returns the domain name for the current process.

```
WRITE $PIECE($USERNAME,"@",2)
```

### See Also

- ObjectScript **$ROLES** special variable
- Caché SQL: **CREATE TABLE CREATE USER DROP USER ALTER USER GRANT REVOKE %CHECKPRIV**
$X

Contains the current horizontal position of the cursor.

Description

$X contains the current horizontal position of the cursor. As characters are written to a device, Caché updates $X to reflect the horizontal cursor position.

Each printable character that is output increments $X by 1. A carriage return (ASCII 13) or form feed (ASCII 12) resets $X to 0 (zero).

$X is a 16-bit unsigned integer.

- On non-Unicode systems, $X wraps to 0 when its value reaches 65536. In other words, if $X is 65535, the next output character resets it to 0.
- On Unicode systems, $X wraps to 0 when its value reaches 16384 (the two remaining bits are used for Japanese pitch encoding).

You can use the SET command to give a value to $X and $Y. For example, you may use special escape sequences that alter the physical cursor position without updating the $X and $Y values. In this case, use SET to assign the correct values to $X and $Y after you use the escape sequences.

Notes

NLS Character Mapping

The National Language Support (NLS) utility $X/$Y tab defines the $X and $Y cursor movement characters for the current locale. For further details, refer to the section on “System Classes for National Language Support” in Caché Specialized System Tools and Utilities.

$X with Terminal I/O

The following table shows the effects of different characters on $X.

<table>
<thead>
<tr>
<th>Echoed Character</th>
<th>ASCII Code</th>
<th>Effect on $X</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;FORM FEED&gt;</td>
<td>12</td>
<td>$X=0</td>
</tr>
<tr>
<td>&lt;RETURN&gt;</td>
<td>13</td>
<td>$X=0</td>
</tr>
<tr>
<td>&lt;LINE FEED&gt;</td>
<td>10</td>
<td>$X=$X</td>
</tr>
<tr>
<td>&lt;BACKSPACE&gt;</td>
<td>8</td>
<td>$X=$X-1</td>
</tr>
<tr>
<td>&lt;TAB&gt;</td>
<td>9</td>
<td>$X=$X+1</td>
</tr>
<tr>
<td>Any printable ASCII character</td>
<td>32-126</td>
<td>$X=$X+1</td>
</tr>
<tr>
<td>Nonprintable characters (such as escape sequences)</td>
<td>127-255</td>
<td>See Using Caché ObjectScript.</td>
</tr>
</tbody>
</table>

The S(ecret) protocol of the OPEN and USE commands turns off echoing. It also prevents $X from being changed during input, so it indicates the true cursor position.
WRITE $CHAR() changes $X. WRITE * does not change $X. For example, WRITE $X, "/", $CHAR(8), $X performs the backspace (deleting the / character) and resets $X accordingly, returning 01. In contrast, WRITE $X, "/", *8, $X performs the backspace (deleting the / character) but does not reset $X; it returns 02. (See the WRITE command for further details.)

Using WRITE *, you can send a control sequence to your terminal and $X will still reflect the true cursor position. Since some control sequences do move the cursor, you can use the SET command to set $X directly. For example, the following commands move the cursor to column 20 and line 10 on a Digital VT100 terminal (or equivalent) and set $X and $Y accordingly:

```
SET dy=10,dx=20
WRITE *27,*91,dy+1,*59,dx+1,*72
SET $Y=dy,$X=dx
```

ANSI standard control sequences (such as escape sequences) that the device acts on but does not output can produce a discrepancy between the $X and $Y values and the true cursor position. To avoid this problem use the WRITE * (integer expression) syntax and specify the ASCII value of each character in the string. For example, instead of using:

```
WRITE !,$CHAR(27)_"[1m"
WRITE !,$X
```

use this equivalent form:

```
WRITE !,*27,*91,*49,*109
WRITE !,$X
```

As a rule, after any escape sequence that explicitly moves the cursor, you should update $X and $Y to reflect the actual cursor position.

You can set how $X handles escape sequences for the current process using the DX() method of the %SYSTEM.Process class. The system-wide default behavior can be established by setting the DX property of the Config.Miscellaneous class.

### $X with TCP and Interprocess Communication

When you use the WRITE command to send data to either a client or server TCP device, Caché first stores the data in a buffer. It also updates $X to reflect the number of characters in the buffer. It does not include the ASCII characters <RETURN> and <LINE FEED> in this count because they are considered to be part of the record.

If you flush the $X buffer with the WRITE ! command, Caché resets $X to 0 and increments the $Y value by 1. If you flush the $X and $Y buffers with the WRITE # command, Caché writes the ASCII character <FORM FEED> as a separate record and resets both $X and $Y to 0.

### See Also

- WRITE command
- $Y special variable
- I/O Devices and Commands in Caché I/O Device Guide
- Terminal I/O in Caché I/O Device Guide
- Local Interprocess Communication in Caché I/O Device Guide
- TCP Communication in Caché I/O Device Guide
$Y

Contains the current vertical position of the cursor.

### Description

$Y$ contains the current vertical position of the cursor. As characters are written to a device, Caché updates $Y$ to reflect the vertical cursor position.

Each line feed (newline) character (ASCII 10) that is output increments $Y$ by 1. A form feed character (ASCII 12) resets $Y$ to 0.

$Y$ is a 16-bit unsigned integer. $Y$ wraps to 0 when its value reaches 65536. In other words, if $Y$ is 65535, the next output character resets it to 0.

You can use the **SET** command to give a value to $X$ and $Y$. For example, you may use special escape sequences that alter the physical cursor position without updating the $X$ and $Y$ values. In this case, use **SET** to assign the correct values to $X$ and $Y$ after you use the escape sequences.

### Notes

#### NLS Character Mapping

The National Language Support (NLS) utility $X$/SY tab defines the $X$ and $Y$ cursor movement characters for the current locale. For further details, refer to the section on “System Classes for National Language Support” in *Caché Specialized System Tools and Utilities*.

#### $Y$ with Terminal I/O

The following table shows the effects of different characters on $Y$.

<table>
<thead>
<tr>
<th>Echoed Character</th>
<th>ASCII Code</th>
<th>Effect on $Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;FORM FEED&gt;</td>
<td>12</td>
<td>$Y=0</td>
</tr>
<tr>
<td>&lt;RETURN&gt;</td>
<td>13</td>
<td>$Y=Y+1</td>
</tr>
<tr>
<td>&lt;LINE FEED&gt;</td>
<td>10</td>
<td>$Y=Y+1</td>
</tr>
<tr>
<td>&lt;BACKSPACE&gt;</td>
<td>8</td>
<td>$Y=Y+1</td>
</tr>
<tr>
<td>&lt;TAB&gt;</td>
<td>9</td>
<td>$Y=Y+1</td>
</tr>
<tr>
<td>Any printable ASCII character</td>
<td>32-126</td>
<td>$Y=Y+1</td>
</tr>
</tbody>
</table>

The S(ecret) protocol of the **OPEN** and **USE** commands turns off echoing. It also prevents $Y$ from being changed during input, so it indicates the true cursor position.

A **WRITE $CHAR()** that changes vertical position also changes $Y$. A **WRITE** * that changes vertical position does not change $Y$. For example, **WRITE $Y,$CHAR(10),$Y** performs the line feed and increments $Y$. In contrast, **WRITE $Y,*10,$Y** performs the line feed but does not increment $Y$. (See the **WRITE** command for further details.)

Because **WRITE** * does not change $Y$, you can send a control sequence to your terminal and $Y$ will still reflect the true cursor position. Since some control sequences do move the cursor, you can use the **SET** command to set $Y$ directly. For example, the following commands move the cursor to column 20 and line 10 on a VT100-type terminal and set $X$ and $Y$ accordingly:
SET dy=10,dx=20
WRITE *27,*91,dy+1,*59,dx+1,*72
SET $Y=dy,$X=dx

ANSI standard control sequences (such as escape sequences) that the device acts on but does not output can produce a discrepancy between the $X and $Y values and the true cursor position. To avoid this problem, use the WRITE * statement and specify the ASCII value of each character in the string. For example, instead of using the following code:

WRITE $CHAR(27)_"[1m"

use this equivalent form:

WRITE *27,*91,*49,*109

As a rule, after any escape sequence that explicitly moves the cursor, you should update $X and $Y to reflect the actual cursor position.

See Also

- $X special variable
- I/O Devices and Commands in Caché I/O Device Guide
- Terminal I/O in Caché I/O Device Guide
- Interprocess Communication in Caché I/O Device Guide
$ZA

Contains the status of the last READ on the current device.

Description

$ZA contains the status of the last READ on the current device.

This special variable cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error.

Notes

$ZA with Terminal I/O

$ZA is implemented as a sequence of bit flags, with each bit indicating a specific piece of information. The following table shows the possible values, their meanings, and how to test them using the modulo (#) and integer divide (\) operators:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Test</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$ZA#2</td>
<td>A &lt;CTRL-C&gt; arrived, whether or not breaks were enabled.</td>
</tr>
<tr>
<td>1</td>
<td>$ZA#2</td>
<td>The READ timed out.</td>
</tr>
<tr>
<td>2</td>
<td>$ZA#2</td>
<td>I/O error.</td>
</tr>
<tr>
<td>8</td>
<td>$ZA#2</td>
<td>Caché detected an invalid escape sequence.</td>
</tr>
<tr>
<td>9</td>
<td>$ZA#2</td>
<td>The hardware detected a parity or framing error.</td>
</tr>
<tr>
<td>11</td>
<td>$ZA#2</td>
<td>The process is disconnected from its principal device.</td>
</tr>
<tr>
<td>12</td>
<td>$ZA#2</td>
<td>For COM ports: CTS (Clear To Send). A signal sent from the modem to its computer indicating that transmission can proceed. For TCP devices: the device is functioning in Server mode.</td>
</tr>
<tr>
<td>13</td>
<td>$ZA#2</td>
<td>For COM ports: DSR (Data Set Ready). A signal sent from the modem to its computer indicating that it is ready to operate. For TCP devices: the device is currently in the Connected state talking to a remote host.</td>
</tr>
<tr>
<td>14</td>
<td>$ZA#2</td>
<td>Ring set if TRUE.</td>
</tr>
<tr>
<td>15</td>
<td>$ZA#2</td>
<td>Carrier detect set if TRUE.</td>
</tr>
<tr>
<td>16</td>
<td>$ZA#2</td>
<td>CE_BREAK COM port error state.</td>
</tr>
<tr>
<td>17</td>
<td>$ZA#2</td>
<td>CE_FRAME COM port error state.</td>
</tr>
<tr>
<td>18</td>
<td>$ZA#2</td>
<td>CE_IOE COM port error state.</td>
</tr>
<tr>
<td>19</td>
<td>$ZA#2</td>
<td>CE_OVERRUN COM port error state.</td>
</tr>
<tr>
<td>20</td>
<td>$ZA#2</td>
<td>CE_RXPARITY COM port error state.</td>
</tr>
<tr>
<td>21</td>
<td>$ZA#2</td>
<td>CE_TXFULL COM port error state.</td>
</tr>
<tr>
<td>22</td>
<td>$ZA#2</td>
<td>TXHOLD COM port error state. Set if any of the following fields are true in the error mask returned by ClearCommError(): fCtsHold, fDsrHold, fRlsdHold, fXoffHold, fXoffSent.</td>
</tr>
</tbody>
</table>
COM ports use bits 12 through 15, 24 and 25 to report the status of modem control pins. This can be done regardless of whether Caché modem control checking is on or off for the port. A user can enable or disable $ZA error reporting for COM ports by setting the OPEN or USE command portstate parameter (byte 8, to be specific). If error reporting is enabled, the port error state is reported in bits 16 through 22. For further details, see Terminal I/O in Caché I/O Device Guide.

You can use the DisconnectErr() method of the %SYSTEM.Process class for modem disconnect detection for the current process. The system-wide default behavior can be established by setting the DisconnectErr property of the Config.Miscellaneous class.

### $ZA With Magnetic Tape I/O

With magnetic tape I/O, the bit fields in $ZA indicate errors and special conditions. Caché updates $ZA after each command that references the magnetic tape device.

The following table shows the meanings of the $ZA bits for magnetic tape I/O. Note the Trap column. The letter Y indicates a <MAGTAPE> error. If you have set the $ZTRAP variable, Caché issues the associated $ZTRAP error code.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>Trap</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Y</td>
<td>Logical Error (mixed Reads and Writes)</td>
<td>To switch between reading and writing, either Close and then Open the device or issue a Forward Space, Backspace, or Rewind command.</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>N</td>
<td>Write Protected</td>
<td>Reflects the state of the OPEN or USE read-only parameter at all times. This bit does not reflect the state of the tape’s physical write protection (write ring or write lock) because many versions of UNIX® give no notice of tape write protection until an actual write to tape is attempted. If you attempt to open a write-protected 9-track tape without the read-only parameter, Caché sets this bit and opens the tape as read only. No error occurs.</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>Y</td>
<td>Error Summary</td>
<td>The error summary is the logical OR of all conditions that cause a Caché error (all conditions marked Y under Trap)</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>N</td>
<td>Beginning of Tape [BOT]</td>
<td>On UNIX® systems this bit is set upon rewind, and cleared when tape is opened.</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>N</td>
<td>On Line</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>128</td>
<td>Y</td>
<td>Controller or drive error</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
<td>N</td>
<td>End of Tape [EOT]</td>
<td>Not supported on most UNIX® platforms.</td>
</tr>
</tbody>
</table>
Caché sets the Tape Mark bit when it encounters a tape mark on Read, Read Block, Forward Space or Backspace. This sets the Error Summary bit and traps to $ZTRAP on Read, Read Label, and Read Block.

Some bits indicate error conditions, while other bits indicate conditions that do not necessarily produce an error. To monitor these non-error conditions, your program must test the appropriate bit of $ZA after every magnetic tape operation. For example, if your program might write beyond the end of the tape, it must check bit 10 (End of Tape).

To test a bit, integer divide $ZA by the value listed for that bit in the table and perform a modulo 2 operation. For example, the following command checks if bit 14 (Tape Mark) is set:

```
USE 47 IF $ZA\16384#2 {DO Endfile}
```

where 16384 equals 2 to the 14th power and #2 indicates the modulo 2 operation. Since any number to the 0 power equals 1, you do not need a divisor to check bit 0 (Logical Error). For example:

```
USE 47 GOTO Logerr:$ZA#2
```

### See Also

- READ command
- $ZB special variable
- $ZTRAP special variable
- I/O Devices and Commands in Caché I/O Device Guide
- Terminal I/O in Caché I/O Device Guide
- Magnetic Tape I/O in Caché I/O Device Guide
- Sequential File I/O in Caché I/O Device Guide

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>Trap</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>16384</td>
<td>Y</td>
<td>Tape Mark</td>
<td>Caché sets the Tape Mark bit when it encounters a tape mark on Read, Read Block, Forward Space or Backspace. This sets the Error Summary bit and traps to $ZTRAP on Read, Read Label, and Read Block.</td>
</tr>
<tr>
<td>15</td>
<td>32768</td>
<td>Y</td>
<td>Tape Not Ready</td>
<td></td>
</tr>
</tbody>
</table>
Contains status information for the current I/O device.

**Description**

$ZB$ contains status information specific to the current I/O device following a **READ** operation.

- When reading from a terminal, sequential file, or other character-based I/O device, $ZB$ contains the *terminating* character of the read operation. This can be a terminator character (such as `<RETURN>`), the final character of the input data if the read operation does not require a terminator character, or the null string if a terminator character is required but was not received (for example, if the read operation timed out).
- When reading from a block-based I/O device, such as magnetic tape, $ZB$ contains the number of bytes remaining in the I/O buffer. $ZB$ also contains the number of bytes in the I/O buffer when writing to magnetic tape.

This special variable cannot be modified using the **SET** command. Attempting to do so results in a `<SYNTAX>` error.

$ZB$ and $KEY$ can both be used to return the **READ** termination character when reading from a character-based device or file. For character-based reads, these two special variables are very similar, but not identical. For block-based reads and writes (such as magnetic tape) use $ZB$; $KEY$ does not provide support for block-based read and write operations. See $KEY$ for further details.

**End-of-File Behavior**

By default, Caché handles an end-of-file on a sequential file by issuing an `<ENDOFFILE>` error; it does not set $ZB$. You can configure end-of-file behavior in a manner compatible with MSM. In this case, when an end-of-file is encountered, Caché does not issue an error, but sets $ZB$ to """ (the null string), and sets $ZEOF$ to -1.

To configure end-of-file handling, go to the Management Portal, select **[System] > [Configuration] > [Compatibility Settings]**. View and edit the current setting of **SetZEOF**. When set to "true", Caché sets $ZB$ to """ (the null string), and sets $ZEOF$ to -1. The default is “false”.

You can control end-of-file handling for the current process using the **SetZEOF()** method of the %SYSTEM.Process class. The system-wide default behavior can be established by setting the **SetZEOF** property of the **Config.Miscellaneous** class.

**Reading from a Terminal or File**

$ZB$ contains the terminating character (or character sequence) from a read operation involving a terminal, sequential file, or other character-based I/O device. $ZB$ can contain any of the following:

- A termination character, such as a carriage return.
- An escape sequence (up to 16 characters).
- The $n$th character in a fixed-length **READ $x#n$**. (In this case, the $KEY$ special variable returns the null string.)
- The single character of **READ *x**.
- A null string ("""") after a timed **READ** expires.

For example, consider the following variable-length read with a five-second timeout:

```
zbread
  READ !,"Enter number:",num:5
  WRITE !, num
  WRITE !, $ASCII($ZB)
  QUIT
```
If the user types 123 at the READ prompt and presses <RETURN>, Caché stores 123 in the num variable and stores <RETURN> (ASCII decimal code 13, hexadecimal 0D) in $ZB. If the READ times out, $ZB contains the null string; $ASCII("") returns a value of –1.

**$ZB on the Command Line**

When issuing commands interactively from the Terminal command line, you press <RETURN> to issue each command line. The $ZB and $KEY special variables record this command line terminator character. Therefore, when using $ZB or $KEY to return the termination status of a read operation, you must set a variable as part of the same command line.

For example, if you issue the command:

>READ x:10

from the command line, then check $ZB it will not contain the results of the read operation; it will contain the <RETURN> character that executed the command line. To return the results of the read operation, set a local variable with $ZB in the same command line, as follows:

>READ x:10 SET rzb=$ZB

This preserves the value of $ZB set by the read operation. To display this read operation value, issue either of the following command line statements:

>WRITE $ASCII(rzb)
  ; returns –1 for null string (time out),
  ; returns ASCII decimal value for terminator character
>ZZDUMP rkey
  ; returns blank line for null string (time out)
  ; returns hexadecimal value for terminator character

**$ZB with Magnetic Tape I/O**

$ZB contains status information about the driver buffer. Specifically, it contains the number of bytes that remain in the magnetic tape drive’s internal buffer.

Immediately after you read a block, Caché sets $ZB to that block’s size. As you transfer logical records from the buffer to variables (with READ commands), Caché decrements the $ZB value until it reaches 0 and the next block read occurs.

When you write to tape, $ZB shows the available space (in bytes) remaining in the driver’s internal buffer. Immediately after you write a block, Caché sets $ZB to the buffer size specified with the OPEN command. As you transfer logical records from Caché variables into the buffer (with WRITE commands), Caché decrements the $ZB number until it reaches 0 and the block write occurs.

Most magnetic tape programs need not be concerned with $ZB unless they must deal with unusual formats and variable length blocks.

To monitor magnetic tape operations, your program can test the appropriate bit of $ZA after each read and write.

The following code checks both $ZA and $ZB after each magnetic tape read, and sets MTER when either of these variables indicates an error. It also sets $ZTRAP when a magnetic tape error occurs.

```ObjectScript
; $$MTIN(mtdev) = the next logical record from magtape
; device mtdev.
; Also returns za=$ZA and zb=$ZB
; On a magtape error, mterr=1 and $$MTIN(mtdev)=""
; Excepts the caller to have set $ZT to trap other
; errors.
MTIN(io)
NEW rec,curdev
SET mterr=0,curdev=$IO,6ZT="MTIERR"
USE io
READ rec
MTIEXIT
SET za=$ZA,zb=$ZB
USE curdev
QUIT rec
MTIERR
```
IF $ERROR("MAGTAPE") {
    USE curdev
    ZQUIT 1
    GOTO @$ZTRAP }
; Use caller's error trap.
ELSE {
    SET $ZTRAP="",mterr=1,rec=""
    GOTO MTIEXIT }

If a terminator completes a **READ**, Caché mode returns the terminator as a string in $ZB.

If an escape sequence terminates a **READ**, Caché mode returns the ASCII escape sequence as a string in $ZB.

**See Also**

- **READ** command
- **WRITE** command
- **$KEY** special variable
- **$ZA** special variable
- **$ZEOF** special variable
- **I/O Devices and Commands** in *Caché I/O Device Guide*
- **Terminal I/O** in *Caché I/O Device Guide*
- **Magnetic Tape I/O** in *Caché I/O Device Guide*
- **Sequential File I/O** in *Caché I/O Device Guide*
- **The Spool Device** in *Caché I/O Device Guide*
$ZCHILD

Contains the ID of the last child process.

Description

$ZCHILD contains the ID of the last child process that the current process created with the JOB command. If your process has not used JOB to create a child process, $ZCHILD returns 0 (zero).

In MSM language mode, the $ZC special variable (spelled as shown) has a different use. It is used for determining end-of-file in sequential file reads.

This special variable cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error.

Notes

$ZCHILD and the Successful Starting of Jobs

$ZCHILD being set does not mean that the job was successfully started. It only means that the process was created and the parameters were passed successfully.

For example, if you use JOB to spawn a routine that does not exist, both $TEST and $ZCHILD report that the JOB command succeeded, although that new job immediately dies with a <NOROUTINE> error.

$ZC in MSM Language Mode

MSM language mode supports a special use of the $ZC special variable.

If you have set the current language mode to MSM using the LanguageMode(8) method of the %SYSTEM.Process class, the $ZC special variable is set during sequential file reads. This provides compatibility with the MSM $ZC variable. (In all other language modes, $ZC is not set during file reads; $ZC is an abbreviation for $ZCHILD and has a completely different functionality.)

In MSM language mode, a successful sequential file read sets $ZC=0.

In MSM language mode, an end-of-file condition sets $ZC=–1 (negative 1). An <ENDOFFILE> error does not occur.

However, Caché $ZC is not identical to the MSM $ZC:

MSM sets its $ZC=–1 (negative 1) if the last line of the file is not terminated with the delimiter character(s). Caché does not check for delimiter characters; it sets $ZC=0 instead of –1 in this case.

MSM sets its $ZC=1 if an I/O error occurs during a read. Caché does not support this functionality; instead, Caché issues a <READ> error.

See Also

Child processes:

• JOB command
• ^$JOB structured system variable
• $JOB special variable
• $TEST special variable
• $ZPARENT special variable
MSM language mode:

- $ZB special variable
- $ZEOF special variable
- Sequential File I/O in Caché I/O Device Guide
$ZEOF

Contains flag indicating whether end-of-file has been reached.

**Description**

Following each sequential file **READ**, Caché sets the $ZEOF special variable to indicate whether or not the end of the file has been reached. This special variable is provided for compatibility with MSM routines that use $ZC device status checking.

Caché sets $ZEOF to the file status of the last device used. For example, if you read from a sequential file then write to the principal device, Caché resets $ZEOF from the sequential file end-of-file status to the principal device status. Therefore, you should check the $ZEOF value (and, if necessary, copy it to a variable) immediately after a sequential file **READ**.

Caché sets $ZEOF to the following values:

-1 End-of-file reached

0 Not at end-of-file

To use this feature, you must disable the **<ENDOFFILE>** error for sequential files.

- To disable this for the current process, call the **SetZEOF()** method of the %SYSTEM.Process class.
- To disable this system-wide, either set the *SetZEOF* property of the *Config.Miscellaneous* class, or go to the Management Portal and select **System Administration**, **Configuration**, **Additional Settings**, **Compatibility** ([System] > [Configuration] > [Compatibility Settings]). View and edit the current setting of **SetZEOF**. This option controls the behavior when Caché encounters an unexpected end-of-file when reading a sequential file. When set to “true”, Caché sets the $ZEOF special variable to indicate that you have reached the end of the file. When set to “false”, Caché issues an **<ENDOFFILE>** error. The default is “false”.

When the end of a file is reached, rather than issuing an **<ENDOFFILE>** error, the **READ** will return a null string, set $ZB=null and set $ZEOF=–1.

$ZEOF does not support all of the features of the MSM $ZC function. Unlike $ZC, $ZEOF does not identify file delimiter characters or I/O errors. $ZEOF does not check for proper file termination with file delimiter characters. I/O errors are detected by a **READ** command error, not by $ZEOF.

You cannot modify this special variable using the **SET** command. Attempting to do so results in a **<SYNTAX>** error.

**See Also**

- **$ZCHILD** special variable
- **$ZB** special variable
- **Sequential File I/O** in *Caché I/O Device Guide*
$ZEOS

Contains end-of-stream status when reading a compressed stream.

**Description**

$ZEOS contains a boolean value that indicates whether the end of an incoming (compressed) stream has been received and processed. If $ZEOS=1, an end-of-stream for a compressed data stream has been received. The $ZEOS value is only meaningful when stream compression/decompression is active (/GZIP=1). You activate stream compression/decompression by issuing the /GZIP command keyword from an OPEN or USE command.

You must check the $ZEOS value before disabling stream compression/decompression by changing the setting to /GZIP=0. If you issue a USE command with /GZIP=0 before the end of a compressed incoming stream has been processed, the USE generates a <TRANSLATE> error. If the end of the compressed incoming stream has not been reached ($ZEOS=0) you must issue block READ commands until $ZEOS=1.

This special variable cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error.

**Example**

The following example begins with /GZIP=1 (compression enabled). It performs a loop which tests for $ZEOS=1, and issues READ commands until $ZEOS=1. It then can set /GZIP=0 (compression disabled):

```
OPEN dev:/GZIP=1
READ block#length
FOR {QUIT:$ZEOS
    READ x:10 }
USE dev:/GZIP=0
```

**See Also**

- Sequential File I/O chapter of Caché I/O Device Guide
- TCP Client/Server Communication chapter of Caché I/O Device Guide
- Terminal I/O chapter of Caché I/O Device Guide
- Magnetic Tape I/O chapter of Caché I/O Device Guide
$ZERROR

Contains the name and location of the last error.

Description

$ZERROR contains the name of the most recent error, the location of the most recent error (where applicable), and (for certain error codes) additional information about what caused the error. $ZERROR always contains the most recent error for the appropriate language mode.

The $ZERROR value is intended for use immediately following an error. Because a $ZERROR value may not be preserved across routine calls, users that wish to preserve a $ZERROR value for later use should copy it to a variable. It is strongly recommended that users set $ZERROR to the null string ("") immediately after use.

The string contained in $ZERROR can be in any of the following forms:

- `<error>`
- `<error>entryref`
- `<error>info`
- `<error>entryref info`

<table>
<thead>
<tr>
<th><code>&lt;error&gt;</code></th>
<th>The error name. The error name is always returned in all capital letters, enclosed in angle brackets. It may contain blank spaces.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>entryref</code></td>
<td>A reference to the line of code in which the error occurred. This consists of the label name and line offset from that label, followed by a ^ and the program name. This <code>entryref</code> follows immediately after the closing angle bracket of the error name. When invoking $ZERROR from Terminal, this <code>entryref</code> information is not meaningful and is therefore not returned. A reference to the routine most recently loaded into the routine buffer using ZLOAD.</td>
</tr>
<tr>
<td><code>info</code></td>
<td>Additional information specific to certain error types (see table below). This information is separated from <code>&lt;error&gt;</code> or <code>&lt;error&gt;entryref</code> by a blank space. If there are multiple components to <code>info</code>, they are separated by a comma.</td>
</tr>
</tbody>
</table>

For example, a program (named zerrortest) contains the following routine (named ZerrorMain) which attempts to write the contents of `fred`, an undefined local variable:

```
ZerrorMain
TRY {
  SET $ZERROR=""
  WRITE "$ZERROR = ",$ZERROR,!
  WRITE fred )
CATCH {
  WRITE "$ZERROR = ",$ZCVT($ZERROR,"O","HTML")
  }
```

In the above example, the first $ZERROR contains a null string (""), because no errors have occurred since $ZERROR was reset to the null string. The attempt to write an undefined variable sets $ZERROR and throws it to the CATCH block. This $ZERROR contains `<UNDEFINED>ZerrorMain+4^zerrortest *fred`, specifying the name of the error, the location, and additional information specific to that type of error. In this case, the additional information is the name of the undefined local variable `fred`; the asterisk prefix indicates that it is a local variable. (Note that $ZCVT($ZERROR, "O", "HTML") is used in this example because Caché error names are enclosed in angle brackets and this example is run from a web browser.)

An `entryref` can appear as follows:
$ZERROR

The maximum length of the $ZERROR value is 512 characters. A value exceeding that length is truncated to 512 characters.

**AsSystemError() Method**

The AsSystemError() method of the %Exception.SystemException class returns the same value as $ZERROR. This is shown in the following example:

```ObjectScript
TRY {
    KILL mylocal
    WRITE mylocal
}
CATCH myerr {
    WRITE "AsSystemError is: ",myerr.AsSystemError(),!
    WRITE "$ZERROR is: ",$ZERROR
}
```

AsSystemError() is preferable to $ZERROR in a TRY/CATCH exception handling block structure, because $ZERROR could be overwritten by an error occurring during exception handling.

**Additional Information For Some Errors**

When certain types of errors occurs, $ZERROR returns the error in the following format:

```ObjectScript
<ERRORCODE>entryref info
```

The info component contains additional information about what caused the error. The following table gives a list of errors that include additional info and the format of that information. The error code is separated from the info component by a space character.
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Info Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;UNDEFINED&gt;</td>
<td>The name of the undefined variable (including any subscripts used). This may be a local variable, a process-private global, a global, or a multidimensional class property. Local variable names are prefixed by an asterisk. Multidimensional property names start with a period to distinguish them from local variable names. You can change Caché behavior to not generate an &lt;UNDEFINED&gt; error when referencing an undefined variable by setting the %SYSTEM.Process.Undefine() method.</td>
</tr>
<tr>
<td>&lt;SUBSCRIPT&gt;</td>
<td>The subscript reference in error: the line reference (routine and line offset) that generated the error, the subscripted variable, and which subscript level is in error. For a Structured System Variable (SSVN), only the line reference (routine and line offset) is provided. You can change default behavior to not generate a &lt;SUBSCRIPT&gt; error when referencing a global variable with a null string subscript by setting the %SYSTEM.Process.NullSubscripts() method. Null string subscripts are not permitted for local variables.</td>
</tr>
<tr>
<td>&lt;NOROUTINE&gt;</td>
<td>Prefixed by an asterisk, the referenced routine name.</td>
</tr>
<tr>
<td>&lt;CLASS DOES NOT EXIST&gt;</td>
<td>Prefixed by an asterisk, the referenced class name.</td>
</tr>
<tr>
<td>&lt;PROPERTY DOES NOT EXIST&gt;</td>
<td>Prefixed by an asterisk, the name of the referenced property, followed by a comma separator and the class name it is supposed to be in.</td>
</tr>
<tr>
<td>&lt;METHOD DOES NOT EXIST&gt;</td>
<td>Prefixed by an asterisk, the name of the method invoked, followed by a comma separator and the class name it is supposed to be in.</td>
</tr>
<tr>
<td>&lt;PROTECT&gt;</td>
<td>The name of the global referenced and the name of the directory containing it, separated by a comma.</td>
</tr>
<tr>
<td>&lt;THROW&gt;</td>
<td>Prefixed by an asterisk, the object name, followed by the value returned by the DisplayString() method.</td>
</tr>
<tr>
<td>&lt;COMMAND&gt;</td>
<td>When invoking TCOMMIT when not in a transaction, the info component is *NoTransaction. When invoking a user-defined function that does not return a value, the info component is a message that includes the location of the command that should have returned the value.</td>
</tr>
<tr>
<td>&lt;DIRECTORY&gt;</td>
<td>Prefixed by an asterisk, the full pathname of the invalid directory.</td>
</tr>
<tr>
<td>&lt;FRAMESTACK&gt;</td>
<td>When a &lt;FRAMESTACK&gt; error terminates a process, the &lt;FRAMESTACK&gt; error with additional information is written as a message to mgr/console.log. The informational message shows the process id (pid) of the terminated process and the line reference (routine and line offset) that generated the error. For example: (pid) 0 &lt;FRAMESTACK&gt; at +13&quot;USER&quot;mytest</td>
</tr>
</tbody>
</table>

The names of variables local to routines (or methods), as well as the names of undefined routines, classes, properties, and methods, are prefixed with an asterisk (*). Process-private globals are identified by their ^|| prefix. Global variables are identified by their ^ (caret) prefix. Class names are presented in their %-prefix form.
The following examples show additional error information specifying the cause of the error. In each case, the specified item does not exist. Note that the info component of the generated error is separated from the error name by a blank space. The asterisk (*) indicates a local variable, a class, a property, or a method. The caret (^) indicates a global, and || indicates a process-private global.

Examples of **<UNDEFINED>** errors:

```objectscript
UndefTest ;

ZNSPACE "SAMPLES"
KILL x, abc(2)
KILL "xyz(1,1)," ^"USER"|xyz(1,2)
KILL ^|ppg(1)," ^ppg(2)
TRY {WRITE x }  // undefined local variable
  CATCH {WRITE $ZERROR, !}
TRY {WRITE abc(2)}  // undefined subscripted local variable
  CATCH {WRITE $ZERROR, !}
TRY {WRITE "xyz(1,1)" }  // undefined global
  CATCH {WRITE $ZERROR, !}
TRY {WRITE ^"USER"|xyz(1,2) }  // undefined global in another namespace
  CATCH {WRITE $ZERROR, !}
TRY {WRITE ^|ppg(1) }  // undefined process-private global
  CATCH {WRITE $ZERROR, !}
TRY {WRITE ^"^"|ppg(2) }  // undefined process-private global
  CATCH {WRITE $ZERROR, !}

<UNDEFINED>UndefTest+5^MyProg *x
<UNDEFINED>UndefTest+7^MyProg *abc(2)
<UNDEFINED>UndefTest+9^MyProg "xyz(1,1)
<UNDEFINED>UndefTest+11^MyProg "xyz(1,2)
<UNDEFINED>UndefTest+13^MyProg ^ppg(1)
<UNDEFINED>UndefTest+15^MyProg ^ppg(2)
```

Examples of **<SUBSCRIPT>** errors:

```objectscript
SubscriptTest ;

DO $SYSTEM.Process.NullSubscripts(0)
KILL abc, xyz
TRY {SET abc(1,2,3,"")=123 }
  CATCH {WRITE $ZERROR, !}
TRY {SET xyz(1,$JUSTIFY(1,1000))=1}
  CATCH {WRITE $ZERROR, !}

<SUBSCRIPT>SubscriptTest+3^MyProg *abc() Subscript 4 is ""
<SUBSCRIPT>SubscriptTest+5^MyProg *xyz() Subscript 2 > 511 chars
```

Examples of **<NOROUTINE>** errors:

```objectscript
NoRoutineTest ;

KILL ^NotThere
TRY {DO ^NotThere }
  CATCH {WRITE $ZERROR, !}
TRY {JOB ^NotThere }
  CATCH {WRITE $ZERROR, !}
TRY {GOTO ^NotThere }
  CATCH {WRITE $ZERROR, !}

<NORoutine>NoRoutineTest+2^MyProg *NotThere
<NORoutine>NoRoutineTest+4^MyProg *NotThere
<NORoutine>NoRoutineTest+6^MyProg *NotThere
```

Examples of object errors:

```objectscript
WRITE $SYSTEM.XXQL.MyMethod()
<CLASS DOES NOT EXIST> ^%SYSTEM.XXQL
DO $SYSTEM.SQL.MyMethod()
<METHOD DOES NOT EXIST> ^MyMethod,%SYSTEM.SQL
SET x=#class($SQL.Statement).%New()
WRITE x, MyProp
<PROPERTY DOES NOT EXIST> ^MyProp,%SQL.Statement
```

Example of **<PROTECT>** error (on Windows):

```objectscript
// user does not have access privileges for %SYS namespace
SET x="%^SYS\"|var
<PROTECT> "var, c:\intersystems\cache\mgr\`
```
Example of a <COMMAND> error when invoking a user-defined function. In this example, the MyFunc QUIT command does not return a value. This generates a <COMMAND> error with the entryref specifying the location of the call to $$MyFunc, and the info message specifying the location of the QUIT command:

```
Main
TRY {
  KILL x
  SET x=$$MyFunc(7,10)
  WRITE "$\text{returned value is $x}$
  RETURN
} CATCH { WRITE "$ZERROR = "$ZCVT($ZERROR,"O","HTML"),!
}
MyFunc(a,b)
SET c=a+b
QUIT
```

The same <COMMAND> error when invoking the function as a procedure with the PUBLIC keyword:

```
Main
TRY {
  KILL x
  SET x=$$MyFunc(7,10)
  WRITE "$\text{returned value is $x}$
  RETURN
} CATCH { WRITE "$ZERROR = "$ZCVT($ZERROR,"O","HTML"),!
}
MyFunc(a,b) PUBLIC {
  SET c=a+b
  QUIT
}
```

Example of <DIRECTORY> error (on Windows):

```
TRY { SET prev=$SYSTEM.Process.CurrentDirectory("bogusdir")
  WRITE "$\text{previous directory: $prev}$
  RETURN
} CATCH { WRITE "$ZERROR = "$ZCVT($ZERROR,"O","HTML"),!
    QUIT
}
```

Pre-5.1 Error Handling Code

A consequence of adding an info component to these error codes for Caché 5.1 and subsequent releases is that pre-5.1 error handling routines that made assumptions about the format of the string in $ZERROR may require redesign to work as before. For example, the following will no longer work in version 5.1:

```
WRITE "$\text{Error line: $PIECE($ERROR, ">", 2)}$
```

and should be changed to be something like:

```
WRITE "$\text{Error line: $PIECE($PIECE($ERROR, ">", 2), ", ", 1)}$
```

Notes

**ZLOAD and Error Messages**

Following a ZLOAD operation, the name of the routine loaded into the routine buffer appears in the entryref portion of subsequent error messages. This persists for the duration of the process, or until removed using ZREMOVE, or removed or replaced by another ZLOAD. The following example shows this display of the contents of the routine buffer:
$ZERROR and the Program Stack

The <error> portion of the $ZERROR string contains the most recent error message. The contents of the entryref portion of the $ZERROR string reflect the stack level of the most recent error. The following terminal session attempts to call the nonsense command GOBBLEDEGOOK, resulting in a <SYNTAX> error. It also runs ZerrorMain (specified above), resulting in the $ZERROR value <UNDEFINED>. Subsequent $ZERROR values during this terminal session reflect this program call, as shown in the following:

$ZERROR Actions when $ZTRAP is Set

When an error occurs and $ZTRAP is set, Caché returns the error message in $ZERROR and branches to the error-trap handler specified for $ZTRAP. (For a list of the possible error texts, refer to System Error Messages in Caché Error Reference.)

Setting $ZERROR

You can set $ZERROR with the SET command to a value of up to 512 characters only in Caché mode. Values longer than 512 characters are truncated to 512.

Resetting $ZERROR to the null string (""") is strongly recommended following error processing.

See Also

- CATCH command
- ZTRAP command
- $ECODE special variable
- $ZTRAP special variable
- Error Handling chapter in Using Caché ObjectScript
- System Error Messages in Caché Error Reference
$ZHOROLOG

Contains the number of seconds elapsed since Caché startup.

$ZHOROLOG
$ZH

Description

$ZHOROLOG contains the number of seconds that have elapsed since the most recent Caché startup. This is a count, which is independent of clock changes and day boundaries. The value is expressed as a floating point number, indicating seconds and fractions of a second. The number of decimal digits is platform-dependent. $ZHOROLOG truncates trailing zeros in this fractional portion.

This special variable cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error.

Note: Because of a limitation in the Windows operating system, putting your Windows system into hibernate or standby mode may cause $ZHOROLOG to return unpredictable values. This problem does not affect $HOROLOG or $ZTIMESTAMP values.

Examples

This example outputs the current $ZHOROLOG value.

```
WRITE $ZHOROLOG
```

returns a value such as: 1036526.244932.

The following example shows how you might use $ZHOROLOG to time events and do benchmarks. This example times an application through 100 executions, then finds the average runtime.

```
Cycletime
  SET start=$ZHOROLOG
  FOR i=1:1:100 { DO Myapp }
  SET end=$ZHOROLOG
  WRITE !,"Average run was ", (end-start)/100, " seconds."
QUIT
Myapp
  WRITE !,"executing my application"
  ; application code goes here
QUIT
```

See Also

- $HOROLOG special variable
- $ZTIMESTAMP special variable
Contains information about the current terminal I/O device.

For a terminal device that is a Terminal, $ZIO contains the string TRM:. If the current terminal device is connected remotely, $ZIO contains information about the remote connection.

For a terminal device connected through TELNET, $ZIO contains the following: host|port:

**host** | The remote host IP address, in either IPv4 format: nnn.nnn.nnn.nnn, where nnn is a decimal number, or in IPv6 format: h:h:h:h:h:h:h:h, where h is a hexadecimal number. Further details on IPv4 and IPv6 formats can be found in the section “Use of IPv6 Addressing” in the chapter “Server Configuration Options” in the Caché Programming Orientation Guide.

**port** | The remote IP port number.

These two values are separated by a vertical bar character. For example, 127.0.0.1|23. For further information, refer to the %Library.NetworkAddress class, a subclass of the data type class %Library.String.

If the current device is not a terminal:

- If a file, $ZIO contains the full canonical pathname of the file.
- If not a file, $ZIO contains the null string.

The following example returns the current device information:

```ObjectScript
SET x=$CASE($ZIO,"TRM:":"a terminal",
              "CON:":"a console",
              "":"neither terminal nor file")
WRITE "The current device is ",x
```

This special variable cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error.

### See Also

- $IO special variable
- $PRINCIPAL special variable
- I/O Devices and Commands in Caché I/O Device Guide
- Terminal I/O in Caché I/O Device Guide
$ZJOB

Contains job status information.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Set to</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Job started from the Terminal prompt.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Job started from a routine.</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Job started by the JOB command.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Job started by signing on either at the Terminal prompt or from a routine.</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>&lt;INTERRUPT&gt; enabled. A CTRL-C can interrupt a running program. Refer to BREAK flag for details.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>&lt;INTERRUPT&gt; disabled except for terminal lines for which &lt;INTERRUPT&gt; has been explicitly enabled by OPEN or USE commands.</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>&lt;INTERRUPT&gt; received and pending.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>&lt;INTERRUPT&gt; not received. The value 8 is cleared by the OPEN and USE commands and by an error trap caused by a CTRL-C.</td>
</tr>
<tr>
<td>1024</td>
<td>1</td>
<td>Journaling is disabled regardless of other conditions.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Journaling is enabled for this job if other conditions indicate journaling.</td>
</tr>
</tbody>
</table>

This special variable cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error.

Examples

The following example returns $ZJOB as an integer:

```
WRITE $ZJOB
```

The following example returns each $ZJOB bit value:

```
WRITE " bit 1=",$ZJOB\1#2,!
WRITE " bit 2=",$ZJOB\2#2,!
WRITE " bit 4=",$ZJOB\4#2,!
WRITE " bit 8=",$ZJOB\8#2,!
WRITE "bit 1024=",$ZJOB\1024#2
```

Bit 1 can also be returned using $ZJOB\2.

See Also

- JOB command
•  $JOB special variable
$ZMODE

Contains current I/O device OPEN parameters.

Syntax

$ZMODE
$ZM

Description

$ZMODE contains the parameters specified with the OPEN or USE command for the current device. The string returned contains the parameters used to open the current I/O device in canonical form. These parameter values are separated by backslash delimiters. Open parameters like "M" on TCP/IP IO are canonicalized to "PSTE". The "Y" and "K" parameter values are always placed last.

This special variable cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error.

Examples

The following example uses $ZMODE to return the parameters of the current device:

```ObjectScript
WRITE !,"The current OPEN modes are: ",$PIECE($ZMODE,"\")
WRITE !,"The NLS collation is: ",$PIECE($ZMODE,"\",2)
WRITE !,"The network encoding is: ",$PIECE($ZMODE,"\",4)
```

The following example sets parameters for the current device with the USE command. It checks the current parameters with $ZMODE before and after the USE command. To test whether a specific parameter was set, this example uses the $PIECE function with the backslash delimiter, and tests for a value using the Contains operator (\). (See  Operators in Using Caché ObjectScript.):

```ObjectScript
Zmodetest
WRITE !, $ZMODE
  IF $PIECE($ZMODE,"\")\["S" {  
    WRITE !, "S is set"  
  } ELSE {WRITE !, "S is not set" }  
USE 0:("":"IS":$CHAR(13,10))
WRITE !, $ZMODE
  IF $PIECE($ZMODE,"\")\["S" {  
    WRITE !, "S is set"  
  } ELSE {WRITE !, "S is not set" }  
QUIT
```

SAMPLES>DO ^zmodetest
RY\Latin1\K\UTF8\nS is not set
SIRy\Latin1\K\UTF8\nS is set

See Also

- OPEN command
- USE command
- $IO special variable
- I/O Devices and Commands in Caché I/O Device Guide
$ZNAME

Contains the current routine name.

<table>
<thead>
<tr>
<th>$ZNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ZN</td>
</tr>
</tbody>
</table>

**Description**

$ZNAME contains the name of the routine executing on the current process. Routine names are case-sensitive. If no routine is currently executing, $ZNAME contains a null string.

This special variable cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error.

To return the routine name for a specified process, use a %SYS.ProcessQuery class method, as shown in the following example:

```objectscript
WRITE ##CLASS(%SYS.ProcessQuery).%OpenId($JOB).RoutineGet()
```

The $ZNAME value can be set by any of the following commands:

- ZLOAD command
- ZSAVE command
- Argumentless ZREMOVE command (sets to a null string)
- DO command
- GOTO command with ^routine
$ZNSPACE

Contains the current namespace name.

| $ZNSPACE |

Description

$ZNSPACE contains the name of the current namespace. By setting $ZNSPACE, you can change the current namespace.

To obtain the current namespace name:

```
WRITE $ZNSPACE
```

You can also obtain the name of the current namespace by invoking the NameSpace() method of %SYSTEM.SYS class, as follows:

```
WRITE %SYSTEM.SYS.NameSpace()
```

You can test whether a namespace is defined by using the Exists() method of the %SYS.Namespace class, as follows:

```
WRITE ##class(%SYS.Namespace).Exists("USER"),! ; an existing namespace
WRITE ##class(%SYS.Namespace).Exists("LOSER")   ; a non-existent namespace
```

These methods are described in the InterSystems Class Reference.

For UNIX® systems, the default namespace is established as a System Configuration option. For Windows systems, it is set using a command line start-up option.

Namespace names are not case-sensitive. Caché always displays explicit namespace names in all uppercase letters, and implied namespace names in all lowercase letters.

To obtain the namespace name of a specified process, use a method of the %SYS.ProcessQuery class, as shown in the following example:

```
WRITE ##CLASS(%SYS.ProcessQuery).%OpenId($JOB).NameSpaceGet()
```

Setting the Current Namespace

You can change the current namespace using a ZNSPACE command, a SET $NAMESPACE, or a SET $ZNSPACE.

SET $ZNSPACE is functionally identical to the ZNSPACE command.

Note: When you wish to temporarily change the current namespace, perform some operation, then revert to the prior namespace, use SET $NAMESPACE, rather than SET $ZNSPACE. Because $NAMESPACE permits you to NEW $NAMESPACE, it reverts to the original namespace when either the subroutine completes or an unexpected error occurs. See $NAMESPACE special variable for details.

You can use the SET command to change the value of $ZNSPACE. This sets the current namespace for the process. Specify the new namespace as a string literal or a variable or expression that evaluates to a quoted string. You can specify an explicit namespace ("namespace") or an implied namespace ("^system\dir" or "^^dir").

If you specify the current namespace, SET $ZNSPACE performs no operation and returns no error. If you specify an undefined namespace, SET $ZNSPACE generates a <NAMESPACE> error.

You cannot NEW the $ZNSPACE special variable.

Example

In the following example, if the current namespace is not USER, a SET $ZNSPACE command changes the current namespace to USER. Note that because of the IF test, the namespace must be specified in all uppercase letters.
SET ns="USER"
IF $ZNSPACE=ns {
    WRITE !,"Namespace already was ",$ZNSPACE }
ELSEIF 1=#class(%SYS.Namespace).Exists(ns) {
    WRITE !,"Namespace was ",$ZNSPACE
    SET $ZNSPACE=ns
    WRITE !,"Set namespace to ",$ZNSPACE }
ELSE { WRITE !,ns," is not a defined namespace" }
QUIT

This example requires that UnknownUser have assigned the %DB_CACHESYS and the %DB_USER roles.

See Also

- SET command
- ZNSPACE command
- $NAMESPACE special variable
- Configuring Namespaces in Caché System Administration Guide
$ZORDER

Contains the value of the next global node.

Description

$ZORDER contains the value of the next global node (in $QUERY sequence, not $ORDER sequence), after the current global reference. If there is no next global node, accessing $ZORDER results in an <UNDEFINED> error, indicating the last global successfully accessed by $ZORDER. For further details on <UNDEFINED> errors, refer to $ZERROR.

This special variable cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error.

Example

The following example uses a WHILE loop to repeatedly call $ZORDER to traverse a series of subscript nodes:

```objectscript
SET ^||a="groceries"
SET ^||a(1)="fruit"
SET ^||a(1,1)="apples"
SET ^||a(1,2)="oranges"
SET ^||a(3)="nuts"
SET ^||a(3,1)="peanuts"
SET ^||a(2)="vegetables"
SET ^||a(2,1)="lettuce"
SET ^||a(2,2)="tomatoes"
SET ^||a(2,1,1)="iceberg"
SET ^||a(2,1,2)="romaine"
SET $ZERROR="unset"
WRITE !,"last referenced: ",^||a(1,1)
WHILE $ZERROR="unset" {
   WRITE !,$ZORDER
}
QUIT
```

The above example starts with the last-referenced global (in this case, process-private globals): ^||a(1,1). $ZORDER does not contain the value of ^||a(1,1), but works forward from that point. Calls to $ZORDER traverse the subscript tree nodes in the following order: (1,2), (2), (2,1), (2,1,1), (2,1,2), (2,2), (3), (3,1). Each WRITE $ZORDER displays the data value in each successive node. It then runs out of nodes and generates the following error: <UNDEFINED> ^||a(3,1). Note that ^||a(3,1) is not undefined; it is specified because $ZORDER could not find another global after this one.

See Also

- $ORDER function
- $QUERY function
- $ZORDER function
- $ZERROR special variable
$ZPARENT

Contains the ID of the parent process of the current process.

| $ZPARENT | $ZP |

**Description**

$ZPARENT contains the ID of the parent process that created the current process with the **JOB** command. If the current process was not created with the **JOB** command, $ZPARENT contains 0 (zero).

This special variable cannot be modified using the **SET** command. Attempting to do so results in a **<SYNTAX>** error.

**See Also**

- **JOB** command
- **$ZCHILD** special variable
- **$JOB** special variable
$ZPI

Contains the value of pi.

Description
$ZPI contains the value of the numeric constant Pi to eighteen decimal places (3.141592653589793238).

This value is frequently used with trigonometric functions, such as the sine function $ZSIN.
Contains the current file position during the reading of a sequential file.

### Description

$ZPOS$ contains the current file position during sequential file reads. If no sequential file read is in progress, $ZPOS$ contains 0 (zero).

When you open a file for sequential reads, each READ from that device sets $ZPOS$ to the position within the file at which the next read will occur. The $ZPOS$ value is the actual file offset in bytes at the conclusion of a READ, READ *, or READ #n. The user must take appropriate care when reading multi-byte character sets.

The current file position can be set using the $ZSEEK$ function. This special variable cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error.

### See Also

- READ command
- $ZSEEK$ function
- Sequential File I/O in Caché I/O Device Guide
$ZREFERENCE

Contains the current global reference.

$ZREFERENCE
$ZR

Description

$ZREFERENCE contains the name and subscript(s) of the last global reference. This is known as the naked indicator.

Note: The last global reference is the most recently accessed global node. Usually, this is the most recent explicit reference to a global. However, certain commands may internally use the $ORDER function to traverse global subscripts (the ZWRITE command is an example of this), or they may refer internally to some other global. When this occurs, $ZREFERENCE contains the last accessed global node, which may not be the global node specified to the command.

The last global reference can be either a global (^myglob) or a process-private global (^||myppg). $ZREFERENCE returns the process-private global prefix in the form that was initially used for that variable, regardless of which process-private global prefix is used subsequently for that variable. In the description of $ZREFERENCE that follows, the word “global” refers to both types of variables.

The last global reference is the global most recently referred to by a command or a function. Because ObjectScript performs operations in left-to-right order, the last global reference is always the rightmost global. When a command or function takes multiple arguments, the global specified in the rightmost argument is the last global reference. When an argument contains multiple global references, the rightmost specified global is the last global reference. This strict left-to-right order holds true even if parentheses are used to define the order of operations.

Caché updates $ZREFERENCE when an explicit global reference is issued. Invoking an expression (such as a local variable) that evaluates to a global reference does not update $ZREFERENCE.

$ZREFERENCE contains the most recent global reference, even if this global reference was not successful. When a command references an undefined global, issuing an <UNDEFINED> error, Caché updates $ZREFERENCE to that global reference, just as if the global were defined. This behavior is unaffected by setting the %SYSTEM.Process.Undefine() method.

$ZREFERENCE often contains the most recent global reference, even if the command execution was not successful. Caché updates $ZREFERENCE as each global is referenced. For example, a command issuing a <DIVIDE> error (attempting to divide a number by 0) updates $ZREFERENCE to the last global referenced in the command before the error occurred. However, a <SYNTAX> error does not update $ZREFERENCE.

Long Global Names

If a global name is longer than 31 characters (excluding global prefix character(s), such as ^), $ZREFERENCE returns the global name shortened to 31 characters. For further information on the handling of long global names, refer to the Global Variables section of the “Variables” chapter of Using Caché ObjectScript.

Naked Global Reference

If the last global reference was a naked global reference, $ZREFERENCE contains the external, readable, full form of the current naked global reference. This is demonstrated in the following example:

```
SET "MyData(1)="fruit"
SET ^MyData(1,1)="apples" ; Full global reference
SET ^2="oranges"        ; Naked global reference,
                    ; implicitly ^MyData(1,2)
WRITE !,$ZREFERENCE     ; Returns "^MyData(1,2)"
```
For further details on naked global references, see Using Caché Multidimensional Storage (Globals) in Using Caché Globals.

**Extended Global Reference**

Extended global reference is used to reference a global that is in a namespace other than the current namespace. If a command references a global variable using an extended global reference, the $ZREFERENCE value contains that extended global reference. Caché returns an extended global reference in the following circumstances:

- If the last global reference uses an extended reference to refer to a global in another namespace.
- If the last global reference uses an extended reference to refer to a global in the current namespace.
- If the last global reference is a remote reference (a global on a remote system).

In all cases, $ZREFERENCE returns the namespace name in all capital letters, regardless of how it was specified in the global reference.

For further details on global subscripts and extended global references, see Global Structure in Using Caché Globals.

**Operations that Update $ZREFERENCE**

The $ZREFERENCE special variable is initialized to the null string ("""). Changing the current namespace resets $ZREFERENCE to the null string.

The following operations set $ZREFERENCE to the most recently referenced global:

- A command or function that uses a global as an argument. If it uses multiple globals, $ZREFERENCE is set to the rightmost occurrence of a global. (Note, however, the exception of $ORDER.)
- A command that uses a global as a postconditional expression.
- Following a ZWRITE, Caché sets $ZREFERENCE to the last accessed subscript node of the specified global reference.
- A command or function that references an undefined global, and either generates an <UNDEFINED> error, or, in the case of $INCREMENT, defines the global.

**Setting $ZREFERENCE**

You can set this special variable using the SET command, as follows:

- Set to the null string ("""). Doing so deletes the naked indicator. If the next global reference is a naked global reference, Caché issues a <NAKED> error.
- Set to a valid global reference (defined or undefined). This causes subsequent naked references to use the value you set as if it were the last actual global reference.

$ZREFERENCE cannot be otherwise modified using the SET command. Attempting to do so results in a <SYNTAX> error.

**Examples**

The following example returns the last global reference:

```objectscript
SET "a(1,1)="Hello" ; Full global reference
SET "(2)=" world!" ; Naked global reference
WRITE $ZREFERENCE
```

returns:

```
"a(1,2)
```
The following example returns global references from several different commands. Note that WRITE and ZWRITE set different representations of the same global reference.

```objectscript
SET (^barney,"betty","wilma","fred")="flintstone"
WRITE !,$ZREFERENCE
KILL ^flies
WRITE !,$ZREFERENCE
WRITE !,"fred"
WRITE !,$ZREFERENCE
ZWRITE ^fred
WRITE !,$ZREFERENCE
```

returns:

```
^fred          ; last of several globals set in left-to-right order
^flies         ; KILL sets global indicator, though no global to kill
flintstone                ; WRITE global
"fred"          ; global from WRITE
"fred="flintstone"   ; ZWRITE global
"fred("())     ; global from ZWRITE
```

The following example returns an extended global reference. Note that the namespace name is always returned in uppercase letters:

```objectscript
SET ^['"samples"]a(1,1)="Hello"
SET ^('2)=" world!"
WRITE $ZREFERENCE
QUIT
```

returns:

```
^['"SAMPLES"]a(1,2)
```

The following example returns the last global reference. In this case, it is ^a(1), used as an argument to the $LENGTH function:

```objectscript
SET ^a(1)="abcdefghijklmnopqrstuvwxyz"
SET ^b(1)="1234567890"
SET x=$LENGTH(^a(1))
WRITE $ZREFERENCE
QUIT
```

The following example returns the value set for $ZREFERENCE as if it were the last global reference. In this case, it is ^a(1,1).

```objectscript
SET ^a(1,1)="abcdefghijklmnopqrstuvwxyz"
SET ^b(1,1)="1234567890"
WRITE !,"(1)
; Naked reference uses last global
SET $ZREFERENCE="^a(1,1)"
WRITE !,$ZREFERENCE
WRITE !,"(1)
; Naked reference uses $ZREFERENCE
; value, rather than last global.
```

The following example sets an extended global reference. Note the doubled quotation marks:

```objectscript
KILL ^x
WRITE !,$ZREFERENCE
SET $ZREFERENCE="^['"samples"]a(1,2)"
WRITE !,$ZREFERENCE
```

See Also

- ZNSPACE command
- $NAMESPACE special variable
- $ZNAMESPACE special variable
- Configuring Namespaces in Caché System Administration Guide
• Global Structure in *Using Caché Globals*
• Naked Global Reference in *Using Caché Globals*
$ZSTORAGE

Contains the maximum available memory for a process.

| $ZSTORAGE |
| $ZS |

**Description**

$ZSTORAGE contains the maximum amount of memory, in Kbytes, for a job’s process-private memory. This memory is available for local variables, stacks, and other tables. This memory limit does not include space for the routine object code. This memory is allocated to the process as needed, for example when allocating an array.

Once this memory is allocated to the process, it is generally not deallocated until the process exits. However, when a large amount of memory is used (for example greater than 32MB) and then freed, Caché attempts to release deallocated memory back to the operating system, where possible.

You can also use $ZSTORAGE to set the maximum memory size. For example, the following statement sets the job’s maximum process-private memory to 524288 Kbytes:

```
SET $ZSTORAGE=524288
```

Changing $ZSTORAGE changes the initial value for the $STORAGE special variable, which contains the current available memory (in bytes) for the process.

The maximum value for $ZSTORAGE is 2147483647. The $ZSTORAGE default is 262144. The minimum value for $ZSTORAGE is 128. $ZSTORAGE values larger than the maximum or smaller than the minimum automatically default to the maximum or minimum value. $ZSTORAGE is set to an integer value; Caché truncates any fractional portion (if specified).

You can change the $ZSTORAGE default by changing the Maximum per Process Memory (KB) system configuration setting. In the Management Portal, go to [System] > [Configuration] > [Memory and Startup]. You can increase Maximum per Process Memory (KB) as desired, up to a maximum of 2147483647 Kbytes. Changing Maximum per Process Memory (KB) changes the $ZSTORAGE value for subsequently initiated processes; it has no effect on the $ZSTORAGE value for current processes.

**Example**

The following example sets $ZSTORAGE to its maximum and minimum values. Attempting to set $ZSTORAGE to a value (16) that is less than the minimum value automatically sets $ZSTORAGE to its minimum value (128):

```
SET $ZS=128
WRITE "minimum storage=",$ZS,!
SET $ZS=16
WRITE "less than minimum storage=",$ZS,!
SET $ZS=2147483647
WRITE "maximum storage=",$ZS,!
```

**See Also**

SET command

KILL command

$STORAGE special variable
$ZTIMESTAMP

Contains the current date and time in Coordinated Universal Time format.

### Description

$ZTIMESTAMP contains the current date and time as a Coordinated Universal Time value. This is a worldwide time and date standard; this value is very likely to differ from your local time (and date) value.

$ZTIMESTAMP represents date and time as a string with the format:

```
dddddd,sssss.fff
```

Where `ddddd` is an integer specifying the number of days since December 31, 1840; `sssss` is an integer specifying the number of seconds since midnight of the current day, and `fff` is a varying number of digits specifying fractional seconds. This format is similar to $HOROLOG, except that $HOROLOG does not contain fractional seconds.

Suppose the current date and time (UTC) is as follows:

```
2011-03-26 15:17:27.984
```

At that time, $ZTIMESTAMP has the value:

```
62176,55047.984
```

$ZTIMESTAMP reports the Coordinated Universal Time (UTC), which is independent of time zone. Consequently, $ZTIMESTAMP provides a time stamp that is uniform across time zones. This may differ from both the local time value and the local date value.

The $ZTIMESTAMP time value is a decimal numeric value that counts the time in seconds and fractions thereof. The number of digits in the fractional seconds may vary from zero to nine, depending on the precision of your computer's time-of-day clock. On Windows systems the fractional precision is three decimal digits; on UNIX® systems it is six decimal digits. $ZTIMESTAMP suppresses trailing zeroes or a trailing decimal point in this fractional portion. Note that within the first second after midnight, seconds are represented as `0.fff` (for example, `0.123`); this number is not in ObjectScript canonical form (for example, `.123`), which affects the string sorting order of these values. You can prepend a plus sign (+) to force conversion of a number to canonical form before performing a sort operation.

The various ways to return the current date and time are compared, as follows:

- **$ZTIMESTAMP** contains the UTC date and time, with fractional seconds, in Caché storage ($HOROLOG) format. Fractional seconds are expressed in three digits of precision (on Windows systems), or six digits of precision (on UNIX® systems).

- **$NOW** returns the local date and time for the current process; local time variants (such as Daylight Saving Time) are **not** applied. $NOW with no parameter value determines the local time zone from the value of the $ZTIMEZONE special variable. $NOW with a parameter value returns the time and date that correspond to the specified time zone parameter. $NOW(0) returns the UTC date and time. The value of $ZTIMEZONE is ignored. $NOW returns the date and time in Caché storage ($HOROLOG) format. It includes fractional seconds; the number of fractional digits is the maximum precision supported by the current operating system. For this reason, $NOW(0) may return UTC time with greater fractional second precision than $ZTIMESTAMP.

- **$HOROLOG** contains the local, variant-adjusted date and time in Caché storage format. It does not record fractional seconds. How $HOROLOG resolves fractional seconds depends on the operating system platform: On Windows, it rounds up any fractional second to the next whole second. On UNIX®, it truncates the fractional portion.
Note: Exercise caution when comparing local time and UTC time:

- The preferred way to convert UTC time to local time is to use the $ZDATETIMEH(utc,-3) function. This function adjusts for local time variants.

- You cannot interconvert local time and UTC time by simply adding or subtracting the value of $ZTIMEZONE * 60. This is because, in many cases, local time is adjusted for local time variants (such as Daylight Saving Time, which seasonally adjusts local time by one hour). These local time variants are not reflected in $ZTIMEZONE.

- UTC time is calculated using a count of time zones from the Greenwich meridian. It is not the same as local Greenwich time. The term Greenwich Mean Time (GMT) may be confusing; local time at Greenwich is the same as UTC during the winter; during the summer it differs from UTC by one hour. This is because a local time variant, known as British Summer Time, is applied.

- Both the timezone offset from UTC and local time variants (such as the seasonal shift to Daylight Saving Time) can affect the date as well as the time. Converting from local time to UTC time (or vice versa) may change the date as well as the time.

This special variable cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error.

**Coordinated Universal Time Conversions**

You can represent local time information as Coordinated Universal Time (UTC) using the $ZDATETIME and $ZDATETIMEH functions with tformat values 7 or 8, as shown in the following example:

```objectscript
WRITE !,$ZDATETIME($ZTIMESTAMP,1,1,2)
WRITE !,$ZDATETIME($HOROLOG,1,7,2)
WRITE !,$ZDATETIME($HOROLOG,1,8,2)
WRITE !,$ZDATETIME($NOW(),1,7,2)
WRITE !,$ZDATETIME($NOW(),1,8,2)
```

The above $ZDATETIME functions all return the current time as Coordinated Universal Time (rather than local time). These time value conversions from local time may differ, because $NOW does not adjust for local time variants; $ZTIMESTAMP and $HOROLOG do adjust for local time variants and may adjust the date accordingly, if necessary. The $ZTIMESTAMP display value and the tformat 7 or 8 converted display values are not identical. The tformat values 7 and 8 insert the letter “T” before, and the letter “Z” after the time value. Also, because $HOROLOG time does not contain fractional seconds, the precision of 2 in the above example pads those decimal digits with zeros.

You can obtain the same time stamp information as $ZTIMESTAMP by invoking the TimeStamp() class method, using either of the following syntax forms:

```objectscript
WRITE !,$SYSTEM.SYS.TimeStamp()
WRITE 1,##class(%SYSTEM.SYS).TimeStamp()
```

Refer to the $SYSTEM special variable in this manual and the %SYSTEM.SYS class in the InterSystems Class Reference for further details.

**Examples**

The following example converts the value of $ZTIMESTAMP to local time, and compares it with two representations of local time: $NOW() and $HOROLOG:
The following example compares the values returned by $ZTIMESTAMP and $HOROLOG, and shows how the time portion of $ZTIMESTAMP may be converted. (Note that in this simple example only one adjustment is made for local time variations, such as Daylight Saving Time. Other types of local variation may cause clocksecs and stampsecs to contain irreconcilable values.)

```objectscript
SET stamp=$ZTIMESTAMP,clock=$HOROLOG,miliclock=$NOW()
WRITE !,"local date and time: ",$ZDATETIME(clock,1,1,2)
WRITE !,"UTC date and time: ",$ZDATETIME(stamp,1,1,2)
IF $PIECE(stamp,"",2) = $PIECE(clock,"",2) {
    WRITE !,"Time difference affects date" }
ELSEIF $PIECE(stamp,"") = $PIECE(clock,"") {
    WRITE !,"Time difference affects date" }
ELSE {
    SET localutc=$ZDATETIMEH(stamp,-3)
    WRITE !,"UTC converted to local: ",$ZDATETIME(localutc,1,1,2)
}
QUIT
```

See Also

- $NOW function
- $ZDATETIME function
- $ZDATETIMEH function
- $HOROLOG special variable
- $ZTIMEZONE special variable
$ZTIMEZONE

Contains the time zone offset from the Greenwich meridian.

| $ZTIMEZONE | $ZTZ |

## Description

$ZTIMEZONE can be used in two ways:

- To return the local time zone offset for your computer.
- To set the local time zone offset for the current process.

$ZTIMEZONE contains the time zone offset from the Greenwich meridian in minutes. (The Greenwich meridian includes all of Great Britain and Ireland.) This offset is expressed as a signed integer in the range -1440 to 1440. Time zones west of Greenwich are specified as positive numbers; time zones east of Greenwich are specified as negative numbers. (Time zones must be expressed in minutes, because not all time zone differences are in whole hours.) By default, $ZTIMEZONE is initialized to the time zone set for the computer’s operating system.

**CAUTION:** $ZTIMEZONE adjusts the local time by a fixed offset. It does not adjust for Daylight Saving Time or other local time variants. Caché takes its local time from the underlying operating system, which applies the local time variant for the location configured for that computer. Therefore, a local time adjusted using $ZTIMEZONE takes its local time variation from the configured locale, not the time zone specified in $ZTIMEZONE.

UTC time is calculated using a count of time zones from the Greenwich meridian ($ZTIMEZONE=0). It is not the same as local Greenwich time. The term Greenwich Mean Time (GMT) may be confusing; local time at Greenwich is the same as UTC during the winter; during the summer it differs from UTC by one hour. This is because a local time variant, known as British Summer Time, is applied.

For functions and programs that use $ZTIMEZONE, elapsed local time is always continuous, but the time value may need to be seasonally adjusted to correspond to local clock time. For more details on local seasonal time variants, refer to $HOROLOG.

## Setting the Time Zone

You can use $ZTIMEZONE to set the time zone used by the current Caché process. Setting $ZTIMEZONE does not change the default Caché time zone or your computer’s time zone setting.

**Note:** Changing the $ZTIMEZONE special variable is a feature designed for some special situations. Changing $ZTIMEZONE is not a consistent way to change the time zone that Caché uses for local date/time operations. The $ZTIMEZONE special variable should not be changed except by those programs that are prepared to handle all the inconsistencies that result.

On some platforms there may be a better way to change time zones than changing the $ZTIMEZONE special variable. If the platform has a process-specific time zone setting (for example, the TZ environment variable on POSIX systems) then making an external system call to change the process-specific time zone may work better than changing $ZTIMEZONE. Changing the process-specific time zone at the operating system level will change both the local time offset from UTC and apply the corresponding algorithm that determines when local time variants are applied. This is especially important if the default system time zone is in the Northern Hemisphere, while the desired process time zone is in the Southern Hemisphere. Changing $ZTIMEZONE changes the local time to a new time zone offset from UTC, but the algorithm that determines when local time variants are applied remains unchanged.
Use the SET command to set $ZTIMEZONE to a specified signed integer number of minutes. Leading zeros and decimal portions of numbers are ignored. If you specify a nonnumeric value or no value when setting $ZTIMEZONE, Caché sets $ZTIMEZONE to 0 (Greenwich meridian).

For example, North American Eastern Standard Time (EST) is five hours west of Greenwich. Therefore, to set the current Caché process to EST you would specify 300 minutes. To specify a time zone one hour east of Greenwich, you would specify –60 minutes. To specify Greenwich itself, you would specify 0 minutes.

Setting $ZTIMEZONE:

• affects the argumentless $NOW() local time value. It changes the time portion of $NOW(), and this change of time can also change the date portion of $NOW() for the current process. $NOW() reflects the $ZTIMEZONE setting exactly, its value is not adjusted for local time variants.

• affects the $HOROLOG local time value. $HOROLOG takes its time zone value from $ZTIMEZONE, then seasonally adjusts for local time variants such as Daylight Saving Time. $HOROLOG therefore always conforms to local clock time, but $HOROLOG elapsed time is not continuous throughout the year.

• affects the %SYSTEM.Util class methods IsDST(), UTCtoLocalWithZTIMEZONE(), and LocalWithZTIMEZONEtoUTC().

• does not affect $ZTIMESTAMP or $ZHOROLOG values.

• does not affect the date and time format conversions performed by $ZDATE, $ZDATEH, $ZDATETIME, $ZDATETIMEH, $ZTIME, and $ZTIMEH functions.

• does not affect the $NOW(n) function.

• does not affect the FixedDate() class method of the %SYSTEM.Process class, which sets the date in $HOROLOG to a fixed value.

The following anomalies occur after changing $ZTIMEZONE:

• $ZDATETIME($HOROLOG,1,7) usually returns an UTC time, but it will not return UTC time if $ZTIMEZONE has changed.

• $ZDATETIME($HOROLOG,1,5) will not return the correct time zone offset if $ZTIMEZONE has changed.

• $ZDATETIME($HOROLOG,-3) and $ZDATETIMEH($ZTIMESTAMP,-3) conversions between Local time and UTC time will not be correct if $ZTIMEZONE has changed.

Other Time Zone Methods

You can obtain the same time zone information by invoking the TimeZone() class method, as follows:

WRITE $SYSTEM.SYS.TimeZone()

Refer to the %SYSTEM.SYS class in the InterSystems Class Reference for further details.

You can return your local time variation as part of a date and time string by using the $ZDATETIME and $ZDATETIMEH functions with tformat values 5 or 6, as shown in the following example:

WRITE !,$ZDATETIME($HOROLOG,1,5)

This returns a value such as:

04/06/2011T12:31:16-04:00

The last part of this string (–04:00) indicates the system’s local time variation setting as an offset in hours and minutes from the Greenwich meridian. Note that this variation is not necessarily the time zone offset. In the above case, the time zone is 5 hours west of Greenwich (–5:00), but the local time variant (Daylight Saving Time) offsets the time zone time
by one hour to -04:00. Setting $ZTIMEZONE changes the current process date and time returned by $ZDATETIME($HOROLOG,1,5), but does not change the system local time variation setting.

$ZDATETIME Uses Time Zone Setting

You can use $ZDATETIME with dformat=3 to convert a Coordinated Universal Time (UTC) date and time value to a local time. This function takes as input a UTC value ($ZTIMESTAMP). It uses the local time zone setting to return the corresponding date and time with local time variants (such as Daylight Saving Time) applied where applicable.

```
SET clock=$HOROLOG
SET stamp=$ZDATETIME($ZTIMESTAMP,-3)
WRITE !,*local/local date and time:  ",$ZDATETIME(clock,1,1,2)
WRITE !,*UTC/local date and time:  ",$ZDATETIME(stamp,1,1,2)
```

Local/UTC Conversion Methods Using $ZTIMEZONE

Two class methods of the %SYSTEM.Util class convert between local date and time and UTC date and time: UTCtoLocalWith$ZTIMEZONE() and LocalWith$ZTIMEZONEtoUTC(). These methods are affected by changes to $ZTIMEZONE.

```
WRITE $SYSTEM.Util.UTCtoLocalWith$ZTIMEZONE($ZTIMESTAMP),!
WRITE $HOROLOG,!
WRITE $SYSTEM.Util.LocalWith$ZTIMEZONEtoUTC($H),!
WRITE $ZTIMEZONE
```

Examples

The following example returns your current time zone:

```
SET zone=$ZTIMEZONE
IF zone=0 {
  WRITE !,"Your time zone is Greenwich Mean Time" }
ELSEIF zone>0 {
  WRITE !,"Your time zone is ",zone/60," hours west of Greenwich" }
ELSE {
  WRITE !,"Your time zone is ",(-zone)/60," hours east of Greenwich" }
```

The following example shows that setting the time zone can change the date as well as the time:

```
SET zonesave=$ZTIMEZONE
WRITE !,"Date in my current time zone:  ",$ZDATE($HOROLOG)
IF $ZTIMEZONE=0 {
  SET $ZTIMEZONE=720 }
ELSEIF $ZTIMEZONE>0 {
  SET $ZTIMEZONE=($ZTIMEZONE-720) }
ELSE {
  SET $ZTIMEZONE=($ZTIMEZONE+720) }
WRITE !,"Date halfway around the world:  ",$ZDATE($HOROLOG)
WRITE !,"Date at Greenwich Observatory:  ",$ZDATE($ZTIMESTAMP)
SET $ZTIMEZONE=zonesave
```

The following example determines if your local time is the same as your time zone time:

```
SET localnow=$HOROLOG, stamp=$ZTIMESTAMP
WRITE !,"local date and time:  ",$ZDATETIME(localnow,1,1)
SET clocksecs=$PIECE(localnow,",",2)
SET stampsecs=$EXTRACT(stamp,7,11)-($ZTIMEZONE*60)
IF clocksecs=stampsecs {
  WRITE !,"No local time variant:" 
  WRITE !,"Local time is timezone time"
}
ELSE {
  IF clocksecs=stampsecs+3600 {
    WRITE !,"Daylight Saving Time variant:" 
    WRITE !,"Local time offset 1 hour from timezone time" 
  }
  ELSE {WRITE !,"Local time and time zone time are ":
    WRITE !,(clocksecs-stampsecs)/60," minutes different"
  }
}
QUIT
See Also

- $HOROLOG special variable
- $ZTIMESTAMP special variable
$ZTRAP

Contains the location of the current error trap handler.

Description

$ZTRAP contains the location of the current error trap handler. For a description of how to code an error trap handler, refer to “Handling Errors with $ZTRAP” in the “Error Processing” chapter of Using Caché ObjectScript.

- In a routine, $ZTRAP can reference a label within the routine or within another routine. It contains the label name and the routine name. For example, if you set $ZTRAP to the label MyHandler within the routine MyRou, the $ZTRAP special variable contains MyHandler^MyRou.

- In a procedure, $ZTRAP must reference a label within that procedure. It contains the line offset of the label from the procedure name. For example, if you set $ZTRAP to the private label MyHandler within the procedure MyProc, the $ZTRAP special variable contains +17^MyProc.

There are three ways to set $ZTRAP:

- SET $ZTRAP="location"
- SET $ZTRAP="^%ETN" (or "^%ET")

You cannot KILL $ZTRAP; attempting to do so results in a <SYNTAX> error. To eliminate the current $ZTRAP value, specify $ZTRAP="".

Location

Using the SET command you can specify location as a quoted string.

- In a routine, you can specify location as label (a line label in the current routine), ^routine (the beginning of a specified external routine), or label^routine (a specified label in a specified external routine). Do not specify a location in a routine that references a procedure or a label within a procedure. This is an invalid location; it results in a runtime error when Caché attempts to execute $ZTRAP.

- In a procedure, you can specify location as label; a private label within that procedure block. $ZTRAP in a procedure block cannot be used to go to a location that is outside the body of the procedure; $ZTRAP in a procedure block can only reference a location within that procedure block. Therefore, in a procedure you cannot set $ZTRAP to ^routine or label^routine. Attempted to do so results in a <SYNTAX> error.

In a procedure, you set $ZTRAP to a private label name, but the $ZTRAP value is not the private label name; it is the offset from the procedure label (the top of the procedure) to the line location of the private label. For example, +17^myproc.

Note: $ZTRAP provides legacy support for label+offset in some contexts (but not in procedures). This optional +offset is an integer specifying the number of lines to offset from label. The label must be in the same routine. The use of +offset is deprecated, and may result in a compilation warning error. InterSystems recommends that you avoid the use of a line offset when specifying location.

You cannot specify an +offset when calling a procedure or a CACHESYS % routine. If you attempt to do so, Caché issues a <NOLINE> error.

The $ZTRAP location must be in the current namespace. $ZTRAP does not support extended routine reference.
If you specify a nonexistent line label, a *location* that does not exist in the current routine, the following occurs:

- **Displaying $ZTRAP:** In a routine, $ZTRAP contains *label*routine. For example, DummyLabel^MyRou. In a procedure, $ZTRAP contains the maximum possible offset: +34463^MyProc.

- **Invoking $ZTRAP:** Caché issues a *<NOLINE>* error message.

Each stack level can have its own $ZTRAP value. When you set $ZTRAP, the system saves the value of $ZTRAP for the previous stack level. Caché restores this value when the current stack level ends. To enable error trapping at the current stack level, set $ZTRAP to the error trap handler by specifying its *location*. For example:

```objectscript
IF $ZTRAP="" {WRITE !,"$ZTRAP not set" }
ELSE {WRITE !,"$ZTRAP already set: ",$ZTRAP
    SET oldtrap=$ZTRAP }
SET $ZTRAP="Etrap1^Handler"
WRITE !,"$ZTRAP set to: ",$ZTRAP
// program code
SET $ZTRAP=oldtrap
WRITE !,"$ZTRAP restored to: ",$ZTRAP
```

When an error occurs, this format unwinds the call stack and transfers control to the specified error trap handler.

In SQLComputeCode, do not set $ZTRAP=$ZTRAP. This can result in significant problems with transactional processing and error reporting.

To disable error trapping, set $ZTRAP to the null string (""). This clears any error trap set at the current DO stack level.

When you set an error handler using $ZTRAP, this handler takes precedence over any existing $ETRAP error handler. Caché implicitly performs a **NEW $ETRAP** command and sets $ETRAP to the null string ("").

**Note:** Use of $ZTRAP from the Terminal prompt is limited to the current line of code. The **SET $ZTRAP** command and the command generating the error must be in the same line of code. Terminal restores $ZTRAP to the system default at the beginning of each command line.

**Location**

In a routine, you can choose to leave the call stack as it is after the error has occurred. To do so, place an asterisk (*) before *location* and within the double quotes. This form is not valid for use within procedures; attempted to do so results in a *<SYNTAX>* error. It can only be used in subroutines that are not procedures, as in this example:

```objectscript
Main
    SET $ZTRAP="*OnError"
    WRITE !,"$ZTRAP set to: ",$ZTRAP
    // program code
OnError
    /// Error handling code
    QUIT
```

This format simply causes a *GOTO* to the line label specified in $ZTRAP; $STACK and $ESTACK are unchanged. The context frame of the $ZTRAP error handling routine is the same as the context frame where the error occurred. However, Caché resets $ROLES to the value that was in effect for the execution level at which $ZTRAP was set; this prevents the $ZTRAP error handler from using elevated privileges that were granted to the routine after establishing the error handler. Upon completion of the $ZTRAP error handling routine, Caché unwinds the stack to the previous context level. This form of $ZTRAP is especially useful for analyzing unexpected errors.

Note that the asterisk sets a $ZTRAP option; it is not part of the *location*. For this reason, this asterisk does not display when performing a **WRITE** or **ZZDUMP** on $ZTRAP.

**^%ETN**

In a routine, **SET $ZTRAP="^%ETN"** establishes the system-supplied error routine %ETN as the current error trap handler. %ETN executes in the context in which the error occurred that invoked it. (%ET is a legacy name for %ETN.)
Their functionality is identical, though %ETN is slightly more efficient.) The error handlers ^%ETN and ^%ET always behave as if they were preceded by an asterisk (*).

Because $ZTRAP in a procedure block cannot be used to go to a location that is outside the body of the procedure, SET $ZTRAP="^%ETN" cannot be used in a procedure. Attempting to do so results in a <SYNTAX> error.

For more information on %ETN and its entrypoints FORE^%ETN, BACK^%ETN, and LOG^%ETN, see the “Logging Application Errors” section of the “Error Processing” chapter of Using Caché ObjectScript.

TRY / CATCH and $ZTRAP

You cannot set $ZTRAP within a TRY block. Attempting to do so generates a compilation error. You can set $ZTRAP prior to the TRY block, or within the CATCH block.

An error that occurs within a TRY block is handled by the CATCH block, regardless of whether a $ZTRAP has previously been set. An error that occurs within a CATCH block is handled by the current error trap handler.

The first of the following examples shows an error occurring in a TRY block. The second of the following examples shows an exception being thrown in a TRY block. In both cases, the CATCH block, not the $ZTRAP, is taken:

```ObjectScript
SET $ZTRAP="Ztrap"
TRY { WRITE 1/0 }    /* divide-by-zero error */
   CATCH { WRITE "Catch taken" }  
   QUIT
Ztrap
   WRITE "$ZTRAP taken"
   SET $ZTRAP=""
   QUIT

SET $ZTRAP="Ztrap"
TRY { SET myvar=#class(Sample.MyException).%New("Example Error",999,,errdatazero)
   WRITE ","Throwing an exception!",
     THROW myvar
   QUIT }
   CATCH { WRITE "Catch taken" }  
   QUIT
Ztrap
   WRITE "$ZTRAP taken"
   SET $ZTRAP=""
   QUIT
```

However, a TRY block can call code that sets and uses $ZTRAP. In the following example, the divide-by-zero error is caught by $ZTRAP, rather than the CATCH block:

```ObjectScript
TRY { DO Errsub }   
   CATCH { WRITE "Catch taken" }  
   QUIT
Errsub
   SET $ZTRAP="Ztrap"
   WRITE 1/0    /* divide-by-zero error */
   QUIT
Ztrap
   WRITE "$ZTRAP taken"
   SET $ZTRAP=""
   QUIT
```

A THROW command from a CATCH block can also invoke a $ZTRAP error handler.

Examples

The following example sets $ZTRAP to the OnError routine in this program. It then calls SubA in which an error occurs (attempting to divide a number by 0). When the error occurs, Caché calls the OnError routine specified in $ZTRAP. OnError is invoked at the context level at which $ZTRAP was set. Because OnError is at the same context level as Main, execution does not return to Main.
The following example is identical to the previous example, with one exception: The $ZTRAP \textit{location} is prefaced by an asterisk (*). When the error occurs in SubA, this asterisk causes Caché to call the OnError routine at the context level of SubA (where the error occurred), not at the context level of Main (where $ZTRAP was set). Therefore, when OnError completes, execution returns to Main at the line following the DO command.

See Also

- THROW command
- ZQUIT command
- ZINSERT command
- $ETRAP special variable
- $ECODE special variable
- $ESTACK special variable
- $STACK special variable
- Error Processing in \textit{Using Caché ObjectScript}
$ZVERSION

Contains a string describing the current version of Caché.

$ZVERSION
$ZV

Description

$ZVERSION contains a string showing the version of the currently running Caché system.

The following example returns the $ZVERSION string:

    WRITE $ZVERSION

This returns a version string such as the following:

Cache for Windows (x86-32) 2010.2.3 (Build 702U) Tue Feb 15 2011 14:14:23 EST

This string includes the type of Caché installation (product and platform, including CPU type), the version number (2010.2.3), the build number within that version (the “U” in the build number indicates a Unicode version), and the date and time that this version of Caché was created. “EST” is Eastern Standard Time (the time zone for the Eastern United States), “EDT” is Eastern Daylight Saving Time (see Daylight Saving Time for details).

The same information can be viewed by going to the Caché Cube and selecting “About...” or by invoking the GetVersion() class method, as follows:

    WRITE $SYSTEM.Version.GetVersion()

You can get the component parts of this version string by invoking other %SYSTEM.Version methods, which you can list by invoking:

    DO $SYSTEM.Version.Help()

The $ZVERSION special variable cannot be modified using the SET command. Attempting to do so results in a <SYNTAX> error.

Example

The following example extracts the create date from the version string to calculate how old the current version of Caché is, in days. Note that this example is specific to Windows platforms:

    SET createdate=$PIECE($ZVERSION," ",9,11)
    WRITE !,"Creation date: ",createdate
    WRITE !,"Current date: ",$ZDATE($HOROLOG,6)
    SET nowcount=$PIECE($HOROLOG," ",)
    SET thencount=$ZDATEH(createdate,6)
    WRITE !,"This version is ",(nowcount-thencount)," days old"

The following example performs the same operation by calling a class method:

    SET createdate=$SYSTEM.Version.GetBuildDate()
    WRITE !,"Creation date: ",$ZDATE(createdate,6)
    WRITE !,"Current date: ",$ZDATE($HOROLOG,6)
    SET nowcount=$PIECE($HOROLOG," ",)
    WRITE !,"This version is ",(nowcount-createdate)," days old"

See Also

• $ZVERSION(1) function
• $SYSTEM special variable
Structured System Variables

A structured system variable name, or SSVN, is a nonscalar system variable that is organized like a global variable. SSVNs allow you to write portable programs that can retrieve information about system data. The same ObjectScript code can retrieve system data information from any Caché implementation using structured system variables.

Each SSVN has a structure where subscript values are any of the following:

- Entity identifiers
- Literals
- Attribute keywords.

You provide information about entities by providing values for subscripts which are identifiers and for attribute nodes. SSVNs use the caret and dollar sign (^$) as a standard prefix followed by:

1. An optional (extended syntax) specification of the namespace about which you want information
2. One of a designated list of names.

You then follow the name with one or more expressions in parentheses. These expressions are called subscripts. The syntax is as follows:

```
^$[ | namespace | ] ssvn_name (expression)
```

Generally, you cannot use SET and KILL commands for structured system variables because they do not necessarily have data values. The information that structured system variables provide is often the existence of specific subscript values. In most cases, you can use the $DATA, the $ORDER, and the $QUERY functions to examine subscript values.

Caché supports the following structured system variables:

- ^$GLOBAL
- ^$JOB
- ^$LOCK
- ^$ROUTINE

The meaning of each of these SSVNs and use of their subscripts is explained in the following sections.

Each description identifies which functions are allowed with the particular structured system variable. Each description contains one or more examples about how to use structured system variable as arguments to the $DATA, $ORDER, and $QUERY functions to scan system table information.
Structured System Variables

**^$GLOBAL**

Provides information about globals.

\[ ^\$| nspace \ GOLBAL(global_name) \]
\[ ^\$| nspace \ G(global_name) \]

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nspace</td>
<td>Optional — An extended SSVN reference, either an explicit namespace name or an implied namespace. Must evaluate to a quoted string, which is enclosed in either square brackets (&quot;nspace&quot;) or vertical bars (</td>
</tr>
<tr>
<td>global_name</td>
<td>An expression that evaluates to a string containing an unsubscripted global name.</td>
</tr>
</tbody>
</table>

**Description**

You can use ^$GLOBAL as an argument to the $DATA, $ORDER, and $QUERY functions to get information about the existence of global variables in the current namespace (the default), or in a specified namespace.

Note: ^$GLOBAL does not support global_name as a process-private global. Attempting to specify global_name as a process-private global results in a <NAME> error.

For more information on global variables, refer to Using Caché Multidimensional Storage in Using Caché Globals.

**Parameters**

**nspace**

This optional parameter allows you to specify a structured system variable in another namespace by using an extended SSVN reference. You can specify the namespace name either explicitly, as a quoted string literal or as a variable, or by specifying an implied namespace. Namespace names are not case-sensitive. You can use either bracket syntax ["USER"] or environment syntax |"USER"|. No spaces are allowed before or after the nspace delimiters.

You can test whether a namespace is defined by using the following method:

```
WRITE ##class(%SYS.Namespace).Exists("USER"),! ; an existing namespace
WRITE ##class(%SYS.Namespace).Exists("LOSER")   ; a non-existent namespace
```

You can use the $NAMESPACE special variable to determine the current namespace, and the ZNSPACE command to change the current namespace.

**global_name**

An expression that evaluates to a string containing an unsubscripted global name.

**Examples**

The following examples show how to use ^$GLOBAL as an argument to the $DATA, $ORDER, and $QUERY functions.
As an Argument to $DATA

^GLOBAL as an argument to $DATA returns an integer value that signifies whether the global name you specify exists as a ^GLOBAL node. The integer values that $DATA can return are shown in the following table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Global name does not exist</td>
</tr>
<tr>
<td>1</td>
<td>Global name is an existing node with data but has no descendants.</td>
</tr>
<tr>
<td>10</td>
<td>Global name is an existing node with no data but has descendants.</td>
</tr>
<tr>
<td>11</td>
<td>Global name is an existing node with data and has descendants.</td>
</tr>
</tbody>
</table>

The following example test for the existence of a global variable in the user’s default namespace.

```
KILL ^GBL
WRITE $DATA(^GLOBAL("^GBL"))
```
returns 0.

The following example test for the existence of a global variable in the SAMPLES namespace.

```
SET ^GBL(1)="TEST"
WRITE $DATA(^$GLOBAL("^GBL"))
```
returns 10.

```
SET ^GBL(1)="TEST"
WRITE $DATA(^GLOBAL("^GBL"))
```

As an Argument to $ORDER

$ORDER(^$GLOBAL(global_name),direction)

^GLOBAL as an argument to $ORDER returns the next or previous global name in collating sequence to the global name you specify. If no such global name node exists in ^GLOBAL, $ORDER returns a null string.

The direction argument specifies whether to return the next or the previous global name. If you do not provide a direction argument, Caché returns the next global name in collating sequence to the one you specify. For further details, refer to the $ORDER function.

The following subroutine searches the user’s default namespace and stores the global names in a local array named GLOBAL.

```
GLOB
SET NAME=""
WRITE !,"The following globals are in ",ZNSPACE
FOR I=1:1 {
  SET NAME=$ORDER(^GLOBAL(NAME))
  WRITE !,NAME
  QUIT:NAME=""
  SET GLOBAL(I)=NAME
}
WRITE !,"All done"
QUIT
```

As an Argument to $QUERY

^GLOBAL as an argument to $QUERY returns the next global name in collating sequence to the global name you specify. If no such global name exists as a node in ^GLOBAL, $QUERY returns a null string.

In the following example, three globals (^GBL1, ^GBL2 and ^GBL3) are present in the USER namespace.
Structured System Variables

NEW $NAMESPACE
SET $NAMESPACE="USER"
SET (^GBL1,^GBL2,^GBL3)="TEST"
NEW $NAMESPACE
SET $NAMESPACE="\SYS"
WRITE $QUERY(^$|"USER"|GLOBAL("^GBL1")),!
WRITE $QUERY(^$|"USER"|GLOBAL("^GBL2"))
NEW $NAMESPACE
SET $NAMESPACE="USER"
KILL ^GBL1,^GBL2,^GBL3

The first WRITE returns ^$|"USER"|GLOBAL("^GBL2")

The second WRITE returns ^$|"USER"|GLOBAL("^GBL3")

See Also

• $DATA function
• $ORDER function
• $QUERY function
• ZNSPACE command
• $NAMESPACE special variable
• Configuring Namespaces in Caché System Administration Guide
^$JOB

Provides Caché process (job) information.

|^$JOB(job_number)
|^$J(job_number)

Parameter

| job_number | The system-specific job number created when you enter the ObjectScript command. Every active Caché process has a unique job number. Logging in to the system initiates a job. On UNIX® systems, the job number is the pid of the child process started when Caché was invoked. job_number must be specified as an integer; hexadecimal values are not supported. |

Description

You can use the ^$JOB structured system variable as an argument to the $DATA, $ORDER, and $QUERY functions to get information about the existence of Caché jobs on the local Caché system.

Examples

The following examples show how to use ^$JOB as an argument to the $DATA, $ORDER, and $QUERY functions.

As an Argument to $DATA

$DATA(^$JOB(job_number))

^$JOB as an argument to $DATA returns an integer value that indicates whether the specified job exists as a node in ^$JOB. The integer values that $DATA can return are shown in the following table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Job does not exist.</td>
</tr>
<tr>
<td>1</td>
<td>Job exists.</td>
</tr>
</tbody>
</table>

The following example tests for the existence of a Caché process.

```
SET x=$JOB
WRITE 1,$DATA(^$JOB(x))
```

The variable x is set to the job number of the current process (for example: 4294219937). The WRITE returns the boolean 1, indicating this process exists.

As an Argument to $ORDER

$ORDER(^$JOB(job_number),direction)

^$JOB as an argument to $ORDER returns the next or previous ^$JOB job number in collating sequence to the job number you specify. If no such job number exists as a ^$JOB node, $ORDER returns a null string.

The direction argument specifies whether to return the next or the previous job number. If you do not provide a direction argument, Caché returns the next job number in collating sequence to the one you specify. For further details, refer to the $ORDER function.

The following subroutine searches the Caché job table and stores the job numbers in a local array named JOB.
JOB
  SET pid=""
  FOR i=1:1 {
    SET pid=$ORDER('^$JOB(pid))
    QUIT:pid=""
    SET JOB(i)=pid
  }
  WRITE "Total Jobs in Job Table: ",i
QUIT

As an Argument to $QUERY

$QUERY('^$JOB(job_number))

^$JOB as an argument to $QUERY returns the next ^$JOB job number in collating sequence to the job number you specify. If no such job number exists as a node in ^$JOB, $QUERY returns a null string.

The following example returns the first two jobs in the Caché job table. Note the use of the indirection operator (@):

  SET x=$QUERY('^$JOB(''))
  WRITE !,x
  WRITE !,$QUERY(@x)

returns values for these jobs such as the following:

^$JOB("4294117993")

^$JOB("4294434881")

See Also

- JOB command
- $DATA function
- $ORDER function
- $QUERY function
- $JOB special variable
- $ZJOB special variable
- $ZCHILD special variable
- $ZPARENT special variable
**\^$LOCK**

Provides lock name information.

\^$ | nspace | \texttt{\^$LOCK(lock\_name,info\_type,pid)}
\^$ | nspace | \texttt{\^$L(lock\_name,info\_type,pid)}

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nspace or [nspace]</td>
<td>Optional — An extended SSVN reference, either an explicit namespace name or an implied namespace. Must evaluate to a quoted string, which is enclosed in either square brackets (&quot;nspace&quot;) or vertical bars (</td>
</tr>
<tr>
<td>lock_name</td>
<td>An expression that evaluates to a string containing a lock variable name, either subscripted or unsubscripted. If a literal, must be specified as a quoted string.</td>
</tr>
<tr>
<td>info_type</td>
<td>Optional — A keyword specifying what type of information about lock_name to return. Must be specified as a quoted string. The available options are &quot;OWNER&quot;, &quot;FLAGS&quot;, &quot;MODE&quot;, and &quot;COUNTS&quot;.</td>
</tr>
<tr>
<td>pid</td>
<td>Optional — For use with the &quot;COUNTS&quot; keyword. An integer that specifies the process ID of the owner of the lock. If specified, at most one list element is returned for &quot;COUNTS&quot;. If omitted (or specified as 0), a list element is returned for each owner holding the specified lock. pid has no effect on the other info_type keywords.</td>
</tr>
</tbody>
</table>

### Description

The \^$LOCK structured system variable returns information about locks in the current namespace or a specified namespace on the local system. You can use \^$LOCK in two ways:

- With info\_type as a stand-alone function that returns information on a specified lock.
- Without info\_type as an argument to the $DATA, $ORDER, or $QUERY functions.

**Note:** \^$LOCK retrieves lock table information from the lock table on the local system. It will not return information from a lock table on a remote server.

### \^$LOCK in an ECP Environment

- Local System: When invoking \^$LOCK for locks held by the local system, the behavior of \^$LOCK is the same as without ECP, with one exception: the "FLAGS" info\_type returns an asterisk (*), signifying that the lock is in an ECP environment. This means that remote Caché instances have the ability to hold the lock once it is released.
- Application Server: When invoking \^$LOCK on an application server for a lock held on another server via ECP (either a data server or another application server), \^$LOCK does not return any information. Note that this is the same behavior as if the lock did not exist.
- Data Server: When invoking \^$LOCK on a data server for a lock held by an application server, \^$LOCK will have slightly different behavior than on a local system, as follows:
  - "OWNER": If the lock is held by an application server connected to a data server that invokes \^$LOCK, \^$LOCK(lockname, "OWNER") returns “ECPCOnfigurationName:MachineName:InstanceName” for the instance holding the lock, but does not identify the specific process holding the lock.
“FLAGS”: If the lock is held by an application server connected to a data server that invokes `\$LOCK(lockname, "FLAGS")` for an exclusive lock returns a “Z” flag. This “Z” signifies a type of legacy lock that is no longer used in Caché except in ECP environments.

“MODE”: If the lock is held by an application server connected to a data server that invokes `\$LOCK(lockname, "MODE")` for an exclusive lock returns “ZAX” instead of “X”. ZA is a type of legacy lock that is no longer used in Caché except in ECP environments. For a shared lock, `\$LOCK(lockname, "MODE")` returns “S”, the same as for a local lock.

**Parameters**

`nspace`

This optional parameter allows you to specify a global in another namespace by using an extended SSVN reference. You can specify the namespace name either explicitly, as a quoted string literal or as a variable, or by specifying an implied namespace. Namespace names are not case-sensitive. You can use either bracket syntax ["USER"] or environment syntax ["USER"]. No spaces are allowed before or after the `nspace` delimiters.

You can test whether a namespace is defined by using the following method:

```
WRITE ##class(%SYS.Namespace).Exists("USER"),! ; an existing namespace
WRITE ##class(%SYS.Namespace).Exists("LOSER")   ; a non-existent namespace
```

You can use the `&NAMESPACE` special variable to determine the current namespace. The preferred way to change the current namespace is `NEW &NAMESPACE` then `SET &NAMESPACE="nspacename"`.

`lock_name`

An expression that evaluates to a string containing a lock variable name, either subscripted or unsubscripted. A lock variable (commonly a global variable) is defined using the `LOCK` command.

`info_type`

An optional keyword specified as a quoted string.

- "OWNER" returns the process ID (pid) of the owner(s) of the lock. If the lock is a shared lock, it returns the process IDs of all of the owners of the lock as a comma-separated list. If the specified lock does not exist, `\$LOCK` returns the empty string.

- "FLAGS" returns the state of the lock. It can return the following values: "D" — in delock pending state; "P" — in lock pending state; "N" — this is a node lock, the descendants are not locked; "Z" — this lock is in ZAX mode; "L" — lock is lost, the server no longer has this lock; "#" — this is a remote lock. If the specified lock is in a normal lock state, or does not exist, `\$LOCK` returns the empty string.

- "MODE" returns the lock mode of the current node. It returns 'X' for exclusive lock mode, 'S' for shared lock mode, and 'ZAX' for ZALLOCATE lock mode. If the specified lock does not exist, `\$LOCK` returns the empty string.

- "COUNTS" returns the lock counts for the lock, specified as a binary list structure. For an exclusive lock, the list contains one element; for a shared lock the list contains an element for each owner of the lock. You can use the `pid` parameter to return only the list element for a specified owner of the lock. Each element contains the owner’s pid, the exclusive mode increment count, and the shared mode increment count. If both the exclusive and shared mode increment counts are 0 (or ""), the lock is in ‘ZAX’ mode. An increment count may be followed by a ‘D’ to indicate that the lock has been unlocked in the current transaction, but its release is delayed (‘D’) until the transaction is committed or rolled back. If the specified lock does not exist, `\$LOCK` returns the empty string.
pid
A process ID of the owner of a lock. Only meaningful when using the "COUNTS" keyword. Used to limit the "COUNTS" return value to (at most) one list element. The pid is specified as an integer on all platforms. If the pid matches the process ID of an owner of lock_name \^$LOCK returns that owner's "COUNTS" list element; if pid does not match the process ID of an owner of lock_name \^$LOCK returns the empty string. Specifying pid as 0 is the same as omitting pid; \^$LOCK returns all "COUNTS" list elements. The pid parameter is permitted with the "OWNER", "FLAGS", or "MODE" keyword, but is ignored.

Examples
The following example shows the values returned by info_type keywords for an exclusive lock:

```cache
LOCK ^B(1,1) ; define lock
WRITE !,"lock owner: ",\^$LOCK("^B(1,1)","OWNER")
WRITE !,"lock flags: ",\^$LOCK("^B(1,1)","FLAGS")
WRITE !,"lock mode: ",\^$LOCK("^B(1,1)","MODE")
WRITE !,"lock counts: "
ZZDUMP \^$LOCK("^B(1,1)","COUNTS")
LOCK -^B(1,1) ; delete lock
```

The following example shows how the value returned by info_type "COUNTS" changes as you increment and decrement an exclusive lock:

```cache
LOCK ^B(1,1) ; define exclusive lock
ZZDUMP \^$LOCK("^B(1,1)","COUNTS")
LOCK +^B(1,1) ; increment lock
ZZDUMP \^$LOCK("^B(1,1)","COUNTS")
LOCK +^B(1,1) ; increment lock again
ZZDUMP \^$LOCK("^B(1,1)","COUNTS")
LOCK -^B(1,1) ; decrement lock
ZZDUMP \^$LOCK("^B(1,1)","COUNTS")
LOCK -^B(1,1) ; decrement lock again
ZZDUMP \^$LOCK("^B(1,1)","COUNTS")
LOCK -^B(1,1) ; delete exclusive lock
```

The following example shows how the value returned by info_type "COUNTS" changes as you increment and decrement a shared lock:

```cache
LOCK ^S(1,1)#"S" ; define shared lock
ZZDUMP \^$LOCK("^S(1,1)#"S")
LOCK +^S(1,1)#"S" ; increment lock
ZZDUMP \^$LOCK("^S(1,1)#"S")
LOCK +^S(1,1)#"S" ; increment lock again
ZZDUMP \^$LOCK("^S(1,1)#"S")
LOCK -^S(1,1)#"S" ; decrement lock
ZZDUMP \^$LOCK("^S(1,1)#"S")
LOCK -^S(1,1)#"S" ; decrement lock again
ZZDUMP \^$LOCK("^S(1,1)#"S")
LOCK -^S(1,1)#"S" ; delete shared lock
```

The following examples show how to use \^$LOCK as an argument to the $DATA, $ORDER, and $QUERY functions.

As an Argument to $DATA

$DATA(\^$\{nspace\}[LOCK(lock_name)])

\^$LOCK as an argument to $DATA returns an integer value that specifies whether the lock name exists as a node in \^$LOCK. The integer values that $DATA can return are shown in the following table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Lock name information does not exist</td>
</tr>
<tr>
<td>10</td>
<td>Lock name information exists</td>
</tr>
</tbody>
</table>
The following example tests for the existence of a lock name in the current namespace. The first WRITE returns 10 (lock name exists), the second WRITE returns 0 (lock name does not exist):

```
LOCK ^B(1,2) ; define lock
WRITE !,$DATA(^$LOCK("^B(1,2)"))
LOCK -^B(1,2) ; delete lock
WRITE !,$DATA(^$LOCK("^B(1,2)"))
```

### As an Argument to $ORDER

$ORDER($|$aspace$|LOCK(lock_name), direction)

^$LOCK as an argument to $ORDER returns the next or previous ^$LOCK lock name node in collating sequence to the lock name you specify. If no such lock name exists as a ^$LOCK node, $ORDER returns a null string.

Locks are returned in case-sensitive string collation order. Subscripts of a named lock are returned in subscript tree order, using numeric collation.

The direction argument specifies whether to return the next or the previous lock name. If you do not provide a direction argument, Caché returns the next lock name in collating sequence to the one you specify. For further details, refer to the $ORDER function.

The following subroutine searches the locks in the SAMPLES namespace and stores the lock names in a local array named LOCKET.

```
LOCKARRAY
SET lname=""
FOR I=1:1 {
    SET lname=$ORDER($|$"SAMPLES$|LOCK(lname))
    QUIT: lname=""
    SET LOCKET(I)=lname
    WRITE !,"the lock name is: ", lname
}
WRITE !,"All lock names listed"
QUIT
```

### As an Argument to $QUERY

$QUERY($|$aspace$|LOCK(lock_name))

^$LOCK as an argument to $QUERY returns the next lock name in collating sequence to the lock name you specify. If there is no next lock name defined as a node in ^$LOCK, $QUERY returns a null string.

Locks are returned in case-sensitive string collation order. Subscripts of a named lock are returned in subscript tree order, using numeric collation.

In the following example, five global lock names are created (in random order) in the current namespace.

```
LOCK (^B(1),^A,^D,^A(1,2,3),^A(1,2))
WRITE !,"lock name: ", $QUERY(^$LOCK(""))
WRITE !,"lock name: ", $QUERY(^$LOCK("^C"))
WRITE !,"lock name: ", $QUERY(^$LOCK("^A(1,2)"))
```

$QUERY treats all global lock variable names, subscripted or unsubscripted, as character strings and retrieves them in string collating order. Therefore, $QUERY(^$LOCK("")) retrieve the first lock name in collating sequence order: either ^$LOCK("^A") or a Caché-defined lock higher in the collating sequence. $QUERY(^$LOCK("^C")) retrieves the next lock name in collating sequence after the nonexistent ^C: ^$LOCK("^D"). $QUERY(^$LOCK("^A(1,2)")) retrieve ^$LOCK("^A(1,2,3)") which follows it in collation sequence.

### See Also

- LOCK command
- $DATA function
- $ORDER function
• $QUERY function
• $NAMESPACE special variable
• Configuring Namespaces in Caché System Administration Guide
^$ROUTINE

Provides routine information.

|^$| nspace | ROUTINE(routine_name)
|^$| nspace | R(routine_name)

Parameters

| nspace |
---
Optional — An extended SSVN reference, either an explicit namespace name or an implied namespace. Must evaluate to a quoted string, which is enclosed in either square brackets (["nspace"]) or vertical bars (|"nspace"|). Namespace names are not case-sensitive; they are stored and displayed in uppercase letters.

| routine_name |
---
An expression that evaluates to a string containing the name of a routine.

Description

You can use the ^$ROUTINE structured system variable as an argument to the $DATA, $ORDER, and $QUERY functions to return routine information from the current namespace (the default) or a specified namespace.

Parameters

nspace

This optional parameter allows you to specify a global in another namespace by using an extended SSVN reference. You can specify the namespace name either explicitly, as a quoted string literal or as a variable, or by specifying an implied namespace. Namespace names are not case-sensitive. You can use either bracket syntax ["USER"] or environment syntax [|"USER"|]. No spaces are allowed before or after the nspace delimiters.

You can test whether a namespace is defined by using the following method:

```objectscript
WRITE #!class(%SYS.Namespace).Exists("USER"),! ; an existing namespace
WRITE #!class(%SYS.Namespace).Exists("LOSER")   ; a non-existent namespace
```

You can use the $NAMESPACE special variable to determine the current namespace. The preferred way to change the current namespace is NEW $NAMESPACE then SET $NAMESPACE="nspacename".

routine_name

An expression that evaluates to a string containing the name of a routine. A routine name must be unique within the first 255 characters; routine names longer than 220 characters should be avoided.

Examples

The following are examples of using ^$ROUTINE as an argument to the $DATA, $ORDER, and $QUERY functions.

As an Argument to $DATA

$DATA(^$|nspace|ROUTINE(routine_name))

^$ROUTINE as an argument to $DATA returns an integer value that specifies whether the routine_name exists as a node in ^$ROUTINE. The integer values that $DATA can return are shown in the following table.
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Routine name does not exist</td>
</tr>
<tr>
<td>1</td>
<td>Routine name exists</td>
</tr>
</tbody>
</table>

The following example tests for the existence of a routine in the default namespace:

```objectscript
XECUTE "ZREMOVE"
XECUTE "ZSAVE ROU"
WRITE !,$DATA(^$ROUTINE("ROU"))
XECUTE "ZSAVE ROU"
WRITE !,$DATA(^$ROUTINE("ROU"))
```

**As an Argument to $ORDER**

```
SORDER(^$|nspace|ROUTINE( routine_name),direction)
```

$^$ROUTINE as an argument to $ORDER returns the next or previous routine name in collating sequence to the routine name you specify. If no such routine name exists as a node in $^$ROUTINE, $ORDER returns a null string.

The direction argument specifies whether to return the next or the previous routine name. If you do not provide a direction argument, Caché returns the next routine name in collating sequence to the one you specify. For further details, refer to the $ORDER function.

The following subroutine searches the USER namespace and stores the routine names in a local array named ROUTINE.

```objectscript
SET rname=""
FOR I=1:1 {
   SET rname=$ORDER(^$|"USER"|ROUTINE(rname))
   QUIT:rname=""
   SET ROUTINE(I)=rname
   WRITE !,"Routine name: ",rname
}
WRITE !,"All routines stored"
QUIT
```

**As an Argument to $QUERY**

```
$QUERY(^$|nspace|ROUTINE(routine_name))
```

$^$ROUTINE as an argument to $QUERY returns the next routine name in collating sequence to the routine name you specify. The specified routine name does not have to exist. If there is no routine name later in the collating sequence, $QUERY($^$ROUTINE) returns a null string.

In the following example, two $QUERY functions return the next routine after the specified routine name in the USER namespace.

```objectscript
SET rname=""
WRITE !,"1st routine: ",$QUERY(^$|"USER"|ROUTINE(rname))
SET rname="%m"
WRITE !,"1st %m routine: ",$QUERY(^$|"USER"|ROUTINE(rname))
QUIT
```

**See Also**

- $DATA function
- $ORDER function
- $QUERY function
- $NAMESPACE special variable
- Configuring Namespaces in *Caché System Administration Guide*
MultiValue Commands, Functions, and Special Variables
MV

Invokes the MultiValue Shell.

**MV:pc commands**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td><em>Optional</em> — A postconditional expression.</td>
</tr>
<tr>
<td>commands</td>
<td><em>Optional</em> — A MultiValue Shell command, specified as a quoted string, or a comma-separated list of MultiValue Shell commands, each command specified as a quoted string.</td>
</tr>
</tbody>
</table>

**Description**

MV command has two forms: argumentless MV and MV with a **commands** argument.

**Argumentless MV**

The MV command invokes the Caché MultiValue Shell. Caché displays the MultiValue Shell prompt. An argumentless MV command can only be issued from Terminal.

At the MultiValue Shell prompt you can issue MultiValue Shell commands, as described in the Caché MultiValue Commands Reference. You can also issue Caché MVBasic commands at this prompt by prefacing each MVBasic command with a semicolon (;). Refer to the Caché MultiValue Basic Reference for further details. You can exit the MultiValue Shell by issuing a QUIT command at the MultiValue prompt.

The argumentless MV command enters the interactive MultiValue Shell and immediately rolls back any open transactions. Upon exiting the MultiValue Shell the current $TLEVEL value is 0.

**Note:** The MV MultiValue Shell invocation command *should not* be issued from within the MultiValue Shell. Such nested MultiValue Shell invocations may cause unexpected problems.

**MV commands**

The MV command with the **commands** option also invokes the Caché MultiValue Shell. It issues the specified MultiValue commands and then exits the MultiValue Shell. It does not display the MultiValue Shell prompt.

You can specify one or more MultiValue Shell commands, as described in the Caché MultiValue Commands Reference. Each command is specified as an expression that evaluates to a quoted string. You can specify multiple MultiValue commands as a comma-separated list.

You can also issue Caché MVBasic commands by prefacing each MVBasic command with a semicolon (;). Refer to the Caché MultiValue Basic Reference for further details.

Because MV commands automatically exits the MultiValue command environment, specifying a MultiValue QUIT command has no effect.

**MV commands** has no effect on open transactions. Upon returning from this command the current $TLEVEL value is unchanged.
Arguments

pc
An optional postconditional expression. Either form of MV can take a postconditional. Caché executes the MV command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

commands
One or more Caché MultiValue commands. Each command must be specified as a quoted string. Multiple commands are separated by commas.

Examples
The following example issues several MultiValue commands and a MVBasic command. It uses MVBasic to set a MultiValue lock, lists all of the processes locks, clears all process-defined locks, and again lists all of the locks.

SET lst="LIST.LOCKS"
SET clr="CLEAR.LOCKS"
SET lck=";LOCK 7"
MV lck,lst,clr,lst

See Also

• Caché MultiValue Commands Reference
• Caché MultiValue Basic Reference
MVCALL

Calls a MultiValue subroutine.

MVCALL:pc subr(args) ,subr2(args)...

Arguments

<table>
<thead>
<tr>
<th>pc</th>
<th>Optional — A postconditional expression. For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.</th>
</tr>
</thead>
<tbody>
<tr>
<td>subr</td>
<td>The name of a MultiValue subroutine, or an indirect reference to this name, using the @ indirection operator. You can specify a single subroutine or a series of subroutines with their arguments, separated by commas (,).</td>
</tr>
<tr>
<td>args</td>
<td>Optional — An argument or comma-separated list of arguments to be passed to the MultiValue subroutine. To pass an argument by reference, use the period (.) prefix. The args parentheses are only required if an argument is specified.</td>
</tr>
</tbody>
</table>

Description

The MVCALL command invokes the Caché MVBasic CALL command to call one or more MultiValue subroutines, then returns to the ObjectScript environment. For further details refer to the Caché MultiValue Basic Reference.

Because MVCALL follows ObjectScript precedence rules, you may need to enclose indirect expressions in parentheses to compel the desired precedence of operators.

See Also

- CALL command in Caché MultiValue Basic Reference
MVCRT

Performs MultiValue printing to terminal.

<table>
<thead>
<tr>
<th>MVCRT:pc printlist</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVC:pc printlist</td>
</tr>
</tbody>
</table>

**Arguments**

| pc | Optional — A postconditional expression. For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript. |
| printlist | Optional — Any MVBasic expression that resolves to a quoted string or a numeric. You can specify a single expression or a series of expressions. If printlist is omitted, a blank line is returned. |

**Description**

The MVCRT command invokes the Caché MVBasic CRT command, performs the specified operation, and returns to the ObjectScript environment. Specify parameter values in MVBasic form.

Within printlist, you can specify tab and line return formatting using features of ObjectScript WRITE command syntax, rather than MVBasic CRT command syntax. Only a limited subset of WRITE formatting syntax is supported: comma (concatenate), question mark (tab), and exclamation mark (line return). Multiple exclamation marks and numeric tabs are not supported. The following table shows the format equivalences:

<table>
<thead>
<tr>
<th>MVCRT (ObjectScript)</th>
<th>CRT (MVBasic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVCRT a</td>
<td>CRT a:</td>
</tr>
<tr>
<td>MVCRT a,!</td>
<td>CRT a</td>
</tr>
<tr>
<td>MVCRT a,b,!</td>
<td>CRT a:b</td>
</tr>
<tr>
<td>MVCRT a,!,b,!</td>
<td>CRT a CRT b</td>
</tr>
<tr>
<td>MVCRT a,?,b,!</td>
<td>CRT a,b</td>
</tr>
</tbody>
</table>

For further details refer to the Caché MultiValue Basic Reference.

**See Also**

- MVPRINT command
- WRITE command
- CRT command in Caché MultiValue Basic Reference
MVDIM

Dimensions an array of variables for MultiValue use.

MVDIM:pc array(rows,columns)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>Optional — A postconditional expression. For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.</td>
</tr>
<tr>
<td>array</td>
<td>Name of an array. Follows standard MVBasic variable naming conventions.</td>
</tr>
<tr>
<td>rows</td>
<td>An integer specifying the number of array elements to dimension for a one-dimensional array, or the number of rows to dimension for a two-dimensional array.</td>
</tr>
<tr>
<td>columns</td>
<td>Optional — For two-dimensional (matrix) arrays, an integer specifying the number of columns per row. Can only be used in conjunction with the optional rows argument.</td>
</tr>
</tbody>
</table>

Description

The MVDIM command explicitly dimensions a one-dimensional or two-dimensional MultiValue static array. Dimensioning of arrays is required for MultiValue applications.

Note: Explicit dimensioning of arrays is not required (or supported) for ObjectScript. Arrays in ObjectScript are dynamic and multidimensional. For a description of arrays in ObjectScript, refer to “Multidimensional Arrays” in Using Caché ObjectScript.

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: MVDIM arrayname(n) or a matrix array with a column dimension of one: MVDIM arrayname(n,1).

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 begin 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.

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

Note: 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 array as a dimensioned array, but the number of dimensions and number of elements in each dimension may be expanded dynamically at runtime. Because ObjectScript normally creates arrays of this type, the MVDIM command does not provide an option for this type of implicit dimensioning.

Refer to the Caché MultiValue Basic Reference for further details.

Examples

The following examples illustrate the use of the MVDIM statement:
MVDIM

; Dimensions a one-dimensional array with 10 elements.
MVDIM MyVector(10)

/* Dimensions a two-dimensional matrix array
   with 10 rows and 10 columns. */
MVDIM MyMatrix(10,5)

/* Dimensions a two-dimensional array using local variables */
SET myrows=4,mycols=5
MVDIM MyMatrix(myrows,mycols)

See Also

- SET command
- DIM statement in *Caché MultiValue Basic Reference*
- “Multidimensional Arrays” in *Using Caché ObjectScript*
MVPRINT

Performs MultiValue printing to terminal or printer.

```
MVPRINT:pc printlist
MVPRINT:pc printlist
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pc</code></td>
<td><em>Optional</em> — A postconditional expression. For further details, refer to <em>Command Postconditional Expressions</em> in <em>Using Caché ObjectScript</em>.</td>
</tr>
<tr>
<td><code>printlist</code></td>
<td><em>Optional</em> — Any MVBasic expression that resolves to a quoted string or a numeric. You can specify a single expression or a series of expressions. If <code>printlist</code> is omitted, a blank line is returned.</td>
</tr>
</tbody>
</table>

**Description**

The **MVPRINT** command invokes the Caché MVBasic **PRINT** command, performs the specified operation, and returns to the ObjectScript environment. Specify parameter values in MVBasic form.

Within `printlist`, you can specify tab and line return formatting using features of ObjectScript **WRITE** command syntax, rather than MVBasic **PRINT** command syntax. Only a limited subset of **WRITE** formatting syntax is supported: comma (concatenate), question mark (tab), and exclamation mark (line return). Multiple exclamation marks and numeric tabs are not supported. The following table shows the format equivalences:

<table>
<thead>
<tr>
<th><strong>MVPRINT (ObjectScript)</strong></th>
<th><strong>PRINT (MVBasic)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>MVPRINT a</td>
<td>PRINT a:</td>
</tr>
<tr>
<td>MVPRINT a,!</td>
<td>PRINT a</td>
</tr>
<tr>
<td>MVPRINT a,b,!</td>
<td>PRINT a:b</td>
</tr>
<tr>
<td>MVPRINT a,!,b,!</td>
<td>PRINT a PRINT b</td>
</tr>
<tr>
<td>MVPRINT a,?,b,!</td>
<td>PRINT a,b</td>
</tr>
</tbody>
</table>

For further details refer to the *Caché MultiValue Basic Reference*.

**See Also**

- **MVCRT command**
- **WRITE command**
- **PRINT command in *Caché MultiValue Basic Reference***
$CHANGE

Performs MultiValue substring replacement.

$CHANGE(string,subout,subin,occurrences,begin)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string in which substring substitutions are made. Any expression that resolves to a valid variable. string may be a dynamic array.</td>
</tr>
<tr>
<td>subout</td>
<td>The substring to be replaced. Any expression that resolves to a valid string or numeric.</td>
</tr>
<tr>
<td>subin</td>
<td>The substring to be inserted in place of subout. Any expression that resolves to a valid string or numeric.</td>
</tr>
<tr>
<td>occurrences</td>
<td>Optional — A positive integer specifying the number of occurrences of subout to replace with subin. 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.</td>
</tr>
<tr>
<td>begin</td>
<td>Optional — An integer specifying which occurrence of subout to begin replacement with. If omitted, or specified as 0 or 1, replacement begins with the first occurrence of subout.</td>
</tr>
</tbody>
</table>

Description

The $CHANGE function invokes the Caché MVBasic CHANGE function, and returns the resulting value to the ObjectScript environment. Specify parameter values in MVBasic form. For further details refer to the Caché MultiValue Basic Reference.
MultiValue Commands, Functions, and Special Variables

$MV

Builds or modifies a MultiValue dynamic array.

```
SET $MV(dynvar,fpos[,vpos])=value
```

**Parameters**

<table>
<thead>
<tr>
<th>dynvar</th>
<th>A variable used to hold the MultiValue dynamic array.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fpos</td>
<td>Field-level element. An integer that specifies where to insert the value as a field-level element in the dynvar dynamic array. A value of 0 appends value to the beginning of the dynamic array. A value of –1 appends value to the end of the dynamic array. A positive integer replaces the field-level element at that position with value.</td>
</tr>
<tr>
<td>vpos</td>
<td>Optional — Value-level element. An integer that specifies where to insert the value as a value-level element within the field specified in fpos in the dynvar dynamic array. A value of 0 appends value to the beginning of the fpos field. A value of –1 appends value to the end of the fpos field. A positive integer replaces the value-level element at that position in the fpos field with value.</td>
</tr>
<tr>
<td>spos</td>
<td>Optional — Subvalue-level element. An integer that specifies where to insert the value as a subvalue-level element within the value-level element specified in vpos in the dynvar dynamic array. A value of 0 appends value to the beginning of the vpos element. A value of –1 appends value to the end of the vpos element. A positive integer replaces the subvalue-level element at that position in the vpos with value.</td>
</tr>
</tbody>
</table>

**Description**

The $MV function creates or modifies a MultiValue dynamic array by inserting element values with the appropriate dynamic array delimiters: fpos inserts field mark (@FM) delimiters (CHAR(254)). vpos inserts value mark (@VM) delimiters (CHAR(253)). spos inserts subvalue mark (@SM) delimiters (CHAR(252)). $MV can add elements to the beginning or end, or replace an existing element value with a specified value.

By default, $MV inserts or appends elements, not delimiters. Therefore, it does not insert dynamic array delimiters without element values. To insert or append dynamic array delimiters, specify the positional parameter value as pos::1 and equate to an empty string. This is shown in the following example:

```
SET $MV(dynval,-1)="apple"
SET $MV(dynval,-1)="orange"
WRITE dynval,"!
SET $MV(dynval,-1)="" // appends empty element
WRITE dynval,"!
SET $MV(dynval,-1)="" // does nothing
WRITE dynval,"!
SET $MV(dynval,-1::1)="" // appends 2nd delimiter
WRITE dynval,"!
SET $MV(dynval,-1::1)="" // appends 3rd delimiter
WRITE dynval
```

**Examples**

The following example creates a dynamic array with three field elements:

```
SET $MV(dynval,-1)="apple"
SET $MV(dynval,-1)="orange"
SET $MV(dynval,-1)="banana"
WRITE dynval
```
The following example appends the element value “FRUIT” to the beginning of a three-element dynamic array, creating a dynamic array with four field elements:

```mv
SET $MV(dynval,-1)="apple"
SET $MV(dynval,-1)="orange"
SET $MV(dynval,-1)="banana"
SET $MV(dynval,0)="FRUIT"
WRITE dynval
```

The following example replaces the value of the second dynamic array element (“orange”) with the value “nectarine”:

```mv
SET $MV(dynval,-1)="apple"
SET $MV(dynval,-1)="orange"
SET $MV(dynval,-1)="banana"
SET $MV(dynval,2)="nectarine"
WRITE dynval
```

The following example creates a dynamic array with three field elements, appends an empty fourth element and deletes the second element value:

```mv
SET $MV(dynval,-1)="apple"
SET $MV(dynval,-1)="orange"
SET $MV(dynval,-1)="banana"
SET $MV(dynval,-1)="
SET $MV(dynval,2)="
WRITE dynval
```

The following example creates a dynamic array with three field elements, then adds two value-level elements within the first field element:

```mv
SET $MV(dynval,-1)="apple"
SET $MV(dynval,-1)="orange"
SET $MV(dynval,-1)="banana"
SET $MV(dynval,1,-1)="Braeburn"
SET $MV(dynval,1,-1)="Macintosh"
WRITE dynval
```

**See Also**

- Dynamic Arrays in the Caché MVBasic Reference
$MVAT

Performs MultiValue screen cursor control or sets screen display option.

$MVAT(x,y)
$MVAT(code,arg)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>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.</td>
</tr>
<tr>
<td>y</td>
<td>Optional — A positive integer specifying the vertical line position of the screen cursor. 0=top of screen. If omitted, defaults to the current line.</td>
</tr>
<tr>
<td>code</td>
<td>A negative integer specifying a screen display option code. See the Caché MVBasic @ function for a list of these codes.</td>
</tr>
<tr>
<td>arg</td>
<td>Optional — An argument required by certain code values.</td>
</tr>
</tbody>
</table>

**Description**

The $MVAT function invokes the Caché MVBasic @ function, performs the specified operation, and returns to the ObjectScript environment. Specify parameter values in MVBasic form. For further details refer to the *Caché MultiValue Basic Reference*.

The @ function *does not change* the ObjectScript $X and $Y special variables.
$MVFMT

Performs MultiValue formatting of a string or numeric.

$MVFMT(exp,format)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>The value to be formatted. An expression that resolves to a numeric value or a quoted string.</td>
</tr>
<tr>
<td>format</td>
<td>The formatting code. A quoted string consisting of positional letter and number codes specifying the display format for exp.</td>
</tr>
</tbody>
</table>

**Description**

The $MVFMT function invokes the Caché MVBasic FMT function, and returns the resulting value to the ObjectScript environment. Specify parameter values in MVBasic form. For further details refer to the *Caché MultiValue Basic Reference*.

**Examples**

The following example formats 12345 as a fractional number by adding a decimal point and zero padding it with four fractional digits:

```
WRITE $MVFMT(12345,"R4")
```

The following example formats 12345 as a fractional number. The decimal point is moved three positions to the left, and the result is rounded to two fractional digits:

```
WRITE $MVFMT(12345,"R27")
```

The following example formats 12345 as a date. Note that it uses MultiValue date counting (where 1 = January 1, 1968) rather than Caché $HOROLOG date counting (where 1 = January 1, 1841):

```
WRITE $MVFMT(12345,"D")
```

The following example formats 12345 as a right-justified string of ten characters, using ^ as a fill character:

```
WRITE $MVFMT(12345,"10^R")
```

**See Also**

- $MVFMTS function
- MVBasic FMT function in *Caché MultiValue Basic Reference*
$MVFMTS

Performs MultiValue formatting of elements of a dynamic array.

$MVFMTS(dynarray,format)

**Parameters**

<table>
<thead>
<tr>
<th>dynarray</th>
<th>A dynamic array to be formatted for display.</th>
</tr>
</thead>
<tbody>
<tr>
<td>format</td>
<td>A quoted string consisting of positional letter and number codes specifying the display format for each element of dynarray.</td>
</tr>
</tbody>
</table>

**Description**

The $MVFMTS function invokes the Caché MVBasic FMTS function, and returns the resulting value to the ObjectScript environment. Specify parameter values in MVBasic form. For further details refer to the *Caché MultiValue Basic Reference*.

**See Also**

- $MVFMT function
- MVBasic FMTS function in *Caché MultiValue Basic Reference*
$MVICONV

Performs MultiValue external-to-internal conversion of a string.

$MVICONV(string,codes)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A string expression that specifies a value in external (output) format.</td>
</tr>
<tr>
<td>codes</td>
<td>A quoted string containing one or more code characters that govern the conversion from external format to internal format.</td>
</tr>
</tbody>
</table>

**Description**

The $MVICONV function invokes the Caché MVBasic ICONV function, and returns the resulting value to the ObjectScript environment. Specify parameter values in MVBasic form. For further details refer to the *Caché MultiValue Basic Reference*. 
$MVICONVS

Performs MultiValue external-to-internal conversion of elements of a dynamic array.

$MVICONVS(dynarray,codes)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dynarray</td>
<td>A dynamic array, each element of which specifies a value in external (output) format.</td>
</tr>
<tr>
<td>codes</td>
<td>A quoted string containing one or more code characters that govern the conversion from external format to internal format.</td>
</tr>
</tbody>
</table>

**Description**

The $MVICONVS function invokes the Caché MVBasic ICONVS function, and returns the resulting value to the ObjectScript environment. Specify parameter values in MVBasic form. For further details refer to the *Caché MultiValue Basic Reference*. 
$MVINMAT

Returns the number of MultiValue array elements.

$MVINMAT(array)

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>Optional — The name of an array. If omitted, the most recently parsed array.</td>
</tr>
</tbody>
</table>

Description

The $MVINMAT function returns the number of elements in an array that has been loaded using a MultiValue MATREAD, MATREADU, or MATPARSE statement.

An array must be dimensioned using the ObjectScript MVDIM command or the MultiValue DIMENSION statement.

The ObjectScript $MVINMAT function corresponds to the Cache MultiValue Basic INMAT function.

Examples

The following example returns the number of elements in a one-dimensional (vector) array.

```
SET x=8
MVDIM myarray(x)
WRITE $MVINMAT(myarray)
```

The following example returns the number of elements in a two-dimensional (matrix) array.

```
SET x=7,y=5
MVDIM myarray(x,y)
WRITE $MVINMAT(myarray)
```

See Also

- MVDIM command
- INMAT function in the Caché MVBasic Reference
- DIM statement in the Caché MVBasic Reference
- MATPARSE statement in the Caché MVBasic Reference
- MATREAD statement in the Caché MVBasic Reference
$MVLOWER

Lowers MultiValue dynamic array delimiters to next level.

$MVLOWER(dynarray)

Parameter

| dynarray | Any valid MultiValue dynamic array. |

Description

The $MVLOWER function returns a MultiValue dynamic array with its delimiters converted to the next lower-level delimiters. For example, @VM value mark delimiters become @SM subvalue mark delimiters. When a delimiter cannot be lowered any further, it is returned unchanged.

The available levels, in descending order, are: @IM (CHAR(255)); @FM (CHAR(254)); @VM (CHAR(253)); @SM (CHAR(252)); @TM (CHAR(251)); and CHAR(250).

The ObjectScript $MVLOWER function corresponds to the Cache MultiValue Basic LOWER function. The $MVRAISE function performs the opposite operation, raising the level of dynamic array delimiters to the next higher level.

Example

The following example uses the $MVLOWER function to convert dynamic array delimiters to the next lower level. It then uses the $MVRAISE function to reverse this operation:

```ObjectScript
SET numvm="123"_$CHAR(253)"_456"_$CHAR(253)"_789"
ZZDUMP numvm  /* Returns 123v456v789 */
SET numlower=$MVLOWER(numvm)
ZZDUMP numlower  /* Returns 123s456s789 */
SET numraise=$MVRAISE(numlower)
ZZDUMP numraise  /* Returns 123v456v789 */
```

See Also

- $MVRAISE function
- MultiValue LOWER function in the Caché MVBasic Reference
- Dynamic Arrays in the Caché MVBasic Reference
$MVOCONV

Performs MultiValue internal-to-external conversion of a string.

$MVOCONV(string, codes)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A string expression that specifies a value in internal (storage) format.</td>
</tr>
<tr>
<td>codes</td>
<td>A quoted string containing one or more code characters that govern the conversion from internal format to external format.</td>
</tr>
</tbody>
</table>

Description

The $MVOCONV function invokes the Caché MVBasic OCONV function, and returns the resulting value to the ObjectScript environment. Specify parameter values in MVBasic form. For further details refer to the Caché MultiValue Basic Reference.
$MVOCONVS

Performs MultiValue internal-to-external conversion of elements of a dynamic array.

$MVOCONVS(dynarray,codes)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dynarray</td>
<td>A dynamic array, each element of which specifies a value in internal (storage) format.</td>
</tr>
<tr>
<td>codes</td>
<td>A quoted string containing one or more code characters that govern the conversion from internal format to external format.</td>
</tr>
</tbody>
</table>

**Description**

The $MVOCONVS function invokes the Caché MVBasic OCONVS function, and returns the resulting value to the ObjectScript environment. Specify parameter values in MVBasic form. For further details refer to the *Caché MultiValue Basic Reference*. 
$MVRAISE

Raises MultiValue dynamic array delimiters to next level.

$MVRAISE(dynarray)

**Parameter**

| dynarray | Any valid MultiValue dynamic array. |

**Description**

The $MVRAISE function returns a MultiValue dynamic array with its delimiters converted to the next higher-level delimiters. For example, @SM subvalue mark delimiters become @VM value mark delimiters. When a delimiter cannot be raised any further, it is returned unchanged.

The available levels, in ascending order, are: CHAR(250); @TM (CHAR(251)); @SM (CHAR(252)); @VM (CHAR(253)); @FM (CHAR(254)); and @IM (CHAR(255)).

The ObjectScript $MVRAISE function corresponds to the Cache MultiValue Basic RAISE function. The $MVLOWER function performs the opposite operation, lowering the level of dynamic array delimiters to the next lower level.

**Examples**

The following example uses the $MVRAISE function to convert dynamic array delimiters to the next higher level. It then uses the $MVLOWER function to reverse this operation:

```OBJECTSCRIPT
SET numvm="123"_$CHAR(253)"456"_$CHAR(253)"789"
ZZDUMP numvm       /* Returns 123v456v789 */
SET numraise=$MVRAISE(numvm)
ZZDUMP numraise    /* Returns 123f456f789 */
SET numlower=$MVLOWER(numraise)
ZZDUMP numlower    /* Returns 123v456v789 */
```

**See Also**

- $MVLOWER function
- MultiValue RAISE function in the Caché MVBasic Reference
- Dynamic Arrays in the Caché MVBasic Reference
**$MVTRANS**

Reads field data from a MultiValue file.

$$MVTRANS(mvfile,recID,fieldno,code)$$

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mvfile</td>
<td>The name of a MultiValue file defined in the VOC, specified as a quoted string. If there are multiple data files, you can specify <code>mvfile</code> as either &quot;dirname, datafile&quot; or simply &quot;datafile&quot;.</td>
</tr>
<tr>
<td>recID</td>
<td>The record ID of the desired record, specified as an integer.</td>
</tr>
<tr>
<td>fieldno</td>
<td>Either an integer or a non-numeric string. If an integer, <code>fieldno</code> 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, <code>fieldno</code> specifies an attribute to look up in the file’s dictionary. If that attribute is a “D” data defining entry, $MVTRANS looks up the data using the corresponding field number recorded in the dictionary. This usage is compatible with UniData systems.</td>
</tr>
<tr>
<td>code</td>
<td>A letter code that specifies what to do if the requested record does not exist. This letter code must be specified as a quoted string.</td>
</tr>
</tbody>
</table>

**Description**

The $MVTRANS 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 $MVTRANS.

The `code` parameter determines how $MVTRANS should respond when you request an invalid field. The following are valid letter codes:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;X&quot;</td>
<td>Return an empty string if the specified record or field cannot be located.</td>
</tr>
<tr>
<td>&quot;V&quot;</td>
<td>Return an empty string and generate an error message if the specified record or field cannot be located.</td>
</tr>
<tr>
<td>&quot;C&quot;</td>
<td>Return the <code>recID</code> value if the specified record or field cannot be located.</td>
</tr>
<tr>
<td>&quot;N&quot;</td>
<td>Return the <code>recID</code> value if the specified field value is null.</td>
</tr>
</tbody>
</table>

The ObjectScript $MVTRANS function corresponds to the Cache MultiValue Basic TRANS function.

**Examples**

The following examples illustrates the use of the $MVTRANS function. Because TEST.FILE does not exist, each returns the error value dictated by its `code` parameter:

```
SET mydyn=$MVTRANS("TEST.FILE",6,1,"X")
WRITE !,"the field value:",mydyn

SET mydyn=$MVTRANS("TEST.FILE",6,1,"V")
WRITE !,"the field value:",mydyn

SET mydyn=$MVTRANS("TEST.FILE",6,1,"C")
WRITE !,"the field value:",mydyn
```
See Also

- **TRANS** function in the *Caché MVBasic Reference*
- **READ** statement in the *Caché MVBasic Reference*
- **STATUS** statement in the *Caché MVBasic Reference*
- **Dynamic Arrays** in the *Caché MVBasic Reference*
$MVname Special Variables by Name

Caché special variables corresponding to MultiValue system variables.

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$MVname</td>
<td>The name of the MultiValue special variable. In most cases, this name corresponds to a MultiValue $name system variable. For example, $MVAM corresponds to @AM.</td>
</tr>
</tbody>
</table>

Description

$MVname provides ObjectScript equivalents for many Caché MVBasic system variables. It also provides additional MultiValue special variables not found in Caché MVBasic.

The following correspondences table lists the $MVname special variable, the Caché MultiValue Basic system variable, and the $MVV(n) special variable. If there is no corresponding special variable, the table item is left blank. For definitions of these special variables, refer to System Variables in the Caché MultiValue Basic Reference.

<table>
<thead>
<tr>
<th>ObjectScript Special Variable</th>
<th>Caché MVBasic System Variable</th>
<th>ObjectScript $MVV(n)</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>@ACCOUNT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVAM</td>
<td>@AM</td>
<td>$MVV(1)</td>
<td>b</td>
</tr>
<tr>
<td>$MVANS</td>
<td>@ANS</td>
<td>$MVV(44)</td>
<td></td>
</tr>
<tr>
<td>@AUTHORIZATION</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVCOMMAND</td>
<td>@COMMAND</td>
<td>$MVV(23)</td>
<td></td>
</tr>
<tr>
<td>@COMMAND.STACK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVCONV</td>
<td>@CONV</td>
<td>$MVV(95)</td>
<td></td>
</tr>
<tr>
<td>@CRTHIGH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>@CRTWIDE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>@DATA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>@DATA.PENDING</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVDICT</td>
<td>@DICT</td>
<td>$MVV(88)</td>
<td></td>
</tr>
<tr>
<td>@FALSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>@FILE.NAME</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVFILENAME</td>
<td>@FILENAME</td>
<td>$MVV(22)</td>
<td></td>
</tr>
<tr>
<td>$MVFNM</td>
<td>@FM</td>
<td>$MVV(1)</td>
<td>b</td>
</tr>
<tr>
<td>$MVFORMAT</td>
<td>@FORMAT</td>
<td>$MVV(96)</td>
<td></td>
</tr>
<tr>
<td>ObjectScript Special Variable</td>
<td>Caché MVBasic System Variable</td>
<td>ObjectScript $MVV(n)</td>
<td>Default Value</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------------------</td>
<td>----------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>$MVHEADER</td>
<td>@HEADER</td>
<td>$MVV(97)</td>
<td></td>
</tr>
<tr>
<td>$MVID</td>
<td>@ID</td>
<td>$MVV(16)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@IM</td>
<td>$MVV(0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@ITYPCACHE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@LASTLOGONPROG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVLEVEL</td>
<td>@LEVEL</td>
<td>$MVV(133)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>@LOGNAME</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@LTPRHIGH</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@LTPRWIDTH</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@ME</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVMONTH</td>
<td>@MONTH</td>
<td>$MVV(39)</td>
<td>integer for current month</td>
</tr>
<tr>
<td></td>
<td>@MV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVNB</td>
<td>@NB</td>
<td>$MVV(156)</td>
<td>0</td>
</tr>
<tr>
<td>$MVND</td>
<td>@ND</td>
<td>$MVV(157)</td>
<td>0</td>
</tr>
<tr>
<td>$MVNI</td>
<td>@NI</td>
<td>$MVV(158)</td>
<td>0</td>
</tr>
<tr>
<td>$MVNS</td>
<td>@NS</td>
<td>$MVV(160)</td>
<td>0</td>
</tr>
<tr>
<td>$MVNV</td>
<td>@NV</td>
<td>$MVV(159)</td>
<td>0</td>
</tr>
<tr>
<td>$MVPARASENTERCE</td>
<td>@PARASENTERCE</td>
<td>$MVV(84)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@PATH</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@PORTNO</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@QWHO</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@RECCOUNT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVRECORD</td>
<td>@RECORD</td>
<td>$MVV(17)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@RECURn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVSELECTED</td>
<td>@SELECTED</td>
<td>$MVV(145)</td>
<td>0</td>
</tr>
<tr>
<td>$MVSENTERCE</td>
<td>@SENTERCE</td>
<td>$MVV(23)</td>
<td></td>
</tr>
<tr>
<td>$MVSM</td>
<td>@SM</td>
<td>$MVV(3)</td>
<td>ü</td>
</tr>
<tr>
<td>$MVSTDFIL</td>
<td>@STDFIL</td>
<td>$MVV(35)</td>
<td></td>
</tr>
<tr>
<td>$MVSVM</td>
<td>@SVM</td>
<td>$MVV(3)</td>
<td>ü</td>
</tr>
<tr>
<td></td>
<td>@SYS.BELL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVSYSENTERCODE</td>
<td>@SYSTEM.RETURN.CODE</td>
<td>$MVV(36)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@SYSTEM.SET</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@TERM.TYPE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### MultiValue Commands, Functions, and Special Variables

<table>
<thead>
<tr>
<th>ObjectScript Special Variable</th>
<th>Caché MVBasic System Variable</th>
<th>ObjectScript $MVV(n)</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$MVTIME</td>
<td>@TIME</td>
<td>$MVV(137)</td>
<td>5-digit MV internal value for current time</td>
</tr>
<tr>
<td>$MVTM</td>
<td>@TM</td>
<td>$MVV(4)</td>
<td>û</td>
</tr>
<tr>
<td></td>
<td>@TRANSACTION</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@TRUE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVTTY</td>
<td>@TTY</td>
<td>$MVV(42)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@UDTNO</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@UID</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@USER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@USER.NO</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@USERN0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVUSERRETCODE</td>
<td>@USER.RETURN.CODE</td>
<td>$MVV(37)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@USER.TYPE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVVM</td>
<td>@VM</td>
<td>$MVV(2)</td>
<td>ý</td>
</tr>
<tr>
<td></td>
<td>@WHO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVYEAR</td>
<td>@YEAR</td>
<td>$MVV(40)</td>
<td>2-digit current year</td>
</tr>
<tr>
<td></td>
<td>@YEAR4</td>
<td>$MVV(41)</td>
<td></td>
</tr>
</tbody>
</table>

The following are $MVname special variables for which there is no corresponding Caché MultiValue Basic system variable:

<table>
<thead>
<tr>
<th>ObjectScript Special Variable</th>
<th>ObjectScript $MVV(n)</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$MVERRORS</td>
<td>$MVV(66)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVFOOTER</td>
<td>$MVV(25)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVMORESUBVALUES</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVMOREVALUES</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVOPTIONS</td>
<td>$MVV(71)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVPERRORS</td>
<td>$MVV(67)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVPROCNAME</td>
<td>$MVV(70)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVPROCPIB</td>
<td>$MVV(62)</td>
<td></td>
<td>Primary input buffer pointer position.</td>
</tr>
<tr>
<td>$MVPROCPIBOFF</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVPROCPOB</td>
<td>$MVV(64)</td>
<td></td>
<td>Primary output buffer pointer position.</td>
</tr>
<tr>
<td>$MVPROCPSIB</td>
<td>$MVV(63)</td>
<td></td>
<td>Secondary input buffer pointer position.</td>
</tr>
<tr>
<td>$MVPROCSIBOFF</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MVPROCSOB</td>
<td>$MVV(65)</td>
<td></td>
<td>Secondary output buffer pointer position.</td>
</tr>
</tbody>
</table>
See Also

- $MVV(n) special variables
- System Variables in the Caché MultiValue Basic Reference
$MVV(n) Special Variables by Number

Caché special variables corresponding to MultiValue system variables.

$MVV(n)

Description

$MVV(n) is an array of Caché MultiValue special variables. The n subscript can be an integer in the range of 0 through 255, inclusive, or a variable set to one of those integer values. Each subscripted $MVV variable has a defined data type and contains a default value. All $MVV variables can be set using the SET command.

Many $MVV variables correspond to Caché MultiValue Basic system variables. For definitions of these MVBasic system variables, refer to System Variables in the Caché MultiValue Basic Reference.

The following are the data types of the $MVV variables:

<table>
<thead>
<tr>
<th>Subscript Range</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 through 15</td>
<td>Single-character data</td>
</tr>
<tr>
<td>16 through 127</td>
<td>String data</td>
</tr>
<tr>
<td>128 through 193</td>
<td>Integer data</td>
</tr>
<tr>
<td>194 through 255</td>
<td>Boolean data</td>
</tr>
</tbody>
</table>

The following are the MultiValue system variable assignments and default values for the $MVV variables:

| Subscript Range     | Default Value                                                      |
|---------------------|====================================================================|
| 0 through 7         | Dynamic array delimiter characters with ASCII values 248 through 255: 0=@IM ($CHAR(255)), 1=@FM/@AM ($CHAR(254)), 2=@VM ($CHAR(253)), 3=@SM/@SVM ($CHAR(252)), 4=@TM ($CHAR(251)), 5=$CHAR(250), 6=$CHAR(249), 7=$CHAR(248). |
| 8 through 14        | Empty string                                                       |
| 15                  | Question mark (?) character                                      |
| 16 through 37       | Empty string. 16=@ID, 17=@RECORD, 19=MVBasic device, 22=@FILENAME, 23=@COMMAND/@SENTENCE, 25=footer, 35=@STDFIL, 36=@SYSTEM.RETURN.CODE. |
| 38 through 41       | Components of the current date, as integers: 38=@DAY, 39=@MONTH, 40=@YEAR (2-digit year), 41=@YEAR4. |
| 42 through 84       | Empty string. 42=@TTY, 44=@ANS, 46= SQL error message during index operations, 51=spooler print job list, 62=proc pib, 63=proc sib, 64=proc pob, 65=proc sob, 66=errors, 67=proc errors, 70= proc name, 71=options, 84=@PARASENTENCE. |
| 85                  | Question mark (?) character                                      |
| 86 through 127      | Empty string. 88=@DICT, 95=@CONV, 96=@FORMAT, 97=@HEADER           |
| 128                 | MVBasic status. On failed lock operation, contains the pid of the lock owner. |
### Subscript Range | Default Value
--- | ---
129 through 131 | The integer 0.
132 | The integer 10
133 through 135 | The integer 0. 133=@LEVEL
136 | 136=@DATE. The date when the current process started, in internal $HOROLOG format. An integer count of days since January 1, 1841.
137 | 137=@TIME. The time when the current process started, in internal $HOROLOG format. An integer count of seconds since midnight.
138 through 147 | The integer 0. 145=@SELECTED
148 | The integer 4
149 through 255 | The integer/boolean value 0. 156=@NB, 157=@ND, 158=@NI, 159=@NV, 160=@NS. 169=SQL error code during index operations. 174=COMMON unique node subscript ID. 176=MVBasic lockport; on failed lock operation, contains the pid of the lock owner. 210=CMQL generated a Select List. 213=STACKCOMMON on/off. 218=%SYSTEM.MV.InputDataOnly() method setting.

### See Also
- $MVname special variables
- System Variables in the Caché MultiValue Basic Reference
Legacy Commands and Functions

In most cases, legacy commands and functions have been replaced by newer implementations with different syntax. These commands and functions should not be used in new programming. They are documented here for compatibility with existing program code. Note that legacy commands and functions are not compatible with their replacements; though they may appear in the same program, they should never be combined. For example, you should not use a block-oriented IF with the legacy line-oriented ELSE, or attempt to manipulate a bitstring created with a $BIT function using a $ZBIT function, or vice versa.

Legacy Commands

The following legacy commands have been replaced by new command syntax:

- FOR
- IF
- ELSE
- DO (without an argument)

The legacy IF, ELSE, and FOR commands do not use curly brace block structure syntax. Instead, they execute commands that appear on the same program line. They are thus much more restricted as to line formatting than the newer block-oriented versions of these commands.

In addition, the legacy IF command cannot use the ELSEIF clause, and uses the $TEST special variable.

The legacy argumentless DO uses a period prefix syntax to indicate block structure. This syntax has been superseded and is not compatible with curly brace syntax. The argumentless form of DO should not be used for future coding.

The following legacy commands are obsolete:

- ZQUIT has been replaced by ZTRAP $ZERROR. Refer to the ZTRAP command for further details.
- ZSYNC is only used for the obsolete Distributed Cache Protocol (DCP).

Legacy Functions

- The following legacy functions have been replaced by the new $BIT functions: $ZBITAND, $ZBITCOUNT, $ZBITFIND, $ZBITGET, $ZBITLEN, $ZBITNOT, $ZBITOR, $ZBITSET, $ZBITSTR, $ZBITXOR.
- The following legacy functions have been replaced by %SYS.ProcessQuery class properties: $ZUTIL(67,0), $ZUTIL(67,4), $ZUTIL(67,5), $ZUTIL(67,6), $ZUTIL(67,7), $ZUTIL(67,8), $ZUTIL(67,9), $ZUTIL(67,10), $ZUTIL(67,11), $ZUTIL(67,12), $ZUTIL(67,13), $ZUTIL(67,14), and $ZUTIL(67,15). Refer to the Replacements for ObjectScript $ZUTIL Functions table for details.
- The following legacy function is no longer needed to distinguish Windows platforms: $ZUTIL(100).
- The following legacy function is no longer needed for Caché upgrades: $ZUTIL(113).
- The following legacy function supports the superseded ViewPoint metric counters: $ZUTIL(133).
DO (legacy version)

Argumentless: executes the block of code that immediately follows it in the same program.

<table>
<thead>
<tr>
<th>PC</th>
<th>Optional — A postconditional expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>One or more ObjectScript commands executed as a code block. Note the presence of a period (.) prefix before each command in this block. Comments and blank lines must also have a period prefix.</td>
</tr>
<tr>
<td>nextcommand</td>
<td>The next ObjectScript command following the DO code block. This is the first line of code after the argumentless DO that is not prefixed by a period.</td>
</tr>
</tbody>
</table>

Description

Note: This page describes the legacy argumentless DO command. The argumentless version of the DO command is considered legacy as of Caché 4.0, and should not be used in new programming. It is described here solely for compatibility with legacy applications.

The legacy argumentless DO command uses a period prefix to group lines of code together into a code block. Curly braces are not used and line formatting is restrictive. This syntax has been superseded by the curly brace syntax. Argumentless DO is no longer needed within IF or FOR commands, and the DO:pc operation has been superseded by the block-structured IF command. This syntax is not compatible with curly brace syntax; therefore, current Caché block structure commands such as IF and FOR that use curly brace syntax may not be used within an argumentless DO code block.

The argumentless DO command executes a block of code that immediately follows it in the same program. Argumentless DO blocks can be nested, and a postconditional expression on the DO command can be used to determine whether or not to execute a code block.

Caché executes the block of code that immediately follows the DO command, then executes the next command after that block of code.

The lines of code executed by an argumentless DO must be written using a special block structure syntax. This syntax uses periods (.) at the beginning of code lines. This block structure syntax is used only with the argumentless DO command.

You can specify the argumentless DO with a postconditional expression. If the postconditional expression tests FALSE (0), Caché skips the immediately following block of code (and any nested blocks of code within it) and continues execution at the same line level as the DO command.

Argumentless DO Block Structure

Because block structures immediately follow argumentless DO commands, you can use them to make your programs easier to read and maintain. As a rule, you should consider using block structures to replace short routines that are called only once and that might otherwise be spread throughout your code. The placement of block structures after their related DO commands makes it easier to find them. The obvious levels of the structures themselves make it easier to read the code.

A block structure consists of one or more blocks of code, with each block consisting of one or more lines on the same level of nesting. All of the lines on a given level are distinguished by having the same number of periods (.) as prefix to the line of code.
**Code Block Syntax**

Periods used to indicate the lines belonging to a code block take the following syntax:

- The argumentless **DO** command should be the last command on its code line. It may be followed, after one or more whitespace characters, by a comment indicator character (; or //) on the same line.
- The code line immediately following the argumentless **DO** command, and every line within the block of code must have a period prefix, even if it is a comment line or a blank line.
- The periods must prefix the code line. Therefore, no commands or comments may appear before the periods or between the periods on a code line.
- The period must be indented. That is, it cannot be in the first column of the code line. Usually, the initial period is indented to the same level as the code line containing the argumentless **DO** command.
- Nested code blocks mark each level of nesting by using additional prefix periods. To indicate nesting, place the first period at the level of the code line containing the outermost **DO**, and indent with additional periods for each additional level of nesting.
- A code block cannot contain labels. It can contain comment lines and blank lines, but these lines must be prefixed by a period, following the same rules as code lines.
- A period must be followed by at least one whitespace character (a blank space or tab).
- This space character must be followed by either a command keyword, another prefix period (.), a comment indicator (; or //), or a line return. Therefore, line breaks within a command should not be used with this syntax.
- Period prefix code blocks should not be combined with code blocks delineated by curly braces.

**Nesting**

Period prefix code blocks can be nested within each other. Because the lines belonging to a given code block all have the same number of prefix periods, you can easily visually distinguish the contents of each block.

When viewed in a listing, the lines in nested code blocks appear indented relative to each other. For example, the lines in an inner block contain one more prefix period character than the lines in the outer block that contains it.

When a block ends, as indicated by a line at with fewer prefix periods than the current line, ObjectScript issues an implicit **QUIT** that exits the block of code, and resumes execution at the previous level. You can code an explicit **QUIT** command as the last line of the block, but it is not necessary.

**Variables**

All code blocks share the same local variables. Therefore, you can supply a value to an inner block by setting a variable before invoking that code block level. Similarly, any results from the execution of an inner block can be preserved at the next higher level by changing the values of shared local variables.

The $TEST special variable is handled differently by argumentless **DO**, than by a **DO** command calling a subroutine or procedure. The argumentless **DO** preserves the initial value of $TEST during execution of its code block. If the block includes a command (such as a legacy **IF** command, or a timed **OPEN**) that resets the value of $TEST, this change is not passed back to the next higher level.

**QUIT Command**

If you issue a **QUIT** command within an argumentless **DO** code block, Caché quits the immediate code block and continues execution with the next command following that code block.

The following example shows this **QUIT** behavior:
DO
.. WRITE "into the DO"!,
.. DO
... WRITE "inner DO"!,
... QUIT
... WRITE "never written"!,
... WRITE "back to outer DO"!,
.. QUIT
. WRITE "never written"!,
WRITE "out of the DO"

Arguments

pc
An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

blockcommand
One or more ObjectScript commands executed as a code block. Each blockcommand line (including comments and blank lines) must be prefixed by one or more periods, as specified in the Code Block Syntax.

Examples

Note: Note that the following examples use the legacy versions of the IF and FOR commands. Commands that delineate a block of code using curly braces (such as the current version of IF and FOR) do not need to use the argumentless DO, and are not compatible with period prefix syntax.

In the following example, the two argumentless DO commands each invoke a block of code. Which DO is executed and, consequently, which block is invoked depends on which operation the user requests, as determined by the IF command. In each case, the result is passed back to the WRITE command through the same shared local variable. The first block (which calculates the square of an integer) contains an implicit QUIT while the second block (which calculates the cube of an integer) contains an explicit QUIT.

Start ; Square or cube an integer.
READ !,"Square (S) or cube (C): ",op QUIT:op=""
READ !,"Integer: ",num QUIT:num=""
IF (op["S")!(op["s") DO
  SET result=num*num            ; Square block
  WRITE !,"Result: ",result
ELSE  DO
  SET result=num*num*num        ; Cube block
  WRITE !,"Result: ",result
  QUIT
GOTO Start

In the following example, the argumentless DO repeatedly executes the code block until the FOR control variable (i) equals the value of y.

FOR i=1:1:y DO
  SET z=z*x
  WRITE !,z
  QUIT

The following example shows nested code blocks in ObjectScript, using implicit QUIT commands to end each block.
As shown in the preceding example, the first argumentless DO begins execution of the lines in the outermost block of code. ObjectScript saves the line level at which the DO resides. If ObjectScript encounters subsequent argumentless DO commands, it executes the next inner block of code and increments the line level by one for each such command. When it finds an implicit or explicit QUIT in a block, ObjectScript decrements the line level count by one and continues execution at the command that follows the DO for that block.

**See Also**

- DO command
- $TEST special variable
FOR (legacy version)

Executes a command loop repeatedly, testing at the beginning of each loop.

```
FOR variable=forparameter,... command
F variable=forparameter,... command
```

where `forparameter` can be:

```
expr
start:increment
start:increment:end
```

and `command` is one or more ObjectScript commands on the same program line. This command (or commands) are executed repeatedly, based on the FOR test.

**Arguments**

| variable | Optional — A counter variable for the FOR loop. |
| expr     | Optional — The value of `variable` before executing the loop commands. |
| start    | Optional — The value of `variable` on the first iteration of the FOR loop. |
| increment| Optional — The value used to increment `variable` after each iteration of the FOR loop. |
| end      | Optional — The value used to terminate the FOR loop. |

**Description**

*Note:* This page describes the legacy version of the FOR command. This version is considered legacy as of Caché 4.0, and should not be used in new programming. It is described here solely for compatibility with legacy applications.

The legacy FOR command is line-oriented; the loop it executes consists of commands that follow it on the same program line. Curly braces are not used and line formatting is restrictive. The new FOR command is block structured; the loop it executes consists of commands found within the curly braces that follow the FOR command. Line formatting (white space, line breaks) is unrestrictive.

The **FOR** command has two basic forms:

- Without an argument
- With an argument

**FOR Without an Argument**

**FOR** without an argument executes the command(s) that follow it on the same line indefinitely. Caché repeats these commands until it encounters a **QUIT** or **GOTO** command.

The **FOR** keyword without an argument must be separated from the command that follows it by at least two blank spaces.

**FOR With an Argument**

The action **FOR** performs depends on the argument form you use.

**FOR variable=expr** executes the commands that follow it on the same line once, setting `variable` to the value of `expr`. 

```
FOR variable=start:increment executes the commands that follow it on the same line indefinitely. On the first iteration, Caché sets variable to the value of start. Each execution of the FOR command increments the variable value by the specified increment value. Caché repeats the commands until it encounters a QUIT or GOTO command.

FOR variable=start:increment:end sets variable to the value of start. Caché then executes the commands that follow it on the same line based on the conditions described in this table:

<table>
<thead>
<tr>
<th>increment is positive</th>
<th>increment is negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>If start=end, do not execute FOR. When variable &gt; (end-increment), or if Caché encounters a QUIT or GOTO command, stop.</td>
<td>If start&lt;end, do not execute FOR. When variable&lt;(end-increment), or if Caché encounters a QUIT or GOTO command, stop.</td>
</tr>
</tbody>
</table>

Caché evaluates the start, increment, and end values when it begins command execution. Any changes made to these values within the loop are ignored and have no effect on the number of loop executions.

When the loop terminates, variable contains a value that reflects the increment resulting from the last execution of the loop. However, variable is never incremented beyond the maximum value specified in end.

**Arguments**

**variable**

A variable that holds the current count for the FOR loop. It is a local variable name which holds the numeric value specified by the other arguments of the FOR command.

**expr**

The numeric value Caché assigns to variable before executing the loop commands. The value of expr can be specified as a literal or any valid expression; Caché evaluates expr for its numeric value.

**start**

The numeric value Caché assigns to variable on the first iteration of the FOR loop. The value of start can be specified as a literal or any valid expression; Caché evaluates start for its numeric value.

**increment**

The numeric value Caché uses to increment variable after each iteration of the FOR loop. It is required if you specify start. The value of increment can be specified as a literal or any valid expression; Caché evaluates increment for its numeric value.

**end**

The numeric value Caché uses to terminate a FOR loop. When variable equals this value, the FOR loop is executed one last time and then terminated. The value of end can be specified as a literal or any valid expression; Caché evaluates end for its numeric value.

**Examples**

**Argumentless FOR**

In the following example, demonstrating argumentless FOR, the user is prompted repeatedly for a number that is then passed to the Calc subroutine by the DO command. The FOR loop terminates when the user enters a null string (presses ENTER without inputting a number), which causes the QUIT command to execute. The argumentless FOR keyword must be followed by two (or more) spaces.

```objectscript
FOR READ !,"Number: ",num QUIT:num="" DO Calc(num)
```

Caché ObjectScript Reference 779
**Using FOR variable=expr**

When you specify variable=expr, Caché executes the **FOR** command(s) once. The value in *expr* can be a literal or any valid expression. If you specify an expression, it must evaluate to a single numeric value.

In the following example, the **WRITE** command on the line with the **FOR** command executes once, with num having the value 4. It writes the number 12:

```
SET val=4
FOR num=val WRITE num*3 QUIT
```

**Using FOR variable=start:increment:end**

The arguments *start*, *increment*, and *end* specify a start, increment, and end value, respectively. All three are evaluated as numbers. They can be integer or real, positive or negative. If you supply string values, they are converted to their numeric equivalents at the start of the loop.

When Caché first enters the loop, it assigns the *start* value to *variable* and compares the *variable* value to the *end* value. If the *variable* value is less than the *end* value (or greater than it, in the case of a negative *increment* value), Caché executes the loop commands. It then updates the *variable* value using the *increment* value. (The *variable* value is decremented if a negative *increment* is used.)

Execution of the loop continues until the incrementing of the *variable* value would exceed (not just equal) the *end* value (or until Caché encounters a **QUIT** or **GOTO**). At that point, to prevent variable from exceeding end, Caché suppresses variable assignment and loop execution ends. If the increment causes the *variable* value to equal the *end* value, Caché executes the **FOR** loop one last time and then terminates the loop.

The following code executes the **WRITE** command repetitively to output, in sequence, all of the characters in string1, except for the last one. Even though the *end* value is specified as *len-2*, only the last character will be skipped. This is because the loop is terminated only when the *variable* value exceeds (not just matches) the *end* value.

```
SET len=$LENGTH(string1)
FOR index=1:1:len-2 WRITE $EXTRACT(string1,index)
```

**Using FOR variable=start:increment**

In this form of the **FOR** command there is no *end* value; the commands following the **FOR** must contain a **QUIT** or **GOTO** command to terminate the loop. In many cases, the **FOR** invokes a **DO** loop, which contains the **QUIT** or **GOTO** command.

The *start* and *increment* values are evaluated as numbers. They can be integer or real, positive or negative. If string values are supplied, they are converted to their numeric equivalents at the start of the loop. Caché evaluates the *start* and *increment* values when it begins execution of the loop. Any changes made to these values within the loop are ignored.

When Caché first enters the loop, it assigns the *start* value to *variable* and executes the loop commands. It then updates the *variable* value using the *increment* value. (The *variable* value is decremented if a negative *increment* is used.) Execution of the loop continues until Caché encounters a **QUIT** or **GOTO** within the loop.

The following example illustrates use of the **FOR variable=start:increment** form to compute an average for a series of user-supplied numbers. The postconditional **QUIT** is included to terminate execution of the loop when the user enters a null string (that is, presses ENTER without inputting a value). When the postconditional expression (num="") tests **TRUE**, Caché executes the **QUIT** and terminates the **DO** loop.

The loop counter (the *i* variable) is used to keep track of how many numbers have been entered. *i* is initialized to 0 because the counter increment occurs after the user inputs a number. Caché terminates the loop when the user enters a null. After the loop is terminated, the **SET** command references *i* (as a local variable) to calculate the average.
Further Examples of FOR with Increment Syntax

The following example of the `FOR variable=start:increment` form prompts the user for a number and then passes the number and the current value of x to the Calc subroutine invoked by the DO command. The value of x is initialized to 1 and is incremented by 1 for each execution of the loop. The `FOR` loop terminates when the user presses `ENTER`, which causes the `QUIT` command to execute.

```
FOR x=1:1 READ !,"Number: ",num QUIT:num="" DO Calc(num,x)
```

Calc(a,b)

The following example of the `FOR variable=start:increment:end` form prompts the user for a number and then passes the number to the Calc subroutine invoked by `DO`. The value of x is initialized to 1 and is incremented by 1 for each execution of the loop. The `FOR` loop terminates either when the value of x would exceed the end value (3) or when the user presses `ENTER`, causing the `QUIT` command to execute.

```
Mainloop
  FOR x=1:1:3 READ !,"Number: ",num QUIT:num="" DO Calc(num)
  Calc(a)
```

The following example of the `FOR variable=start:increment:end` form shows a `FOR` loop where the `WRITE` command is never executed, since the first value of i (10) is already greater than the end value (1) minus the increment value (1).

```
FOR i=10:1:1 WRITE i
```

The following example shows a `FOR` loop that executes the `WRITE` command 10 times then completes with `i=10`.

```
FOR i=1:1:10 WRITE i
```

The following example shows a `FOR` loop that executes the `WRITE` command 10 times then completes with `i=11`.

```
SET i=1
FOR i=1:0:10 WRITE i,!, SET i=i+1
```

Using FOR with Multiple forparameters

A single `FOR` command can contain both types of parameter syntax. The following is an example of `FOR` with multiple `forparameters`. The first `forparameter` is the `expr` syntax. The second `forparameter` is the `start:increment:end` syntax. The two `forparameters` are separated by a comma. The first time through the `FOR`, Caché uses the `expr` syntax, and invokes the Test subroutine with x equal to the value of y. In the second (and subsequent) iterations, Caché uses the `start:increment:end` syntax. It sets x to 1, then 2, etc. On the final iteration, `x=10`.

```
Mainloop
  FOR x=y,1:1:10 DO Test
  Test
```

Incrementing with Argumentless FOR

The argumentless `FOR` operates the same as the `FOR variable=start:increment` form. The only difference is that it does not provide a way to keep track of the number of loop executions. Two (or more) spaces are required after the `FOR` keyword in an argumentless `FOR` command.

The following example shows how the previous loop counter example might be rewritten using the argumentless `FOR`. The assignment `i=i+1` replaces the loop counter. Note the two spaces following the argumentless `FOR` keyword.
Average2loop
SET sum=0
SET i=0
FOR READ !,"Number: ",num QUIT:num=" SET sum=sum+num,i=i+1
SET average=sum/i
WRITE !!,"Average is: ",average
QUIT

Notes

FOR and Watchpoints

You have limited use of watchpoints with FOR. If you establish a watchpoint for the control (index) variable of a FOR command, Caché triggers the specific watchpoint action only on the initial evaluation of each FOR command argument. This restriction is motivated by performance considerations.

The following example contains three kinds of FOR command arguments for the watched variable X: a range, with initial value, increment, and limit (final value); a single value; and a range with initial value, increment, and no limit. Breaks occur when X has the initial values 1, 20, and 50.

USER>ZBREAK *X
USER>FOR X=1:1:10,20,50:2 SET T=X QUIT:X>69
<BREAK>
USER 2f0>w
X=1
USER 2f0>g
USER>FOR X=1:1:10,20,50:2 SET T=X QUIT:X>69
<BREAK>
USER 2f0>w
T=10
X=20
USER 2f0>g
USER>FOR X=1:1:10,20,50:2 SET T=X QUIT:X>69
<BREAK>
USER 2f0>w
T=20
X=50
USER 2f0>g
USER>w
T=70
X=70

Terminating a FOR Loop with QUIT or GOTO

A FOR loop is terminated by a QUIT only if the QUIT appears as one of the loop commands. If Caché encounters a QUIT in a subroutine called by the DO command, it terminates only the subroutine, not the FOR loop itself.

A FOR loop is terminated by a GOTO that appears anywhere within the loop. If a GOTO exits from a subroutine called by the DO command, Caché terminates both the subroutine and the FOR loop.

See Also

• FOR command
• DO (legacy version) command
• GOTO command
• QUIT command
IF (legacy version)

Evaluates an expression, then selects which line of code to execute based on the truth value of the expression.

<table>
<thead>
<tr>
<th>IF expression command1</th>
<th>ELSE command2</th>
</tr>
</thead>
<tbody>
<tr>
<td>I expression command1</td>
<td>E command2</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th><strong>expression</strong></th>
<th>Optional — An expression (or a comma-separated list of expressions) that evaluates to a boolean value.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>command1</strong></td>
<td>One or more Caché commands that are executed if expression evaluates to TRUE, or if expression is omitted.</td>
</tr>
<tr>
<td><strong>ELSE command2</strong></td>
<td>One or more Caché commands that are executed if expression evaluates to FALSE. The ELSE keyword is optional.</td>
</tr>
</tbody>
</table>

**Description**

*Note:* This page describes the legacy version of the `IF` command. This version is considered legacy as of Caché 4.0, and should not be used in new programming. It is described here solely for compatibility with legacy applications.

The legacy `IF` command is line-oriented; commands to be executed must follow it on the same program line. Curly braces are not used and line formatting is restrictive. The new `IF` command is block structured; the block it executes consists of commands found within the curly braces that follow the `IF` command. Line formatting (white space, line breaks) is unrestrictive. The new version of `IF` does not use the `$TEST` special variable.

The old and new forms of `IF` and `ELSE` are syntactically different and should not be combined; therefore, an `IF` of one type should not be paired with an `ELSE` of the other type.

The `IF` command has two forms:

Without an argument

With an argument

**IF Without an Argument**

`IF` without an `expression` argument executes the commands that follow it on the same line if the current value of the `$TEST` variable is TRUE (nonzero). If `$TEST` is FALSE (0), Caché ignores the remaining commands on that line and continues execution with the next line (commonly the `ELSE` command). Thus, the `ELSE` keyword is equivalent to `IF ' $TEST` (if `$TEST` is FALSE).

**IF With an Argument**

`IF` with an `expression` argument executes the commands that follow it on the same line if `expression` evaluates to TRUE (nonzero). (Caché also sets `$TEST` to TRUE.) If `expression` evaluates to FALSE, Caché sets `$TEST` to FALSE (0), ignores the remaining commands on that line, and continues execution with the next line.

The next line typically begins with an `ELSE` keyword, followed by one or more commands on the same line. The `IF` command does not require an `ELSE` keyword, but `ELSE` is required to specify an option that is only executed when `IF` is false. Always specifying `ELSE` is a recommended programming practice.
This line-oriented ELSE keyword can be abbreviated as E. (The newer block-oriented ELSE keyword cannot be abbreviated.) The old and new forms of IF and ELSE are syntactically different and should not be combined; therefore, an IF of one type should not be paired with an ELSE of the other type.

The expression argument can take the form of a single expression or a comma-separated list of expressions. For an expression list, Caché evaluates the individual expressions in left to right order. It stops evaluation if it encounters an expression that is FALSE. If all expressions evaluate to TRUE, Caché executes the command(s) following the expression argument on that line. If any expression evaluates to FALSE, Caché ignores any remaining expressions, and does not execute the commands on that line, but continues execution at the next line of code.

See Also

- IF command
- DO (legacy version) command
- GOTO command
- QUIT command
- $TEST special variable
ZQUIT (legacy command)

Exits a program with error handling.

\[
\text{ZQ:pc expression}
\]

### Arguments

<table>
<thead>
<tr>
<th>pc</th>
<th>Optional — A postconditional expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>Optional — An expression that evaluates to an integer greater than 0.</td>
</tr>
</tbody>
</table>

### Description

Caché platforms recognize only the ZQ abbreviation for ZQUIT.

**Note:** This page describes the legacy ZQUIT command. The ZQUIT command is considered legacy as of Caché 5.1, and should not be used in new programming. It is described here solely for compatibility with legacy applications. New programming should use the ZTRAP command with the $ZERROR argument to pass control between error handlers. A ZQUIT command in new procedure code will result in a <COMMAND> error. Refer to the ZTRAP command for further details.

The ZQUIT command exits a coded routine and provides explicit control for error handling.

Each time a DO, FOR, or XECUTE command is executed, or a user-defined function is invoked, Caché places return information on the program stack (sometimes called the "call stack"). ZQUIT provides a mechanism whereby a nested error-handling routine can pass control to a higher level error handler. ZQUIT re-signals the error condition and causes Caché to unwind the program (call) stack.

ZQUIT is similar to the QUIT command, but it provides explicit control for error handling. Unlike the QUIT command, ZQUIT can be used only in a coded routine. Caché ignores ZQUIT when entered from the programmer prompt.

ZQUIT has two forms:

- Without an argument.
- With an argument.

#### ZQUIT Without an Argument

Argumentless ZQUIT clears the entire stack.

#### ZQUIT With an Argument

ZQ expression unwinds the stack to another call stack level with a $ZTRAP error handler. The value of expression specifies the number of handler-specified stack levels ZQUIT unwinds. If you use 1 as expression, Caché unwinds the stack to the first encountered $ZTRAP handler. If you use 2 as expression, Caché unwinds the stack to the second encountered $ZTRAP handler, and so on.

### Arguments

<table>
<thead>
<tr>
<th>pc</th>
</tr>
</thead>
</table>

An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.
expression

Any valid ObjectScript expression that evaluates to an integer greater than 0. Caché platforms use the settings of $ZTRAP to clear the program stack back to the setting indicated by the resulting integer.

Example

The following is an example of a magnetic-tape read routine that reads a record into x, returns a tapemark flag (tm) TRUE (1) if it reads a tape mark (an "expected" error), and lets its caller's errortrap routine, if any, handle any other errors:

```objectscript
Mtread
SET $ZTRAP="Mterr"
USE 47
READ x
SET tm=0
QUIT ; Normal return
Mterr
IF $ZERROR?1"<MAGTAPE>" ; Pattern matching: repcount stringliteral { } ; Automatic branch to error-trap handler ELSE { SET tm=123 }
IF tm=1 { QUIT }
ELSE {ZQ 1 }
IF $ZTRAP'="" {GOTO @$ZTRAP} ; To caller's error routine ELSE { ; No caller's error routine USE 0
WRITE !,$ZERROR ZQ
QUIT }
```  

Notes

**Behavior of Argumentless ZQUIT**

When specified with no arguments, ZQUIT unwinds the entire stack. Execution continues at the main program level; that is, the level at which the job began, in either programmer mode or application mode.

ZQUIT does not change the flow of control. Execution continues with the ObjectScript command that follows ZQUIT. To mimic the errortrap behavior of M/11, an error processing routine should end with the commands

```
ZQ  QUIT
```

Note that there are two spaces after ZQ.

Here the QUIT command returns to programmer mode if the job began in programmer mode or it exits if the job began in application mode. To log application errors, end your errortrap code with:

```
ZQ  GOTO ^%ET
```

Again, note the two spaces after ZQ.

**Behavior of ZQUIT with an Argument**

When specified with an argument, ZQUIT does not clear the entire stack, but instead clears it back one or more levels to the level at which a $ZTRAP special variable was set. The number of levels is specified by the expression argument, which must evaluate to an integer greater than 0. $ZTRAP directs error trapping to the routine to which it is set.

For example:

```
ZQ 1
```

clears the stack back to the previous level at which $ZTRAP was set. Passing an argument of "2" to ZQUIT clears the stack back to the second to last level at which $ZTRAP was set, and so on.
Caché continues program execution on the execution level that contains the errortrap routine currently set in $ZTRAP. When that errortrap routine quits, control reverts to the same place that it would on a QUIT from the subroutine that set $ZTRAP.

If Caché should find an error handler specified by the $ETRAP special variable at a stack level before reaching the stack level of the specified $ZTRAP error handler, Caché passes control to that $ETRAP error handler. If the $ETRAP error handler cannot dismiss the error, the $ZTRAP error handler that issued the original ZQUIT gains control again. At that point, the $ZTRAP error handler can use a GOTO command to transfer control to the $ZTRAP handler originally specified.

**Note:** Do not use an indirect GOTO (for example, ZQ 1 GOTO @$ZT) within a procedure or an object method with the default ProcedureBlock attribute. This results in a compile error, because you cannot use a GOTO to exit a procedure. Existing routines that use this construct should not be converted to procedures. Object methods that use this construct must be excepted from the ProcedureBlock attribute.

You can alter this error-return sequence by using ZQUIT in any of the following ways:

- **ZQ 1 QUIT** — Returns to the caller of the subroutine that set the previous error trap.
- **ZQ 1 GOTO @$ZT** — Invokes the previous errortrap routine. (See Note above.)
- **ZQ QUIT** — Either halts the application or enters programmer mode with an empty program stack. Note the two spaces between ZQUIT and QUIT.
- **ZQ GOTO ^routine** — Clears the program stack and resumes execution at the specified top-level routine that is typically a function driver. Note the two spaces between ZQUIT and GOTO.

**See Also**

- QUIT command
- ZTRAP command
- $SZTRAP special variable
- Error Handling in *Using Caché ObjectScript*
ZSYNC (legacy command)

Synchronizes and forces the end of networked transactions for the current job.

ZSYNC:pc

**Argument**

| pc | Optional — A postconditional expression. |

**Description**

**Note:** This page describes the legacy ZSYNC command. The ZSYNC command is considered legacy as of Caché 2010.2 because its only use is to support the obsolete Distributed Cache Protocol (DCP). ZSYNC should not be used in new programming. It is described here solely for compatibility with legacy applications.

ZSYNC synchronizes and forces the conclusion of all network transactions for the current job. You can also use ZSYNC in non-networked applications that might later be run in a networked environment. Simply place it at the end of each complete transaction.

ZSYNC ensures that if multiple updates to a global (SET or KILL operations) are synchronized by LOCK operations, that READs of that global return those updates in the correct order. It does this by synchronizing delock operations across a network. When a delock occurs on an existing lock, any updates to a global that have been initiated (on any system) before the delock is initiated will be completed before the delock completes.

**Note:** ZSYNC is used with Distributed Cache Protocol (DCP). ZSYNC performs no operation on a system configured for an Enterprise Cache Protocol (ECP) data server. ECP uses synchronous-commit transactions to perform similar synchronization functionality. Because ECP is commonly used with Caché server system networks and clusters, your use of ZSYNC may be limited. For further details on ECP, see the Caché Distributed Data Management Guide.

**Argument**

**pc**

An optional postconditional expression. Caché executes the command if the postconditional expression is true (evaluates to a nonzero numeric value). Caché does not execute the command if the postconditional expression is false (evaluates to zero). For further details, refer to Command Postconditional Expressions in Using Caché ObjectScript.

**Notes**

**ZSYNC Guarantees Return of Asynchronous Errors**

Depending on the operation requested, network errors are reported at different times. Errors for synchronous operations, such as LOCK, $GET, $ORDER and $QUERY, are reported immediately through the normal error trap on the command that performed the error.

Errors for asynchronous operations, such as SET and KILL, may be reported either immediately or after a delay.

Delayed errors return as a result of buffered asynchronous operations. When a job invokes either SET or KILL, processing can continue on the client while the operation on the server completes. In this case, a special interrupt trap may occur sometime later in the program. The error code returned is identical to the error that originally occurred on the server.

When you use the ZSYNC command to synchronize the activity of clients and servers, you force the return of any asynchronous errors. That is to say, all global errors return to the job before the ZSYNC command completes.
You can control whether asynchronous errors can interrupt a process on a per-process basis using the `AsynchError()` method of the `%SYSTEM.Process` class. The system-wide default behavior can be established by setting the `AsynchError` property of the `Config.Miscellaneous` class.

**ZSYNC Guarantees Transaction Completion**

The `ZSYNC` command lets applications synchronize activity on network servers. Executing this command forces the completion of all network transactions for this job. Do not use this command at the end of an update transaction that completes the filing of all global nodes on all servers, because it forces any delayed errors to be returned. In other words, all global errors are delivered to the job before the `ZSYNC` command completes.

**See Also**

- Using ObjectScript for Transaction Processing in *Using Caché ObjectScript*
$ZBITAND (legacy function)

Bitstring function – AND.

$ZBITAND(bitstring1,bitstring2)

Description
This page describes the legacy function $ZBITAND. It is described here for compatibility with legacy applications.

$ZBITAND returns a bitstring formed by the (bitstring1 AND bitstring2) operation. The length of the resulting bitstring equals the length of the shorter of the two specified bitstrings. The value of the resulting bitstring is 1 in all positions where both bitstrings held a 1, and 0 in all other positions.

Example
If bitstring1 = [0,1,1,0] and bitstring2 = [0,1,0,0], then the result of $ZBITAND would be [0,1,0,0]. That is:

Position 1: 0 and 0 = 0
Position 2: 1 and 1 = 1
Position 3: 1 and 0 = 0
Position 4: 0 and 0 = 0

See Also
- $BITLOGIC function
- $ZBOOLEAN function
- $ZBITOR (legacy function) function
- $ZBITXOR (legacy function) function
$ZBITCOUNT (legacy function)

Bitstring function – COUNT.

$ZBITCOUNT(bitstring)

Description
This page describes the legacy function $ZBITCOUNT. It is described here for compatibility with legacy applications. $ZBITCOUNT returns the number of ON bits (1's) in bitstring.

Example
If bitstring = [0,0,1,1,0], then the result of $ZBITCOUNT would be 2.

See Also
• $BITCOUNT function
• $ZBOOLEAN function
$ZBITFIND (legacy function)

Bitstring function – FIND.

$ZBITFIND(bitstring, truthval, position)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitstring</td>
<td>A bitstring.</td>
</tr>
<tr>
<td>truthval</td>
<td>Binary value (0 or 1).</td>
</tr>
<tr>
<td>position</td>
<td>Optional — Specifies a starting bit position in bitstring.</td>
</tr>
</tbody>
</table>

**Description**

This page describes the legacy function $ZBITFIND. It is described here for compatibility with legacy applications.

$ZBITFIND returns the position immediately after the first bit in bitstring that equals truthval. position specifies an optional starting position for the search. If position is omitted, the search starts at position 1.

If truthval is not found, $ZBITFIND returns zero.

$ZBITFIND is similar to the $FIND function in that the returned position is the next one after the matching bit. This permits a loop of the form:

```
Findabit
  FOR p=1:0 {
    SET p=$ZBITFIND(bstring,1,p)
    QUIT: p
    DO Foundone
  }
Foundone
```

**Parameters**

**truthval**

Binary value (0 or 1).

**position**

Specifies a starting bit position in bitstring. The default is 1. Bits are counted left-to-right, with the first bit counted as number 1.

**Examples**

Note that the result is the position immediately after the bit that matches truthval.

Given the bitstring[0,0,1,1,0]:

If truthval=0 then the result of $ZBITFIND would be position 2.

If truthval=1 then the result of $ZBITFIND would be position 4.

If truthval=0 and position=2, then the result of $ZBITFIND would be position 3.

If truthval=0 and position=3, then the result of $ZBITFIND would be position 6.

**See Also**

- $ZBOOLEAN function
$ZBITGET (legacy function)

Bitstring function – GET.

$ZBITGET(bitstring, position)

Parameters

<table>
<thead>
<tr>
<th>bitstring</th>
<th>The bitstring to be evaluated.</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>Value or expression that evaluates to a positive integer.</td>
</tr>
</tbody>
</table>

Description

This page describes the legacy function $ZBITGET. It is described here for compatibility with legacy applications. $ZBITGET returns the bit value (0 or 1) at the specified bit position within bitstring. The first bit position is 1.

Parameters

position

Value or expression that evaluates to a positive integer. Bit positions are counted left-to-right, with the first bit counted as number 1. If position is set to zero, or to an integer larger than the length of bitstring, a <FUNCTION> error occurs.

Examples

Given the bitstring [0,0,1,1,0]:

If position=1, then the result of $ZBITGET would be 0.
If position=2, then the result of $ZBITGET would be 0.
If position=3, then the result of $ZBITGET would be 1.
If position=4, then the result of $ZBITGET would be 1.
If position=5, then the result of $ZBITGET would be 0.
If position=6, then $ZBITGET would generate a <FUNCTION> error.

See Also

- $BIT function
- $ZBOOLEAN function
- $ZBITSET (legacy function)
$ZBITLEN (legacy function)

Bitstring function – LENGTH.

$ZBITLEN(bitstring)

**Description**
This page describes the legacy function $ZBITLEN. It is described here for compatibility with legacy applications. $ZBITLEN returns the number of bits in bitstring.

**Example**
If bitstring=[0,0,1,1,0], then the result of $ZBITLEN would be the length 5.

**See Also**
- $BITCOUNT function
- $ZBOOLEAN function
$ZBITNOT (legacy function)

Bitstring function – NOT

$ZBITNOT(bitstring)

Description

This page describes the legacy function $ZBITNOT. It is described here for compatibility with legacy applications.

$ZBITNOT returns a permutation of bitstring with each of its bit positions inverted -- that is, 1s become 0s, and 0s become 1s.

Example

If bitstring=[0,0,1,1,0], then the result of $ZBITNOT would be a bitstring with the value [1,1,0,0,1].

See Also

• $BITLOGIC function
• $ZBOOLEAN function
$ZBITOR (legacy function)

Bitstring function – OR.

$ZBITOR(bitstring1,bitstring2)

Description

This page describes the legacy function $ZBITOR. It is described here for compatibility with legacy applications.

$ZBITOR returns a bitstring formed by the (bitstring1 OR bitstring2) operation. The length of the resulting bitstring equals the longer length of the two bitstrings. The value of the resulting bitstring is 1 in all positions where either or both of the bitstrings held a 1, and 0 in all other positions. Bitstrings are compared left-to-right.

Example

If bitstring1=[0,0,1,1,0] and bitstring2=[0,1,1,0], then the result of $ZBITOR would be [0,1,1,1,0]. That is:

- Position 1: 0 OR 0 = 0
- Position 2: 0 OR 1 = 1
- Position 3: 1 OR 1 = 1
- Position 4: 1 OR 0 = 1
- Position 5: 0 OR null = 0

See Also

- $BITLOGIC function
- $ZBOOLEAN function
- $ZBITXOR (legacy function) function
- $ZBITAND (legacy function) function
$ZBITSET (legacy function)

Bitstring function – SET.

$ZBITSET(bitstring,position,truthval)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitstring</td>
<td>The bitstring to be set.</td>
</tr>
<tr>
<td>position</td>
<td>Positive integer that specifies a location in bitstring.</td>
</tr>
<tr>
<td>truthval</td>
<td>Single-bit binary value (0 or 1).</td>
</tr>
</tbody>
</table>

Description

This page describes the legacy function $ZBITSET. It is described here for compatibility with legacy applications.

$ZBITSET returns a permutation of bitstring in which the bit at position is set to truthval (0 or 1).

Parameters

position

Positive integer that specifies a location in bitstring. It can be specified as a value, a variable, or an expression. Bits are counted left-to-right, with the first bit counted as number 1.

truthval

Single-bit binary value (0 or 1).

Example

The following example creates a bitstring using $ZBITSTR, then sets a bit using $ZBITSET.

SET bitstring=$ZBITSTR(5,0)
SET newbitstring=$ZBITSET(bitstring,3,1)

If bitstring=[0,0,0,0,0], position=3, and truthval=1, then the result of $ZBITSET would be a bitstring with the value [0,0,1,0,0].

See Also

- $BIT function
- $ZBOOLEAN function
- $ZBITGET (legacy function)
**$ZBITSTR (legacy function)**

Bitstring function – STRING.

$ZBITSTR(size,truthval)

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>Positive integer.</td>
</tr>
<tr>
<td>truthval</td>
<td>Optional — Specifies a binary value (0 or 1) for the returned bits.</td>
</tr>
</tbody>
</table>

**Description**

This page describes the legacy function $ZBITSTR. It is described here for compatibility with legacy applications.

$ZBITSTR returns a bitstring whose length is size bits long, with all of its bits set to truthval.

**Parameters**

**size**

Positive integer. The integer must be in the range 1 to 262,128 bits (32766 x 8). It can be specified as a value, a variable, or an expression.

**truthval**

Specifies a binary value (0 or 1) for the returned bits. If truthval is 0, or is omitted, all the bits are set to 0 (OFF). If truthval is 1, all the bits are set to 1 (ON).

**Example**

The following example creates a bitstring named bs:

```objectscript
SET bs=$ZBITSTR(4,1)
```

If size=4 and truthval=1, then the result of $ZBITSTR would be a bitstring with the value [1,1,1,1].

**See Also**

- $ZBOOLEAN function
$ZBITXOR (legacy function)

Bitstring function – XOR.

$ZBITXOR(bitstring1,bitstring2)

Description

This page describes the legacy function $ZBITXOR. It is described here for compatibility with legacy applications.

$ZBITXOR returns a bitstring formed by the (bitstring1 XOR bitstring2) operation. The length of the resulting bitstring equals the shorter length of the two bitstrings.

The following table shows the four possible results obtained by performing the XOR operation on two bits (b1 and b2):

<table>
<thead>
<tr>
<th>b1</th>
<th>b2</th>
<th>equals</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Example

If bitstring 1=[0,0,1,1,0] and bitstring 2=[0,1,1,0], then the result of $ZBITXOR would be [0,1,0,1]. That is:

Position 1: 0 XOR 0 = 0
Position 2: 0 XOR 1 = 1
Position 3: 1 XOR 1 = 0
Position 4: 1 XOR 0 = 1
Position 5: 0 XOR null = null

See Also

- $BITLOGIC function
- $ZBOOLEAN function
- $ZBITOR (legacy function) function
- $ZBITAND (legacy function) function
$ZINCREMENT (legacy function)

Adds a specified increment to the existing value of a global or local variable.

$ZINCREMENT(variable,num)

Parameters

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>The variable whose value is to be incremented.</td>
</tr>
<tr>
<td>num</td>
<td>Optional — The numeric increment you want to add to variable. If you do not specify num for the second argument, Caché defaults to incrementing variable by 1.</td>
</tr>
</tbody>
</table>

Description

This page describes the legacy function $ZINCREMENT. It is described here for compatibility with legacy applications. $ZINCREMENT is an alias of $INCREMENT, which is the function call that should be used for new applications.

See Also

- $INCREMENT function
$ZNEXT (legacy function)

Returns a full reference to the next array node.

$ZNEXT(global-reference)

**Parameter**

| global-reference | A subscripted global array node. The subscript is required because it provides the starting point. You cannot specify just the array name. |

**Description**

This page describes the legacy function $ZNEXT. It is described here for compatibility with legacy applications. $ZNEXT is similar to the $QUERY function, except that $ZNEXT uses “–1” as the starting point and failure code, while $QUERY uses the null string (""). The $QUERY function should be used for new applications.

$ZNEXT returns a full reference to the array node defined next in the collating sequence after the specified array node. $ZNEXT accepts a naked global reference. For further details, see Naked Global Reference in *Using Caché Globals.*

**Examples**

The following example uses $ZNEXT to list the subscripts of a global:

```plaintext
ZNSPACE "samples"
SET x="CinemaproFilmCategoryI(-1)
Loop
  SET a=$ZNEXT(x)
  IF a=-1 {
    WRITE !,"All done"
    QUIT }
  ELSE {
    SET x=a
    WRITE !,a
    GOTO Loop }
```

The following example uses $QUERY to list the subscripts of a global:

```plaintext
ZNSPACE "samples"
SET x="CinemaproFilmCategoryI("
Loop
  SET a=$QUERY(x)
  IF a="" {
    WRITE !,"All done"
    QUIT }
  ELSE {
    SET x=a
    WRITE !,a
    GOTO Loop }
```

**See Also**

- $QUERY function
$ZORDER (legacy function)

Returns the full reference for the next array node subscript.

$ZORDER(variable)

Parameter

| variable | Name of a subscripted local or global variable, or the null string. It can be a value, a variable, or an expression. |

Description

This page describes the legacy function $ZORDER. It is described here for compatibility with legacy applications. $ZORDER is identical to the $QUERY function. New programs should use $QUERY instead of $ZORDER.

$ZORDER returns the full reference for the next array node subscript that follows the node indicated by the variable parameter. The variable can be a local or global subscripted array node. The returned subscript is at the same level as the one for the specified variable. If there is no subsequent array node at the variable’s node level, $ZORDER returns the null string (""").

Notes

$ZORDER and the Naked Indicator

$ZORDER can be used with a naked global reference. If the naked indicator is undefined, $ZORDER generates a <NAKED> error. If the naked indicator references the last element in an array, $ZORDER generates an <UNDEFINED> error.

When you attempt to use the naked indicator after $ZORDER has generated an <UNDEFINED> error at the end of an array, the naked indicator references the last element in the array.

See Also

• $QUERY function
$ZPREVIOUS (legacy function)

Returns the previous array node subscript for the specified variable.

\[
\begin{array}{|c|}
\hline
\text{\$ZPREVIOUS (variable)} \\
\text{\$ZP (variable)} \\
\hline
\end{array}
\]

**Parameter**

<table>
<thead>
<tr>
<th>variable</th>
<th>Name of local or global array variable or the null string. It can be specified as a value, a variable, or an expression.</th>
</tr>
</thead>
</table>

**Description**

This page describes the legacy function $ZPREVIOUS. It is described here for compatibility with legacy applications. Use of $ZPREVIOUS is discouraged. Rather, use the form $ORDER(variable,-1).

$ZPREVIOUS returns the previous array node subscript for the specified variable. $ZPREVIOUS is similar to the $ORDER function, except that it returns the previous, rather than the next, subscript in the collation sequence.

You can use it to traverse a global in reverse subscript order. To start at the last subscript on a given level, specify the null string for the subscript reference. $ZPREVIOUS returns the null string after it returns the first subscript on the root level.

As with $ORDER, you can use $ZPREVIOUS on both local and global variables.

**Example**

Given the array node definitions shown, and starting with the last subscript, $ZPREVIOUS returns the following results:

```objectscript
SET ^ABC(1)=1
SET ^ABC(2)=2
SET ^ABC(3)=3
SET x=$ZPREVIOUS(^ABC(“”)) ; 3
WRITE !,"Third is ",x
SET x=$ZPREVIOUS(^ABC(x)) ; 2
WRITE !,"Second is ",x
SET x=$ZPREVIOUS(^ABC(x)) ; 1
WRITE !,"First is ",x
SET x=$ZPREVIOUS(^ABC(x)) ; ""
WRITE !,"Top is ",x
```

**See Also**

- $ORDER function
- $ZNEXT function
- $ZREFERENCE special variable
$ZSORT (legacy function)

Returns the next subscript in the array of the specified subscripted variable.

$ZSORT(variable,direction)

Parameters

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>A subscripted local or global variable. The subscript is required; you cannot specify just the array name.</td>
</tr>
<tr>
<td>direction</td>
<td>Optional — The subscript order in which to traverse the target array. Use 1 (the default) for ascending subscript order and -1 for descending subscript order.</td>
</tr>
</tbody>
</table>

Description

This page describes the legacy function $ZSORT. It is described here for compatibility with legacy applications. $ZSORT is identical to the $ORDER function in Caché, which is the function call that should be used for new applications.

$ZSORT returns the next subscript in the subscripted variable. The variable parameter can specify a subscripted local or global variable.

See Also

- $ORDER function
- $ZORDER function