APPENDIX B

ColdFusion Function Reference

In this appendix

Using ColdFusion Functions 132

String Manipulation Functions 133

`Date` and `Time` Functions 146

Data Formatting Functions 164

Mathematical Functions 172

International Functions 174

List Manipulation Functions 181

Array Manipulation Functions 187

Structure Manipulation Functions 195

Query Manipulation Functions 200

Security Functions 202

System Functions 204

Client Variable Manipulation Functions 208

Expression Evaluation Functions 209

Bit and Set Manipulation Functions 211

Conversion Functions 211

Miscellaneous Functions 213

USING COLDFUSION FUNCTIONS

ColdFusion provides a complete set of data manipulation and formatting functions. Here are some things to remember when using functions:

- Function names are not case sensitive, so `NOW()` is the same as `now()`, which is the same as `Now()`.
- When functions are used in body text rather than within a ColdFusion tag, they must be enclosed within `<CFOUTPUT>` tags.
- Functions can be nested.

To make this function reference easier to use, the functions have been grouped into logical sets. Because some functions overlap these groupings, each function has a "see also" list of related functions.

- **String manipulation functions**—A complete set of text parsing, comparison, and conversion functions
- **Date and time functions**—Can be used to create, parse, compare, and manipulate date and time values
- **Data formatting functions**—Enable you to display data in a variety of formats
- **Mathematical functions**—Can be used to perform calculations, perform conversions, and generate random numbers
- **International functions**—Provide localization support for dates, times, and other data types
- **List manipulation functions**—Used to control lists of values
- **Array manipulation functions**—Used to create and manage two- and three-dimensional arrays
- **Structure manipulation functions**—Used to create and manage ColdFusion structures
- **Query manipulation functions**—Used to create and manage ColdFusion queries
- **Security functions**—Give you access to security information returned by a `<CFAUTHENTICATE>` call
- **System functions**—Give you access to system directories, temporary files, and path manipulation functions
- **Client variable manipulation functions**—Enable you to control client variables
- **Expression evaluation functions**—Enable you to create and evaluate expressions on-the-fly
- **Bit and set manipulation functions**—Can be used to perform bit-level operations
- **Conversion functions**—Used to convert data from one form to another
- **Miscellaneous functions**—An assortment of functions that you can use to check for the existence of parameters, format URLs, and manipulate lists to be passed to SQL statements

STRING MANIPULATION FUNCTIONS

The ColdFusion string manipulation functions can be used to perform operations on character data. Strings can be hard-coded constants, table column values, or ColdFusion fields. As with all ColdFusion functions, these string manipulation functions can be nested.

Asc()

Description: Asc returns the ASCII value of the leftmost character of a string. The Asc() function will return 0 if the string being evaluated is empty.

Syntax:

```
Asc(character)
```

Example: The following example returns 72, the ASCII value of the character H:

```
Asc("Hello")
```

Tip

The Asc() function processes only the leftmost character in a string. To return the ASCII characters of an entire string, you must loop through the string and process each character individually.

See also: Chr(), Val()

Chr()

Description: Chr() converts an ASCII value into a printable character. The Chr() function takes a single parameter, the ASCII value to be converted (valid ASCII values range from 0 to 255), and returns the specified ASCII value as a printable character.

Syntax:

```
Chr(number)
```

```
Example: The following example returns the letter H, whose ASCII value is
72:Chr(72)
```

See also: Asc(), Val()

Cjustify()

Description: CJustify() centers a string within a field of a specified length. It does this by padding spaces before and after the specified text. Cjustify() takes two parameters, the string to process and the desired string length.

Syntax: CJustify(string, length)

Example: The following example justifies the word `Hello` so that it is centered within a 20-character–wide field:

```
CJustify("Hello", 20)
```

See also: `Ljustify()`, `Ltrim()`, `Rjustify()`, `Rtrim()`, `Trim()`

`Compare()`, `CompareNoCase()`

Description: The `Compare()`and `CompareNoCase()` functions compare two string values. `Compare()` performs a case-sensitive comparison; `CompareNoCase()` performs a non–case-sensitive function. Both of these functions return a negative number if the first string is less than the second string, a positive number if the first string is greater than the second string, and `0` if the strings are the same.

Syntax:

```
Compare(String1, String2)
CompareNoCase(String1, String2)
```

Example: This example returns a negative value because the first string is less than the second string:

```
#Compare("Ben", "Bill")#
```

This next example uses the non–case-sensitive comparison function and returns `0` because, aside from case, the strings are the same:

```
#CompareNoCase("Michigan", "MICHIGAN")#
```

Note

The two comparison functions treat whitespace as characters to be compared. Therefore, if you compare two strings that are identical except for extra spaces at the end of one of them, the compare will not return `0`.

Tip

You can create an alphabetical list of strings easily by sorting all the strings in increasing order with the `Compare()` function.

`Decrypt()`, `Encrypt()`

Description: The `Encrypt()` and `Decrypt()` functions work in parallel to enable you to encode and decode strings based on a user-specified key. The `Encrypt()` and `Decrypt()` functions use a symmetric-key–based algorithm. This means that the same key that was used to encrypt the string must be used to decrypt the string. Both the `Encrypt()` and `Decrypt()` functions take two parameters. The first parameter is the string on which to perform encryption or decryption. The second parameter is the key with which to encrypt or decrypt the string.

Syntax:

```
Encrypt(string, seed)
Decrypt(encrypted_string, seed)
```

Example: The following example encrypts a person's name with the key `butterfly` and subsequently decrypts that same string:

```
Encrypt("John", "butterfly")
```

Tip

Remember that when encrypting a string, the string will be UUEncoded after encryption; therefore, the size of the encrypted string can be as much as three times larger than the original string.

`Find()`, `FindNoCase()`, `REFind()`, `REFindNoCase()`

Description: ColdFusion provides four functions to search for specific text within another string. `Find()` performs a case-sensitive search, `FindNoCase()` performs a non–case-sensitive search, `REFind()` performs a case-sensitive search using regular expressions, and `REFindNoCase()` performs a non–case-sensitive search using regular expressions. The parameters for all these functions are the same. The first parameter is the string (or regular expression) for which to search, and the second parameter is the target string, or string to be searched. An optional third parameter can be provided to specify the position in the target string from which to start the search. All these functions return the starting position of the first occurrence of the search string within the specified target string. If the search string is not found, `0` is returned.

Syntax:

```
Find(SearchString, TargetString [, StartPosition])
FindNoCase(SearchString, TargetString [, StartPosition])
REFind(RegularExpression, TargetString [, StartPosition])
REFindNoCase(RegularExpression, TargetString [, StartPosition])
```

Example: This first example returns `18`, the start position of the word `America`:

```
#Find("America", "United States of America")#
```

The next example returns `0` because `Find()` performs a case-sensitive search:

```
#Find("AMERICA", "United States of America")#
```

The `FindNoCase()` function performs a non–case-sensitive search, as this example shows:

```
#FindNoCase("AMERICA", "United States of America")#
```

The next example searches for the word `of` in the string `The Flag of the United States of America` and specifies that the search should start from position `15`. This example returns `31`, the position of the second `of`. Had the optional start position parameter been omitted, the return value would have been `10`, the position of the first `of`:

```
#Find("of", "The Flag of the United States of America", 15)#
```

Tip

Regular expressions enable you to perform complex searches through text strings with relative ease. Rather than being forced to provide the exact text for which you are searching, using the `REFind` and `REFindNoCase` functions gives you the flexibility of searching for dynamic data. Suppose you wanted to search a string for the first occurrence of any character except for vowels. In this case, using `REFindNoCase` would make your job much easier, as this example demonstrates:

```
#REFindNoCase("[aeiou]", "somestring", "1")#
```

See also: `FindOneOf()`, `GetToken()`, `Left()`, `Mid()`, `Right()`

FindOneOf()

Description: `FindOneOf()` returns the position of the first target string character that matches any of the characters in a specified set. `FindOneOf()` takes three parameters. The first parameter is a string containing the set of characters for which to search, the second parameter is the target string (the string to be searched), and the third parameter is an optional starting position from which to start the search. These functions return the starting position of the first occurrence of any characters in the search set within the specified target string. If no matching characters are found, `0` is returned.

Syntax:

```
FindOneOf(SearchSet, TargetString, [, StartPosition])
```

Example: The following example returns the position of the first vowel with a ColdFusion field called `LastName`:

```
The first vowel in your last name is at position #FindOneOf("aeiou", LastName)#
```

Tip

The `FindOneOf()` function is case sensitive, and there is no non–case-sensitive equivalent function. To perform a non–case-sensitive `FindOneOf()` search, you first must convert both the search and target strings to either upper- or lowercase (using the `UCase()` or `LCase()` function).

See also: `Find()`, `FindNoCase()`

GetToken()

Description: *Tokens* are delimited sets of data within a string. The `GetToken()` function enables you to extract a specific token from a string by specifying the *token number*, or index. `GetToken()` takes three parameters. The first is the string to search; the second is the index of the token to extract, so `3` will extract the third token and `5` will extract the fifth. The third parameter is an optional set of delimiters `GetToken()` uses to determine where each token starts and finishes. If the delimiter's parameter is not provided, the default of spaces, tabs, and newline characters is used. The default delimiters effectively enable this

function to be used to extract specific words for a string. `GetToken()` returns the token in the specified position or any empty string if `Index` is greater than the number of tokens present.

Syntax:

```
GetToken(String, Index [, Delimiters])
```

Example: The following example uses a hyphen as a delimiter to extract just the area code from a phone number:

```
#GetToken("800-555-1212", 1, "-")#
```

Tip

Use the ColdFusion list functions instead of `GetToken()` when working with strings that contain lists of data.

Hash()

Description: The `Hash()` function converts a string into a 32-byte hexadecimal string using the one-way MD5 algorithm. Because this conversion is one-way, a string that has been converted to hexadecimal using the `Hash()` function can't be converted back to its original form. The `Hash()` function takes a single parameter, the string to be converted using `Hash()`.

Syntax:

```
Hash(string)
```

Example: The following example hashes a user-defined password value:

```
#Hash("#form.password#")#
```

Tip

The `Hash()` function is useful for converting sensitive data (such as user passwords) into a hexadecimal string for storage in the database. If you store the converted value in the database, you can then have the user reenter her password, hash the value she enters, and compare the hexadecimal string returned against the hexadecimal value stored in the database.

InputBaseN()

Description: `InputBaseN()` converts a string into a number using the base specified by radix. Valid radix values are 2–36.

Syntax:

```
InputBaseN(String, Radix)
```

Example: The following example converts the string containing the binary number `10100010` into its base-10 equivalent of `162`:

```
#InputBaseN("10100010", 2)#
```

Tip

The code `InputBaseN(String, 10)` is functionally equivalent to the code `Val(String)`. If you are converting a number that is base 10, the `Val()` function is simpler to use.

See also: `FormatBaseN()`, `Val()`

`Insert()`

Description: `Insert()` is used to insert text into a string and takes three parameters. The first parameter, `SourceString`, is the string you want to insert. The second parameter, `TargetString`, is the string into which you will insert `SourceString`. The third parameter, `Position`, is a numeric value that specifies the location in the `TargetString` at which to insert the `SourceString`. `Insert()` returns the modified string.

Syntax:

```
Insert(SourceString, TargetString, Position)
```

Example: The following example inserts a field called area code in front of a phone number:

```
#Insert(area_code, phone, 0)#
```

Tip

To insert a string at the very beginning of another, use the `Insert()` function specifying a `Position` of `0`.

See also: `RemoveChars()`, `SpanExcluding()`, `SpanIncluding()`

`LCase()`

Description: `LCase()` converts a string to lowercase. `LCase()` takes a single parameter, the string to be converted, and returns the converted string.

Syntax:

```
LCase(String)
```

Example: The following example converts a user-supplied string to lowercase:

```
#LCase(string_field)#
```

See also: `UCase()`

`Left()`

Description: `Left()` returns the specified leftmost characters from the beginning of a string. `Left()` takes two parameters: the string from which to extract the characters and the number of characters to extract.

Syntax:

```
Left(String, Count)
```

Example: The following example returns the first three characters of a phone number column:

```
#Left(phone_number, 3)#
```

See also: `Find()`, `Mid()`, `RemoveChars()`, `Right()`

Len()

Description: `Len()` returns the length of a specified string. `Len()` takes a single parameter, the string whose length you want to determine.

Syntax:

```
Len(String)
```

Example: The following example returns the length of a user-supplied address field after it has been trimmed:

```
#Len(Trim(address))#
```

LJustify()

Description: `LJustify()` left-aligns a string within a field of a specified length. It does this by padding spaces after the specified text. `LJustify()` takes two parameters: the string to process and the desired string length.

Syntax:

```
LJustify(String, Length)
```

Example: The following example left-justifies the string `"First Name:"` so that it is left-aligned within a 25-character–wide field:

```
#LJustify("First Name:", 25)#
```

See also: `CJustify()`, `LTrim()`, `RJustify()`, `RTrim()`, `Trim()`

LTrim()

Description: `LTrim()` trims whitespace (spaces, tabs, and newline characters) from the beginning of a string. `LTrim()` takes a single parameter, the string to be trimmed.

Syntax:

```
LTrim(String)
```

Example: The following example trims spaces from the beginning of a table note field:

```
#LTrim(notes)#
```

See also: `CJustify()`, `LJustify()`, `RJustify()`, `RTrim()`, `Trim()`, `StripCR()`

Mid()

Description: Mid() returns a string of characters from any location in a string. Mid() takes three parameters. The first is the string from which to extract the characters, the second is the desired characters' starting position, and the third is the number of characters required.

Syntax:

```
Mid(String, StartPosition, Count)
```

Example: The following example extracts eight characters from the middle of a table column, starting at position 3:

```
#Mid(order_number, 3, 8)#
```

See also: Find(), Left(), RemoveChars(), Right()

RemoveChars()

Description: RemoveChars() returns a string with specified characters removed from it. This function is the exact opposite of the Mid() function. RemoveChars() takes three parameters: The first is the string from which to remove the characters, the second is the starting position of the characters to be removed, and the third is the number of characters to be removed. If no characters are found in the string being evaluated, the RemoveChars function returns 0.

Syntax:

```
RemoveChars(String, StartPosition, Length)
```

Example: The following example returns a field with characters 10 through 15 removed:

```
#RemoveChars(product_code, 10, 5)#
```

See also: Left(), Mid(), Right()

RepeatString()

Description: RepeatString() returns a string made up of a specified string multiple times. RepeatString() takes two parameters: The first is the string to repeat, and the second is the number of occurrences.

Syntax:

```
RepeatString(String, Count)
```

Example: The following example creates a horizontal line made up of equal signs:

```
#RepeatString("=", 80)#
```

Replace(), REReplace(), REReplaceNoCase()

Description: Replace(), REReplace(), and REReplaceNoCase() enable you to replace text within strings with alternative text. Replace() does a simple text comparison to locate the text to be replaced; REReplace() and REReplaceCase() perform the same operation but use regular expressions.

All three functions take four parameters. The first parameter is the string to be processed, the second is the text to be replaced, and the third is the text to replace it with. The fourth parameter is optional and specifies the scope of the replacements. Possible scope values are `"ONE"` to replace the first occurrence only, `"ALL"` to replace all occurrences, and `"RECURSIVE"` to replace all occurrences recursively.

Syntax:

```
Replace(String, WhatString, WithString [, Scope])
REReplace(String, WhatString, WithString [, Scope])
REReplaceNoCase(String, WhatString, WithString [, Scope])
```

Example: The following example replaces all occurrences of the text `"US"` in an address field with the text `"USA"`:

```
#Replace(address, "US", "USA", "ALL")#
```

This next example replaces the area code `"(313)"` with the area code `"(810)"`, and because no scope is specified, only the first occurrence of `"(313)"` is replaced:

```
#Replace(phone, "(313)", "(810)")#
```

Tip

The `Replace()` function is case sensitive, and there is no non–case-sensitive equivalent function. To perform a non–case-sensitive replacement, you first must convert both the search and target strings to either upper- or lowercase (using the `UCase()` or `LCase()` function).

```
<P>REReplaceNoCase("John","J|N","C","ALL"):
<CFOUTPUT>#REReplaceNoCase("John","J|N","C","ALL")#</CFOUTPUT>
<P>REReplaceNoCase("John","[A-Z]","C","ALL")
<CFOUTPUT>#REReplaceNoCase("John","[A-Z]","C","ALL")#</CFOUTPUT>
```

See also: `ReplaceList()`

ReplaceList()

Description: `ReplaceList()` replaces all occurrences of elements in one string with corresponding elements in another. Both sets of elements must be specified as comma-delimited values, and an equal number of values must exist in each set. `ReplaceList` takes three parameters. The first is the string to be processed, the second is the set of values to be replaced, and the third is the set of values with which to replace them.

Syntax:

```
ReplaceList(String, FindWhatList, ReplaceWithList)
```

Example: The following example replaces all occurrences of state names with their appropriate abbreviations:

```
#ReplaceList(address, "CA, IN, MI", "California, Indiana, Michigan")#
```

Tip

The `ReplaceList()` function is case sensitive, and there is no non–case-sensitive equivalent function. To perform a non–case-sensitive replacement, you first must convert both the search and target strings to either upper- or lowercase (using the `UCase()` or `LCase()` function).

Note

Unlike other replacement functions, the `ReplaceList()` function takes no scope parameter. `ReplaceList()` replaces all occurrences of matching elements.

See also: `Replace()`

`Reverse()`

Description: `Reverse()` reverses the characters in a string. `Reverse()` takes a single parameter, the string to be reversed.

Syntax:

```
Reverse(String)
```

Example: The following example reverses the contents of a user-supplied field:

```
#Reverse(sequence_id)#
```

`Right()`

Description: `Right()`returns the specified rightmost characters from the end of a string. `Right()` takes two parameters: the string from which to extract the characters and the number of characters to extract.

Syntax:

```
Right(String, Count)
```

Example: The following example returns the last seven characters of a phone number column:

```
#Right(phone_number, 7)#
```

Tip

`Right()` does not trim trailing spaces before extracting the specific characters. To ignore whitespace when using `Right()`, you should nest the `RTrim()` within `Right()`, as in `#Right(RTrim(String), Count)#`.

See also: `Find()`, `Left()`, `Mid()`, `RemoveChars()`

RJustify()

Description: RJustify() right-aligns a string within a field of a specified length. It does this by padding spaces before the specified text. RJustify() takes two parameters—the string to process and the desired string length.

Syntax:

```
RJustify(string, length)
```

Example: The following example right-justifies the contents of a field named Zip so that it is right-aligned within a 10-character–wide field:

```
#RJustify(Zip, 10)#
```

See also: CJustify(), LJustify(), LTrim(), RTrim(), Trim()

RTrim()

Description: RTrim()trims whitespace (spaces, tabs, and newline characters) from the end of a string. RTrim() takes a single parameter, the string to be trimmed.

Syntax:

```
RTrim(String)
```

Example: The following example trims spaces from the end of a user-supplied field:

```
#RTrim(first_name)#
```

See also: CJustify(), LJustify(), LTrim(), RJustify(), Trim(), StripCR()

SpanExcluding()

Description: SpanExcluding() extracts characters from the beginning of a string until a character that is part of a specified set is reached. SpanExcluding() takes two parameters, the string to process and a comma-delimited set of values to compare against.

Syntax:

```
SpanExcluding(String,  Set)
```

Example: The following example extracts the first word of a sentence by specifying a space as the character to compare against:

```
#SpanExcluding(sentence, " ")#
```

Tip

The SpanExcluding() function is case sensitive, and there is no non–case-sensitive equivalent function. To perform a non–case-sensitive extraction, you first must convert both the search and target strings to either upper- or lowercase (using the UCase() or LCase() function).

See also: SpanIncluding()

SpanIncluding()

Description: SpanIncluding() extracts characters from the beginning of a string only as long as they match characters in a specified set. SpanIncluding() takes two parameters: the string to process and a comma-delimited set of values to compare against.

Syntax:

```
SpanIncluding(String,  Set)
```

Example: The following example extracts the house number from a street address by specifying a set of values that are digits only:

```
#SpanIncluding(address, "1234567890")#
```

Tip

The SpanIncluding() function is case sensitive, and there is no non–case-sensitive equivalent function. To perform a non–case-sensitive extraction, you first must convert both the search and target strings to either upper- or lowercase (using the UCase() or LCase() function).

See also: SpanExcluding()

StripCR()

Description: StripCR() removes all carriage return characters from a string. StripCR() takes a single parameter, the string to process.

Syntax:

```
StripCR(String)
```

Example: The following example removes carriage returns for a field to be displayed in a preformatted text block:

```
<PRE>#StripCR(comments)#</PRE>
```

Tip

The StripCR() function is particularly useful when displaying a string within HTML preformatted text tags (<PRE> and </PRE>) where carriage returns are not ignored.

See also: CJustify(), LJustify(), LTrim(), RJustify(), RTrim(), Trim()

ToBase64(), ToBinary()

Description: The ToBase64() function returns the Base64 representation of a specified string or binary object. Base64 format converts a string or an object to printable characters that can then be sent as text in e-mail or stored in a database. The ToBinary() function either converts a Base64 encoded string to a binary object or tests an existing binary value to ensure that it is correct. The ToBase64() function takes a single parameter, the string or

binary value to be encoded. The `ToBinary()` function also takes a single parameter, the Base64 encoded string to be converted or the binary object to be tested.

Syntax:

```
ToBase64(string or binary_value)
ToBinary(encoded_string or binary_value)
```

Example: The following example converts a simple text string to Base64-encoded format:

```
#ToBase64("Laura is a pretty girl")#
```

Example: The following example reads the contents of an image file into memory, converts the image file to a Base64-encoded string, and then converts the file from a Base64-encoded string back to a binary value:

```
<CFFILE ACTION="READ"
        FILE="FULL PATH OF THE IMAGE FILE"
        VARIABLE="ImgFile">
<CFSET EncodedImage = #ToBase64(ImgFile)#>
<CFSET BinaryImage = #ToBinary(EncodedImage)#>
```

Tip

If you receive data in Base64-encoded format, the `ToBinary` function is useful for re-creating, out of the Base64-encoded string, the original binary object (.gif, .jpg, and so on).

Note

When trying to convert a Base64-encoded string back to its original state, you first must convert this string to a binary object. After you've done this, you can use the `ToString()` function to convert the resulting binary object back into a string.

See also: `IsBinary()`, `ToString()`

`Trim()`

Description: `Trim()` trims whitespace (spaces, tabs, and newline characters) from both the beginning and the end of a string. `Trim()` takes a single parameter, the string to be trimmed.

Syntax:

```
Trim(String)
```

Example: The following example trims spaces from both the beginning and the end of a user-supplied field:

```
#Trim(notes)#
```

See also: `CJustify()`, `LJustify()`, `LTrim()`, `RJustify()`, `RTrim()`, `StripCR()`

UCase()

Description: UCase() converts a string to uppercase. UCase() takes a single parameter, the string to be converted, and returns the converted string.

Syntax:

```
UCase(String)
```

Example: The following example converts the contents of a table column called States to uppercase:

```
#UCase(State)#
```

See also: LCase()

Val()

Description: Val() converts the beginning of a string to a number. Val() takes a single parameter, the string to process. Conversion is possible only if the string begins with numeric characters. If conversion is impossible, 0 is returned.

Syntax:

```
Val(String)
```

Example: The following example extracts the hour portion from a time field:

```
Hour: #Val(time)#
```

Tip

Val() converts characters to numbers using a base of 10 only. To convert the string to numbers with a base other than 10, use the InputBaseN() function.

See also: Asc(), Chr(), InputBaseN(), IsNumeric()

Date AND Time FUNCTIONS

The ColdFusion Date and Time functions enable you to perform date and time manipulation on table columns and user-supplied fields.

Many of these functions work with date/time objects. A *date/time object* is a ColdFusion internal representation of a complete date and time with accuracy to the second. These objects are designed to facilitate the passing of date/time information between various ColdFusion functions and are not designed to be displayed as is. If you need to display a date/time object, you must use one of the date/time formatting functions.

Note

ColdFusion date/time objects are not the same as ODBC date/time fields. Use the CreateODBCDateTime() function to convert ColdFusion date/time objects to the ODBC format.

Many ColdFusion date and time functions take date and time values as parameters. These parameters must be valid and within a set range; otherwise, a ColdFusion syntax error is generated. The range of values allowed for each date and time field is listed in Table B.1.

Table B.1 Valid ColdFusion Date and Time Values

Field	Min	Max
Year	0	9999
Month	1	12
Day	1	31
Hour	0	23
Minute	0	59
Second	0	59

Note

Year values of less than 100 are treated as twentieth century values, and 1900 is added automatically to them.

Several of the ColdFusion date and time functions enable you to work with parts of the complete date/time object—to add days or weeks to a date, or to find out how many weeks apart two dates are, for example. These functions require you to pass a date/time part specifier that is passed as a string. (They must have quotation marks around them.) The complete list of specifiers is explained in Table B.2.

Table B.2 ColdFusion Date/Time Specifiers

Specifier	Description
D	Day
H	Hour
M	Month
N	Minute
Q	Quarter
S	Second
W	Weekday (day of week)
WW	Week
Y	Day of year
YYYY	Year

CreateDate()

Description: The CreateDate() function returns a ColdFusion date/time object that can be used with other date-manipulation or formatting functions. CreateDate() takes three parameters: the date's year, month, and day.

Syntax:

```
CreateDate(Year, Month, Day)
```

Example: The following example creates a date/time object based on three user-supplied fields:

```
#CreateDate(birth_year, birth_month, birth_day)#
```

Tip

When specifying the year using the CreateDate() function, be aware that ColdFusion will interpret the numeric values 0–29 as twenty-first century years. The numeric values 30–99 are interpreted as twentieth century values.

Note

Because the CreateDate() function takes no time values as parameters, the time portion of the created date/time object is set to all 0s.

See also: CreateDateTime(), CreateODBCDate(),CreateTime()

CreateDateTime()

Description: The CreateDateTime() function returns a ColdFusion date/time object that can be used with other date and timemanipulation or formatting functions. CreateDateTime() takes six parameters: the date's year, month, and day, and the time's hour, minute, and second.

Syntax:

```
CreateDateTime(Year, Month, Day, Hour, Minute, Second)
```

Example: The following example creates a date/time object for midnight on New Year's Day, 1997:

```
#CreateDateTime(1997, 1, 1, 0, 0, 0)#
```

Tip

When specifying the year using the CreateDateTime() function, be aware that ColdFusion will interpret the numeric values 0–29 as twenty-first century years. The numeric values 30–99 are interpreted as twentieth century values.

See also: CreateDate(), CreateODBCDateTime(), CreateTime(), ParseDateTime()

CreateODBCDate()

Description: The CreateODBCDate() function returns a date in an ODBC date/time field that can safely be used in SQL statements. CreateODBCDate() takes a single parameter, a ColdFusion date/time object.

Syntax:

```
CreateODBCDate(Date)
```

Example: The following example creates an ODBC date/time field for the current day (retrieved with the Now() function):

```
#CreateODBCDate(Now())#
```

Note

CreateODBCDate() always creates an ODBC date/time field that has the time values set to 0s, even if the passed date/time object had valid time values.

Tip

When specifying the year using the CreateODBCDate() function, be aware that ColdFusion will interpret the numeric values 0–29 as twenty-first century years. The numeric values 30–99 are interpreted as twentieth century values.

Tip

CreateODBCDate() takes a date/time object as a parameter. If you want to pass individual date values as parameters, use CreateDate() as the function parameter and pass it the values.

See also: CreateDate(), CreateODBCDateTime(), CreateODBCTime()

CreateODBCDateTime()

Description: The CreateODBCDateTime() function returns an ODBC date/time field that safely can be used in SQL statements. CreateODBCDateTime() takes a single parameter, a ColdFusion date/time object.

Syntax:

```
CreateODBCDate(Date)
```

Example: The following example creates an ODBC date/time field for the current day (retrieved with the Now() function):

```
#CreateODBCDateTime(Now())#
```

Tip

When specifying the year using the CreateODBCDateTime() function, be aware that ColdFusion will interpret the numeric values 0–29 as twenty-first century years. The numeric values 30–99 are interpreted as twentieth century values.

Tip

`CreateODBCDateTime()` takes a date/time object as a parameter. If you want to pass individual date and time values as parameters, use `CreateDateTime()` as the function parameter and pass it the values.

See also: `CreateDate()`, `CreateODBCDate()`, `CreateODBCTime()`

CreateODBCTime()

Description: The `CreateODBCTime()` function returns a time in an ODBC date/time field that can safely be used in SQL statements. `CreateODBCTime()` takes a single parameter, a ColdFusion date/time object.

Syntax:

```
CreateODBCTime(Date)
```

Example: The following example creates an ODBC date/time field for the current day (retrieved with the `Now()` function):

```
#CreateODBCTime(Now())#
```

Note

`CreateODBCTime()` always creates an ODBC date/time field that has the date values set to 0s, even if the passed date/time object had valid date values.

Tip

`CreateODBCTime()` takes a date/time object as a parameter. If you want to pass individual time values as parameters, use the `CreateTime()` as the function parameter and pass it the values.

Tip

Always enclose date/time values in quotes when passing them to the `CreateODBCTime()` function as strings. Without quotes, the value passed is interpreted as a numeral representation of a date/time object.

See also: `CreateODBCDate()`, `CreateODBCDateTime()`, `CreateTime()`

CreateTime()

Description: The `CreateTime()` function returns time in a ColdFusion date/time object that can be used with other time-manipulation or formatting functions. `CreateTime()` takes three parameters: the time's hour, minute, and second.

Syntax:

```
CreateTime(Hour, Minute, Second)
```

Example: The following example creates a date/time object based on three ColdFusion fields:

```
#CreateTime(act_hr, act_mn, act_se)#
```

Note

Because the `CreateTime()` function takes no date values as parameters, the date portion of the created date/time object is set to all `0`s.

See also: `CreateDate()`, `CreateDateTime()`, `CreateODBCTime()`

`CreateTimeSpan()`

Description: `CreateTimeSpan()` creates a date/time object that can be used to rapidly perform date and time-based calculations. `CreateTimeSpan()` takes four parameters: days, hours, minutes, and seconds. Any of these values can be set to `0` if not needed.

Syntax:

```
CreateTimeSpan(Days, Hours, Minutes, Seconds)
```

Example: The following example creates a date/time object with a time exactly six hours from now:

```
<CFSET #detonation# = #Now()# + #CreateTimeSpan(0, 6, 0, 0)#>
```

Tip

The `CreateTimeSpan()` function is designed to speed the process of performing date- and time-based calculations. Creating a date/time object with 30 days—and using standard addition operators to add this to an existing date/time object—is quicker than using the `DateAdd()` function.

Tip

The `CreateTimeSpan()` function can be used in conjunction with the `CACHEDWITHIN` attribute of the `CFQUERY` tag. You can specify a period of time, using `CreateTimeSpan()`, in which the results of a query will be cached. This use of the `CreateTimeSpan()` function is effective only if you have enabled query caching in the ColdFusion Administrator.

See also: `DateAdd()`

`DateAdd()`

Description: `DateAdd()` is used to add or subtract values to a date/time object—you can add a week or subtract a year, for example. `DateAdd()` takes three parameters. The first is the date specifier (refer to Table B.2), the second is the number of units to add or subtract, and the third is the date/time object to be processed. `DateAdd()` returns a modified date/time object.

Syntax:

```
DateAdd( Specifier, Units, Date)
```

Example: The following example returns tomorrow's date (it adds one day to today's date):

```
#DateAdd('D', 1, Now())#
```

The next example returns a date exactly 10 weeks earlier than the date in a table column:

```
#DateAdd('WW', -10, Now())#
```

Tip

To subtract values from a date/time object, use the `DateAdd()` function and pass a negative number of units. For example, `-5` subtracts five units of whatever specifier was passed.

Tip

You can add or subtract the modified date/time object that `DateAdd()` returns from other date/time objects and use this value in the `CACHEDWITHIN` attribute of the `<CFQUERY>` tag.

DateCompare()

Description: `DateCompare()` enables you to compare two dates to determine whether they are the same or whether one is greater than the other. `DateCompare()` takes two parameters—the dates to compare, which can be specified as date/time objects or string representations of dates. `DateCompare()` returns `-1` if the first date is less than the second date, `0` if they are the same, and `1` if the first date is greater than the second date.

Syntax:

```
DateCompare(Date1, Date2)
```

Example: The following example verifies that a user-supplied order ship date is valid (not already passed):

```
<CFIF #DateCompare(ship_date, Now())# IS -1>
 We can't ship orders yesterday!
</CFIF>
```

See also: `DateDiff()`, `DatePart()`

DateConvert()

Description: The `DateConvert()` function converts local machine time to UTC (Universal Coordinated Time) time or vice versa. The `DateConvert()` function takes two parameters. The first is the conversion type, and the second is the date/time value to be converted. Valid conversion types for the `DateConvert()` function are `"local2utc"` and `"utc2local"`. The date/time value to be converted can be constructed using any ColdFusion date/time string. If a calculation for daylight savings time is required, the function uses the settings for the machine that is executing the code.

Syntax:

```
DateConvert(conversion_type, date)
```

Example: The following example converts the local machine date/time to UTC:

```
<CFSET TheTime = #CreateDateTime(2006, 4, 26, 12, 41, 32)#>
#DateConvert("local2utc", TheTime)#
```

See also: CreateDateTime(), DatePart()

DateDiff()

Description: DateDiff() returns the number of units of a passed specifier by which one date is greater than a second date. Unlike DateCompare(), which returns the greater date, DateDiff() tells you how many days, weeks, or months it is greater by. DateDiff() takes three parameters: The first is the date specifier (refer to Table B.2), and the second and third are the dates to compare.

Syntax:

```
DateDiff(Specifier, Date1, Date2)
```

Example: The following example returns how many weeks are left in this century, by specifying today's date (using the Now() function) and the first date of the next century (using the CreateDate() function) as the two dates to compare:

```
There are #DateDiff("WW", Now(), CreateDate(2001, 1, 1))
➥# weeks left in this year!
```

Note

If the first date passed to DateDiff() is greater than the second date, a negative value is returned. Otherwise, a positive value is returned.

Tip

You can add or subtract the modified date/time object that DateDiff() returns from other date/time objects and use this value in the CACHEDWITHIN attribute of the <CFQUERY> tag.

See also: DateCompare(), DatePart()

DatePart()

Description: DatePart() returns the specified part of a passed date. DatePart() takes two parameters: The first is the date specifier (refer to Table B.2), and the second is the date/time object to process.

Syntax:

```
DatePart(Specifier, Date)
```

Example: The following example returns the day of week on which a user was born (and converts it to a string date using the `DayOfWeekAsString()` function):

```
You were born on a #DayOfWeekAsString(DatePart('W', dob))#
```

See also: `DateCompare()`, `DateDiff()`, `Day()`, `DayOfWeek()`, `DayOfYear()`, `Hour()`, `Minute()`, `Month()`, `Quarter()`, `Second()`, `Week()`, `Year()`

Day()

Description: `Day()` returns a date/time object's day of month as a numeric value with possible values of `1–31`. `Day()` takes a single parameter, the date/time object to be processed.

Syntax:

```
Day(Date)
```

Example: The following example returns today's day of month:

```
Today is day #Day(Now())# of this month
```

Tip

When specifying the year using the `Day()` function, be aware that ColdFusion will interpret the numeric values `0–29` as twenty-first century years. The numeric values `30–99` are interpreted as twentieth century values.

See also: `DayOfWeek()`, `DayOfYear()`, `Hour()`, `Minute()`, `Month()`, `Quarter()`, `Second()`, `Week()`, `Year()`

DayOfWeek()

Description: `DayOfWeek()` returns a date/time object's day of week as a numeric value with possible values of `1–7`. `DayOfWeek()` takes a single parameter, the date/time object to be processed.

Syntax:

```
DayOfWeek(Date)
```

Tip

When specifying the year using the `DayOfWeek()` function, be aware that ColdFusion will interpret the numeric values `0–29` as twenty-first century years. The numeric values `30–99` are interpreted as twentieth century values.

Tip

Always enclose date/time values in quotes when passing them to the `DayOfWeek()` function as strings. Without quotes, the value passed is interpreted as a numeral representation of a date/time object.

Example: The following example returns today's day of week:

```
Today is day #DayOfWeek(Now())# of this week
```

See also: Day(), DayOfYear(), Hour(), Minute(), Month(), Quarter(), Second(), Week(), Year()

DayOfWeekAsString()

Description: DayOfWeekAsString() returns the English weekday name for a passed day of week number. DayOfWeekAsString() takes a single parameter, the day of week to process, with a value of 1–7.

Syntax:

```
DayOfWeekAsString(DayNumber)
```

Example: The following example returns today's day of week:

```
Today is day #DayOfWeekAsString(DayOfWeek(Now()))# of this week
```

Tip

When specifying the year using the DayOfWeekAsString() function, be aware that ColdFusion will interpret the numeric values 0–29 as twenty-first century years. The numeric values 30–99 are interpreted as twentieth century values.

See also: DayOfWeek(), MonthAsString()

DayOfYear()

Description: DayOfYear() returns a date/time object's day of year as a numeric value, taking into account leap years. DayOfYear() takes a single parameter, the date/time object to be processed.

Syntax:

```
DayOfYear(Date)
```

Example: The following example returns today's day of year:

```
Today is day #DayOfYear(Now())# of year #Year(Now())#
```

Tip

When specifying the year using the DayOfYear() function, be aware that ColdFusion will interpret the numeric values 0–29 as twenty-first century years. The numeric values 30–99 are interpreted as twentieth century values.

Tip

Always enclose date/time values in quotes when passing them to the DayOfYear() function as strings. Without quotes, the value passed is interpreted as a numeral representation of a date/time object.

See also: Day(), DayOfWeek(), Hour(), Minute(), Month(), Quarter(), Second(), Week(), Year()

DaysInMonth()

Description: DaysInMonth() returns the number of days in a specified month, taking into account leap years. DaysInMonth() takes a single parameter, the date/time object to evaluate.

Syntax:

```
DaysInMonth(Date)
```

Example: The following example returns the number of days in the current month:

```
This month has #DaysInMonth(Now())# days
```

Tip

DaysInMonth()takes a date/time object as a parameter, and there is no equivalent function that takes a year and month as its parameters. Fortunately, this easily can be accomplished by combining the DaysInMonth() and CreateDate() functions. For example, to determine how many days are in February 2000, you can create a statement that looks like this: #DaysInMonth(CreateDate(2000, 2, 1))#.

Tip

When specifying the year using the DaysInMonth() function, be aware that ColdFusion will interpret the numeric values 0–29 as twenty-first century years. The numeric values 30–99 are interpreted as twentieth century values.

Tip

Always enclose date/time values in quotes when passing them to the DaysInMonth() function as strings. Without quotes, the value passed is interpreted as a numeral representation of a date/time object.

See also: DaysInYear(), FirstDayOfMonth()

DaysInYear()

Description: DaysInYear() returns the number of days in a specified year, taking into account leap years. DaysInYear() takes a single parameter, the date/time object to evaluate.

Syntax:

```
DaysInYear(Date)
```

Example: The following example returns the number of days in the current year:

```
This year, #Year(Now())#, has #DaysInYear(Now())# days
```

Tip

DaysInYear() takes a date/time object as a parameter, and there is no equivalent function that takes just a year as its parameter. Fortunately, this easily can be accomplished by combining the DaysInYear() and CreateDate() functions. For example, you can create a statement that looks like this to determine how many days are in the year 2000: #DaysInYear(CreateDate(2000, 1, 1))#.

Tip

When specifying the year using the `DaysInYear()` function, be aware that ColdFusion will interpret the numeric values `0–29` as twenty-first century years. The numeric values `30–99` are interpreted as twentieth century values.

Tip

Always enclose date/time values in quotes when passing them to the `DaysInYear()` function as strings. Without quotes, the value passed is interpreted as a numeral representation of a date/time object.

See also: `DaysInMonth()`, `FirstDayOfMonth()`

`FirstDayOfMonth()`

Description: `FirstDayOfMonth()` returns the day of the year on which the specified month starts. `FirstDayOfMonth()` takes a single parameter, the date/time object to evaluate.

Syntax:

```
FirstDayOfMonth(Date)
```

Example: The following example returns the day of the year on which the current month starts:

```
#FirstDayOfMonth(Now())#
```

Tip

`FirstDayOfMonth()` takes a date/time object as a parameter, and no equivalent function exists that takes just a month and year as its parameters. Fortunately, this easily can be accomplished by combining the `FirstDayOfMonth()` and `CreateDate()` functions. For example, to determine the day of year on which March 1999 starts, you can create a statement that looks like this: `#FirstDayOfMonth(CreateDate(1999, 3, 1))#`.

Tip

When specifying the year using the `FirstDayOfMonth()` function, be aware that ColdFusion will interpret the numeric values `0–29` as twenty-first century years. The numeric values `30–99` are interpreted as twentieth century values.

Tip

Always enclose date/time values in quotes when passing them to the `FirstDayOfMonth()` function as strings. Without quotes, the value passed is interpreted as a numeral representation of a date/time object.

See also: `DaysInMonth()`, `DaysInYear()`

GetHttpTimeString()

Description: This function formats a ColdFusion date/time object according to the HTTP standard outlined in RFC 1123. This function takes a single parameter, the date/time object to be formatted.

Syntax:

```
GetHttpTimeString(date_time)
```

Example: The following example converts the current system time to a date/time format compatible with RFC 1123:

```
#GetHttpTimeString(#CreateODBCDateTime(Now())#)#
```

Tip

The date/time value that is returned by the `GetHttpString()` function always is formatted as GMT (Greenwich Mean Time) to remain consistent with RFC 1123.

GetTimeZoneInfo()

Description: The `GetTimeZoneInfo()` function returns a structure containing relevant time zone information from the machine on which the code is run. The `GetTimeZoneInfo()` function does not take any parameters, and the structure returned contains four elements with the keys outlined in Table B.3.

TABLE B.3 `GetTimeZoneInfo`—KEYS RETURNED IN STRUCTURE

Key	Description
`utcTotalOffset`	The difference in local time (time captured from the machine executing the code) from UTC (Universal Coordinated Time). A + sign lets you know that the time zone you are comparing is west of UTC; a –sign lets you know that the time zone you are comparing is east of UTC.
`utcHourOffset`	The difference in local time from UTC in hours.
`utcMinuteOffset`	The difference in local time from UTC in minutes, after the hours offset has been figured in. For some countries, such as those in North America, the minute offset is 0 because these countries are offset from UTC by exactly *n* hours. However, some countries in the world are offset from UTC by *n* hours and *n* minutes.
`isDSTOn`	Returns `TRUE` if daylight savings time is on in the machine executing the code; otherwise, it returns `FALSE` if daylight savings time is off.

Syntax:

```
GetTimeZoneInfo()
```

Example: The following example uses `GetTimeZoneInfo()` to compare UTC offsets to the local time:

```
<CFSET myTime = GetTimeZoneInfo()>
<CFOUTPUT>
The difference in local time from UTC in seconds is #myTime.utcTotalOffset#.<BR>
The difference in local time from UTC in hours is #myTime.utcHourOffset#.<BR>
The difference in local time from UTC in minutes is #mytime.utcMinuteOffset#.<BR>
</CFOUTPUT>
```

See also: `DateConvert()`, `CreateDateTime()`, `DatePart()`

Hour()

Description: `Hour()` returns a date/time object's hour as a numeric value with possible values of `0`–`23`. `Hour()` takes a single parameter, the date/time object to be processed.

Syntax:

```
Hour(Date)
```

Example: The following example returns the current hour of day:

```
This is hour #Hour(Now())# of the day
```

Tip

When specifying the year using the `Hour()` function, be aware that ColdFusion will interpret the numeric values `0`–`29` as twenty-first century years. The numeric values `30`–`99` are interpreted as twentieth century values.

Tip

Always enclose date/time values in quotes when passing them to the `Hour()` function as strings. Without quotes, the value passed is interpreted as a numeral representation of a date/time object.

See also: `Day()`, `DayOfWeek()`, `DayOfYear()`, `Minute()`, `Month()`, `Quarter()`, `Second()`, `Week()`, `Year()`

IsDate

Description: `IsDate()` checks whether a string contains a valid date; it returns `TRUE` if it does and `FALSE` if it does not. `IsDate()` takes a single parameter, the string to be evaluated.

Syntax:

```
IsDate(String)
```

Example: The following example checks whether a user-supplied date string contains a valid date:

```
<CFIF #IsDate(ship_date)# IS "No">
 You entered an invalid date!
</CFIF>
```

Note

`IsDate()` checks U.S.-style dates only. Use the `LSIsDate()` function for international date support.

Tip

When specifying the year using the `IsDate()` function, be aware that ColdFusion will interpret the numeric values `0–29` as twenty-first century years. The numeric values `30–99` are interpreted as twentieth century values.

See also: `IsLeapYear()`, `LSIsDate()`, `ParseDateTime()`

IsLeapYear()

Description: `IsLeapYear()` checks whether a specified year is a leap year. `IsLeapYear()` takes a single parameter, the year to check. It returns `TRUE` if it is a leap year and `FALSE` if not.

Syntax:

```
IsLeapYear(Year)
```

Example: The following example checks whether this year is a leap year:

```
<CFIF #IsLeapYear(Year(Now()))#>
 #Year(Now())# is a leap year
<CFELSE>
#Year(Now())# is not a leap year
</CFIF>
```

Tip

`IsLeapYear()` takes a year as a parameter, not a date/time object. To check whether a date stored in a date/time object is a leap year, use the `-Year()` function to extract the year and pass that as the parameter to `IsLeapYear()`.

See also: `IsDate()`

IsNumericDate()

Description: `IsNumericDate()` checks whether a value passed as a date in the ColdFusion internal date format is in fact a legitimate date. `IsNumericDate()` takes a single parameter, the date to be checked. This date is a floating point value with precision until the year 9999. `IsNumericDate()` returns `TRUE` if the passed date value is valid and `FALSE` if it is not.

Syntax:

```
IsNumericDate(Real)
```

Example: The following example checks whether a local variable contains a valid date:

```
<CFIF #IsNumericDate(var.target_date)# IS "Yes">
```

See also: `IsDate()`

Minute()

Description: Minute() returns a date/time object's hour as a numeric value with possible values of 0–59. Minute() takes a single parameter, the date/time object to be processed.

Syntax:

```
Minute(Date)
```

Example: The following example returns the current time's minutes:

```
#Minute(Now())# minutes have elapsed since #Hour(Now())# o'clock
```

Tip

When specifying the year using the Minute() function, be aware that ColdFusion will interpret the numeric values 0–29 as twenty-first century years. The numeric values 30–99 are interpreted as twentieth century values.

Tip

Always enclose date/time values in quotes when passing them to the Minute() function as strings. Without quotes, the value passed is interpreted as a numeral representation of a date/time object.

See also: Day(), DayOfWeek(), DayOfYear(), Hour(), Month(), Quarter(), Second(), Week(), Year()

Month()

Description: Month() returns a date/time object's month as a numeric value with possible values of 1–12. Month() takes a single parameter, the date/time object to be processed.

Syntax:

```
Month(Date)
```

Example: The following example returns the current month:

```
It is month #Month(Now())# of year #Year(Now())#
```

See also: Day(), DayOfWeek(), DayOfYear(), Hour(), Minute(), Quarter(), Second(), Week(), Year()

MonthAsString()

Description: MonthAsString() returns the English month name for a passed month number. MonthAsString() takes a single parameter, the number of the month to process, with a value of 1–12.

Syntax:

```
MonthAsString(MonthNumber)
```

Example: The following example returns the English name of the current month:

```
It is #MonthAsString(Month(Now()))#".
```

See also: `DayOfWeek()`, `Month()`

Now()

Description: `Now()` returns a date/time object containing the current date and time precisely to the second. `Now()` takes no parameters.

Syntax:

```
Now()
```

Example: The following example returns the current date and time formatted for correct display:

```
It is now #DateFormat(Now())# #TimeFormat(Now())#
```

Note

The `Now()` function returns the system date and time of the computer running the ColdFusion service, not of the system running the Web browser. The date/time object that the `Now()` function returns can be passed to many other date/time functions.

ParseDateTime()

Description: `ParseDateTime()` converts a date in string form into a ColdFusion date/time object. `ParseDateTime()` takes two parameters: the string to be converted and, if specified, the POP-conversion method. When a POP-conversion method (POP or STANDARD) is specified, this function parses the date/time string passed from a POP mail server.

Syntax:

```
ParseDateTime(String)# or #ParseDateTime(String [,pop-conversion method])
```

Note

When using the `ParseDateTime()` function with a POP-conversion method, one of two POP-conversion methods can be specified. The first of these methods, POP, converts the date/time string passed from a POP mail server to GMT for the U.S. locale. The second POP-conversion method available, STANDARD, provides no conversion on the date/time string passed and simply returns the time string as it was retrieved from the POP mail server.

Example: The following example converts a user-supplied string containing a date into a ColdFusion date/time object:

```
<CFSET ship_date = #ParseDateTime(FORM.ship_date)#>
```

Note

ParseDateTime() supports U.S.-style dates and times only. Use the LSParseDateTime() function for international date and time support.

See also: CreateDateTime(), LSParseDateTime()

Quarter()

Description: Quarter() returns a date/time object's quarter as a numeric value with possible values of 1–4. Quarter() takes a single parameter, the date/time object to be processed.

Syntax:

```
Quarter(Date)
```

Example: The following example returns the current quarter:

```
We are in quarter #Quarter(Now())# of year #Year(Now())#
```

Tip

When specifying the year using the Quarter() function, be aware that ColdFusion will interpret the numeric values 0–29 as twenty-first century years. The numeric values 30–99 are interpreted as twentieth century values.

Tip

Always enclose date/time values in quotes when passing them to the Quarter() function as strings. Without quotes, the value passed is interpreted as a numeral representation of a date/time object.

See also: Day(), DayOfWeek(), DayOfYear(), Hour(), Minute(), Month(), Second(), Week(), Year()

Second()

Description: Second() returns a date/time object's hour as a numeric value with possible values of 0–59. Second() takes a single parameter, the date/time object to be processed.

Syntax:

```
Second(Date)
```

Example: The following example returns the current minute's seconds:

```
We are now #Second(Now())# seconds into the current minute
```

Tip

Always enclose date/time values in quotes when passing them to the Second() function as strings. Without quotes, the value passed is interpreted as a numeral representation of a date/time object.

See also: Day(), DayOfWeek(), DayOfYear(), Hour(), Minute(), Month(), Quarter(), Week(), Year()

Week()

Description: Week() returns a date/time object's week in year as a numeric value with possible values of 1–52. Week() takes a single parameter, the date/time object to be processed.

Syntax:

```
Week(Date)
```

Example: The following example returns the current week in year:

```
This is week #Week(Now())# of year #Year(Now())#
```

Tip

When specifying the year using the Week() function, be aware that ColdFusion will interpret the numeric values 0–29 as twenty-first century years. The numeric values 30–99 are interpreted as twentieth century values.

Tip

Always enclose date/time values in quotes when passing them to the Week() function as strings. Without quotes, the value passed is interpreted as a numeral representation of a date/time object.

See also: Day(), DayOfWeek(), DayOfYear(), Hour(), Minute(), Month(), Quarter(), Second(), Year()

Year()

Description: Year() returns a date/time object's year as a numeric value with possible values of 100–9999. Year() takes a single parameter, the date/time object to be processed.

Syntax:

```
Year(Date)
```

Example: The following example returns the current year value:

```
It is year #Year(Now())#
```

See also: Day(), DayOfWeek(), DayOfYear(), Hour(), Minute(), Month(), Quarter(), Second(), Week()

DATA FORMATTING FUNCTIONS

Powerful data-manipulation functions and database-interaction capabilities are pretty useless unless you have ways to display data in a clean, readable format. ColdFusion data addresses this need by providing an array of highly capable formatting functions.

Many of these functions take optional format masks as parameters, thereby giving you an even greater level of control over the final output.

DateFormat()

Description: DateFormat() displays the date portion of a date/time object in a readable format. DateFormat() takes two parameters—the first is the date/time object to be displayed, and the second is an optional mask value enabling you to control exactly how the data is formatted. If no mask is specified, the default mask of DD-MMM-YY is used. The complete set of date masks is listed in Table B.4.

Syntax:

```
DateFormat(Date [, mask ])
```

TABLE B.4 DateFormat() MASK CHARACTERS

Mask	Description
D	Day of month in numeric form with no leading 0 for single-digit days.
DD	Day of month in numeric form with a leading 0 for single-digit days.
DDD	Day of week as a three-letter abbreviation (Sun for Sunday is an example).
DDDD	Day of week as its full English name.
M	Month in numeric form with no leading 0 for single-digit months.
MM	Month in numeric form with a leading 0 for single-digit months.
MMM	Month as a three-letter abbreviation (Jan for January is an example).
MMMM	Month as its full English name.
Y	Year as last two digits of year with no leading 0 for years less than 10.
YY	Year as last two digits of year with a leading 0 for years less than 10.
YYYY	Year as full four digits.
GG	Representative of a period or an era. This mask is currently ignored by ColdFusion but has been reserved for future use.

Example: The following example displays today's date with the default formatting options:

```
Today is: #DateFormat(Now())#
```

The next example displays the same date but uses the full names of both the day of week and the month:

```
It is #DateFormat(Now(), "DDDD, MMMM DD, YYYY")#
```

The final example displays today's date in the European format (day/month/year):

```
It is #DateFormat(Now(), "DD/MM/YY")#
```

Note

Unlike the `TimeFormat()` function mask specifiers, the `DateFormat()` function mask specifiers are not case sensitive.

Note

`DateFormat()` supports U.S.-style dates only. Use the `LSDateFormat()` function for international date support.

See also: `LSDateFormat()`, `TimeFormat()`

DecimalFormat()

Description: `DecimalFormat()` is a simplified number formatting function that outputs numbers with two decimal places, commas to separate the thousands, and a minus sign for negative values. `DecimalFormat()` takes a single parameter, the number to display. The `DecimalFormat()` function will round numbers to the nearest hundredth.

Syntax:

```
DecimalFormat(Number)
```

Example: The following example displays a table column in the decimal format:

```
Quantity: #DecimalFormat(quantity)#
```

Tip

For more precise numeric display, use the `NumberFormat()` function instead.

See also: `NumberFormat()`

DollarFormat()

Description: `DollarFormat()` is a simplified U.S. currency formatting function that outputs numbers with a dollar sign at the front, two decimal places, commas to separate the thousands, and a minus sign for negative values. `DollarFormat()` takes a single parameter, the number to display.

Syntax:

```
DollarFormat(Number)
```

Example: The following example displays the results of an equation (quantity multiplied by item cost) in the dollar format:

```
Total cost: #DollarFormat(quantity*item_cost)#
```

Tip

For more precise currency display, use the `NumberFormat()` function instead.

Note

`DollarFormat()` supports U.S. dollars only. Use the `LSCurrencyFormat()` function for international currency support.

See also: `LSCurrencyFormat()`, `NumberFormat()`

FormatBaseN()

Description: `FormatBaseN()` converts a number to a string using the base specified. Valid radix values are `2–36`.

Syntax:

```
FormatBaseN(Number, Radix)
```

Example: The following example converts a user-supplied number into hexadecimal notation:

```
#FormatBaseN(Number, 16)#
```

To convert a number to its binary format, you can do the following:

```
#FormatBaseN(Number, 2)#
```

See also: `InputBaseN()`

HTMLCodeFormat()

Description: `HTMLCodeFormat()` displays text with HTML codes with a preformatted HTML block (using the `<PRE>` and `</PRE>` tags). `HTMLCodeFormat()` takes a single parameter, the text to be processed. When you evaluate a string with the `HTMLCodeFormat()` function, all carriage returns are removed from the string and all special characters contained within the string are escaped.

Syntax:

```
HTMLCodeFormat(Text)
```

Example: The following example uses preformatted text to display the code used to generate a dynamic Web page:

```
#HTMLCodeFormat(page)#
```

Tip

`HTMLCodeFormat()` is useful for displaying data in `FORM TEXTAREA` fields.

See also: `HTMLEditFormat()`, `ParagraphFormat()`

HTMLEditFormat()

Description: `HTMLEditFormat()` converts supplied text into a safe format, with any HTML control characters converted to their appropriate entity codes. `HTMLEditFormat()` takes a single parameter, the text to convert.

Syntax:

```
HTMLEditFormat(Text)
```

Example: The following example displays the HTML code used to render a dynamic Web page inside a bordered box:

```
<TABLE BORDER>
 <TR>
  <TD>#HTMLEditFormat(page)#</TD>
 </TR>
</TABLE>
```

Tip

Use `HTMLEditFormat()` to display HTML code and tags within your page.

See also: `HTMLCodeFormat()`, `ParagraphFormat()`

NumberFormat()

Description: `NumberFormat()` enables you to display numeric values in a readable format. `NumberFormat()` takes two parameters: the number to be displayed and an optional mask value. If the mask is not specified, the default mask of `",99999999999999"` is used. The complete set of number masks is listed in Table B.5.

Syntax:

```
NumberFormat(Number [, mask ])
```

TABLE B.5 `NumberFormat()` MASK CHARACTERS

Mask	Description
_	Optional digit placeholder
9	Optional digit placeholder (same as _ but shows decimal place more clearly)
.	Location of decimal point
0	Forces padding with 0s
()	Displays parentheses around the number if it is less than 0.
+	Displays a plus sign in front of positive numbers and a minus sign in front of negative numbers
-	Displays a minus sign in front of negative numbers and leaves a space in front of positive numbers
,	Separates thousands with commas
C	Centers number within mask width
L	Left-justifies number within mask width
$	Places a dollar sign in front of the number
^	Specifies the exact location for separating left and right formatting

Example: To demonstrate how the number masks can be used, Table B.6 lists examples of various masks being used to format the numbers 1453.876 and –1453.876.

TABLE B.6 NUMBER FORMATTING EXAMPLES

Mask	Result	Notes
`NumberFormat(1453.876, "9999")`	1454	No decimal point was specified in the mask, so the number is rounded to the nearest integervalue.
`NumberFormat(-1453.876, "9999")` `the number is rounded to`	–1454	No decimal point was specified in the mask, so the nearest integer value.
`NumberFormat(1453.876, "9999.99")`	1453.88	Even though a decimal point is provided, the number of decimal places specified is less than needed; the decimal portion must be rounded to the nearest integer value.
`NumberFormat(1453.876, "(9999.99)")`	1453.88	The number is a positive number, so the parentheses are ignored.
`NumberFormat(-1453.876, "(9999.99)")`	(1453.88)	The number is a negative number, so parentheses are displayed around the number.
`NumberFormat(1453.876, "-9999.99")`	1453.88	The number is a positive number, so the minus sign is ignored.
`NumberFormat(-1453.876, "-9999.99")`	-1453.88	The number is a negative number, so the minus sign is displayed.
`NumberFormat(1453.876, "+9999.99")`	+1453.88	The number is a positive number, so a plus sign is displayed.
`NumberFormat(-1453.876, "+9999.99")`	-1453.88	The number is a negative number, so a minus sign is displayed.
`NumberFormat(1453.876, "$9999.99")`	$1453.88	Using the dollar sign as the first character of the mask will place a dollar sign at the beginning of the output.

TABLE B.6 CONTINUED

Mask	Result	Notes
`NumberFormat(1453.876, "C99999^9999")`	1453.876	Position six of the mask is a carat character, so the decimal point is positioned there even though less than six digits appear before the decimal point. This enables you to align columns of numbers at the decimal point.

Note

Use the `LSNumberFormat()` function for international number support.

See also: `DecimalFormat()`, `DollarFormat()`, `LSNumberFormat()`

`TimeFormat()`

Description: `TimeFormat()` displays the time portion of a date/time object in a readable format. `TimeFormat()` takes two parameters. The first is the date/time object to be displayed, and the second is an optional mask value enabling you to control exactly how the data is formatted. If no mask is specified, the default mask of `hh:mm:tt` is used. The complete set of date masks is listed in Table B.7.

Syntax:

```
TimeFormat(Date [, mask ])
```

TABLE B.7 `TimeFormat()` MASK CHARACTERS

Mask	Description
`h`	Hours in 12-hour clock format with no leading 0 for single-digit hours
`hh`	Hours in 12-hour clock format with a leading 0 for single-digit hours
`H`	Hours in 24-hour clock format with no leading 0 for single-digit hours
`HH`	Hours in 24-hour clock format with a leading 0 for single-digit hours
`m`	Minutes with no leading 0 for single-digit minutes
`mm`	Minutes with a leading 0 for single-digit minutes
`s`	Seconds with no leading 0 for single-digit seconds
`ss`	Seconds with a leading 0 for single-digit seconds
`t`	Single character meridian specifier, either `A` or `P`
`tt`	Two-character meridian specifier, either `AM` or `PM`

Example: The following example displays the current time with the default formatting options:

```
The time is: #TimeFormat(Now())#
```

The next example displays the current time with seconds in 24-hour clock format:

```
The time is: #TimeFormat(Now(), "HH:mm:ss")#
```

Note

Unlike the `DateFormat()` function mask specifiers, the `TimeFormat()` function mask specifiers are case sensitive on the hour mask.

Note

`TimeFormat()` supports U.S.-style times only. Use the `LSTimeFormat()` function for international time support.

See also: `DateFormat()`, `LSTimeFormat()`

ParagraphFormat()

Description: `ParagraphFormat()` converts text with embedded carriage returns for correct HTML display. HTML ignores carriage returns in text, so they must be converted to HTML paragraph markers (the `<P>` tag) to be displayed correctly. `ParagraphFormat()` takes a single parameter, the text to be processed.

Syntax:

```
ParagraphFormat(Text)
```

Example: The following example displays a converted text file inside a `FORM TEXTAREA` field:

```
<TEXTAREA NAME="comments">#ParagraphFormat(comments)#</TEXTAREA>
```

Tip

`ParagraphFormat()` is useful for displaying data in `FORM TEXTAREA` fields.

See also: `HTMLCodeFormat()`, `HTMLEditFormat()`

YesNoFormat()

Description: `YesNoFormat()` converts `TRUE` and `FALSE` values to `Yes` and `No`. `YesNoFormat()` takes a single parameter—the number, string, or expression to evaluate. When evaluating numbers, `YesNoFormat()` treats `0` as `NO` and any nonzero value as `YES`.

Syntax:

```
YesNoFormat(Value)
```

Example: The following example converts a table Boolean value to a `Yes` or `No` string:

```
Member: #YesNoFormat(member)#
```

See also: `IsBoolean()`

MATHEMATICAL FUNCTIONS

To assist you in performing calculations, ColdFusion comes with a complete suite of mathematical functions, random-number–generation functions, and arithmetic expressions. As with all ColdFusion functions, these mathematical functions can be nested.

Some of the mathematical functions take one or more numeric values as parameters. You can pass real values, integer values, and ColdFusion fields to these functions.

Table B.8 lists the complete set of ColdFusion mathematical functions. Table B.9 lists the supported arithmetic expressions.

TABLE B.8 COLDFUSION MATHEMATICAL FUNCTIONS

Function	Parameters	Returns
`Abs()`	`(number)`	Absolute value of passed number.
`Acos()`	`(number)`	The arc cosine of a passed number.
`Asin()`	`(number)`	The arc sine of a passed number in radians.
`Atn()`	`(number)`	Arc tangent of passed number.
`Ceiling()`	`(number)`	The closest integer greater than passed number.
`Cos()`	`(number)`	Cosine of passed number.
`DecrementValue()`	`(number)`	Number decremented by 1.
`Exp()`	`(number)`	E to the power of passed number.
`Fix()`	`(number)`	If passed number is greater than or equal to `0`, it returns the closest integer smaller than the passed number. If not, it returns the closest integer greater than the passed number.
`IncrementValue()`	`(number)`	Number incremented by 1.
`Int()`	`(number)`	The closest integer smaller than passed number.
`Log()`	`(number)`	Natural logarithm of passed number.
`Log10()`	`(number)`	Base 10 log of passed number.
`Max()`	`(number1, number2)`	The greater of the two passed numbers.
`Min()`	`(number1, number2)`	The smaller of the two passed numbers.

Function	Parameters	Returns
`Pi()`		Value of pi as 3.14159265359.
`Rand()`		A random number between 0 and 1.
`Randomize()`	`(number)`	Seeds the random number generator with the passed number.
`RandRange()`	`(number1, number2)`	A random integer value between the two passed numbers.
`Round()`	`(number)`	The integer closest (either greater or smaller) to the passed number.
`Sgn()`	`(number)`	Sign—either –1, 0, or 1, depending on whether the passed number is negative, 0, or positive.
`Sin()`	`(number)`	Sine of passed number.
`Sqr()`	`(number)`	Square root of passed number.
`Tan()`	`(number)`	Tangent of passed number.

TABLE B.9 COLDFUSION ARITHMETIC EXPRESSIONS

Expression	Description
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`MOD`	Modular (finds remainder)
`\`	Integer division (both values must be integers)
`^`	Power

Example: This first example returns the natural logarithm of the number `10`:

```
#Log(10)#
```

This next example returns the value of pi rounded to the nearest integer value, `3`:

```
#Round(Pi())#
```

The following example uses the `Min()` and `Max()` functions to determine the greater and smaller of two ColdFusion fields:

```
Of the two numbers #Num1# and #Num2#, Max(#Num1#, #Num2#) is the greater,
➥ and Min(#Num1#, #Num2#) is the smaller
```

To generate random numbers, you can use the `Rand()` and `RandRange()` functions. The following example generates a random number between 1 and 1,000:

```
#RandRange(1, 1000)#
```

This next example creates a variable that contains a total cost of several items:

```
<CFSET total = quantity * item_price>
```

Note

If the `Rand()` or `RandRange()` function is used prior to issuing a `Randomize()` statement, the random number generator is seeded with a random value.

INTERNATIONAL FUNCTIONS

ColdFusion fully supports the display, formatting, and manipulation of international dates, times, numbers, and currencies. To use ColdFusion's international support, you must specify the locale. A *locale* is an encapsulation of the set of attributes that govern the display and formatting of international date, time, number, and currency values. The complete list of supported locales is shown in Table B.10.

TABLE B.10 COLDFUSION LOCALES

Locale
Dutch (Belgian)
Dutch (Standard)
English (Australian)
English (Canadian)
English (New Zealand)
English (UK)
English (U.S.)
French (Belgian)
French (Canadian)
French (Standard)
French (Swiss)
German (Austrian)
German (Standard)
German (Swiss)
Italian (Standard)
Italian (Swiss)
Norwegian (Bokmal)
Norwegian (Nynorsk)
Portuguese (Brazilian)
Portuguese (Standard)

Locale
Spanish (Mexican)
Spanish (Modern)
Spanish (Standard)
Swedish

You must use the `SetLocale()` function to set the locale. You can retrieve the name of the locale currently in use with the `GetLocale()` function.

To use ColdFusion's international support, you must use the LS functions listed (later in this section). These functions behave much like the standard date, time, and formatting functions, but they honor the current locale setting.

Note

The ColdFusion server variable `Server.ColdFusion.SupportedLocales` contains a comma-delimited list of the supported locales.

GetLocale()

Description: `GetLocale()` returns the name of the locale currently in use. The locale returned by the `GetLocale()` function is determined by the native operating system running ColdFusion Application Server.

Syntax:

```
GetLocale()
```

Example: The following example saves the current locale to a local variable:

```
<CFSET current_locale = #GetLocale()#>
```

See also: `SetLocale()`

LSCurrencyFormat()

Description: `LSCurrencyFormat()` displays currency information formatted for the current locale. `LSCurrencyFormat()` takes two parameters: the number to display and an optional format type. If a type is specified, its value must be `none`, `local`, or `international`. `Type` defaults to `none`.

Syntax:

```
LSCurrencyFormat(Number [, Type])
```

Example: The following example displays the results of an equation (quantity multiplied by item cost) in formatting appropriate for the French locale:

```
<CFSET previous_locale = SetLocale("French (Standard)")>
Total cost: #LSCurrencyFormat(quantity*item_cost)#
```

Tip

For more precise currency display, use the `NumberFormat()` function instead.

Note

You can use the simpler `DollarFormat()` function for U.S. currency formatting.

See also: `DollarFormat()`, `NumberFormat()`

LSDateFormat()

Description: `LSDateFormat()` displays the date portion of a date/time object in a readable format. `LSDateFormat()` is the locale-specific version of the `DateFormat()` function. Similar to `DateFormat()`, `LSDateFormat()` takes two parameters; the first is the date/time object to be displayed, and the second is an optional mask value enabling you to control exactly how the data is formatted. If no mask is specified, a format suitable for the current locale is used. The complete set of date masks is listed in Table B.4 in the description of the `DateFormat()` function.

Syntax:

```
LSDateFormat(Date [, mask ])
```

Example: The following example displays today's date with the default formatting options for the current locale:

```
Today is: #LSDateFormat(Now())#
```

The next example displays the same date but uses the current locale's full names of both the day of week and the month:

```
It is #LSDateFormat(Now(), "DDDD, MMMM DD, YYYY")#
```

Note

You can use the simpler `DateFormat()` function for U.S. dates.

See also: `DateFormat()`, `LSNumberFormat()`, `LSTimeFormat()`

LSEuroCurrencyFormat()

Description: `LSEuroCurrencyFormat()` returns a currency value formatted in the convention of the locale enabled on the executing machine, with the Euro as the currency symbol. The `LSEuroCurrencyFormat` function takes two parameters. The first, `currency-value` is the actual amount or value of the currency you want to format. The second parameter, `type`, is an optional parameter that enables you to specify a currency type of `none`, `local`, or `international`. If no currency type is specified, the default type of `local` is used. Depending on which type you choose, ColdFusion will display the value with differing currency symbols. If `none` is chosen, the value is displayed as a simple numeric value. If `local` is chosen, the value is displayed in the convention of the locale that is set on the executing

machine, displaying the symbol for the Euro if an accommodating character set is installed. If `international` is chosen then the value is displayed with `EUR`, the international symbol for the Euro.

Syntax:

```
LSEuroCurrencyFormat(currency-value [,type])
```

Example: The following formats the value `70,000` to display as `EUR70,000.00`:

```
#LSEuroCurrencyFormat(70000, "international")#
```

See also: `LSParseEuroCurrency()`, `SetLocale()`

LSIsCurrency()

Description: `LSIsCurrency()` checks whether a string contains a valid currency for the current locale; it returns `TRUE` if it does and `FALSE` if it does not. `LSIsCurrency()` takes a single parameter, the string to be evaluated.

Syntax:

```
LSIsCurrency(String)
```

Example: The following example checks whether a user-supplied date string contains a valid German currency value:

```
<CFSET previous_locale = #SetLocale("German (Standard)")#>
<CFIF #LSIsCurrency(total)# IS "No">
 You entered an invalid currency amount!
</CFIF>
```

See also: `IsNumber()`, `LSIsNumeric()`

LSIsDate()

Description: `LSIsDate()` checks whether a string contains a valid date for the current locale; it returns `TRUE` if it does and `FALSE` if it does not. `LSIsDate()` takes a single parameter, the string to be evaluated.

Syntax:

```
LSIsDate(String)
```

Example: The following example checks whether a user-supplied date string contains a valid German date:

```
<CFSET previous_locale = #SetLocale("German (Standard)")#>
<CFIF #LSIsDate(ship_date)# IS "No">
 You entered an invalid date!
</CFIF>
```

Note

To check U.S. dates, you can use the `IsDate()` function.

See also: `IsDate()`, `IsLeapYear()`, `LSParseDateTime()`, `ParseDateTime()`

LSIsNumeric()

Description: LSIsNumeric() checks whether a specified value is numeric. LSIsNumeric() is the locale-specific version of the IsNumeric() function. LSIsNumeric() takes a single parameter, the value to be evaluated.

Syntax:

```
LSIsNumeric(Value)
```

Example: The following example checks to ensure that a user entered a valid locale-specific age (numeric characters only):

```
<CFIF #LSIsNumeric(age)# IS "No">
 You entered an invalid age!
</CFIF>
```

Note

You can use the simpler IsNumeric() function for U.S. number support.

See also: InputBaseN(), IsNumeric(), Val()

LSNumberFormat()

Description: LSNumberFormat() enables you to display numeric values in a locale-specific readable format. LSNumberFormat() is the local-specific version of the NumberFormat function. LSNumberFormat() takes two parameters—the number to be displayed and an optional mask value. If the mask is not specified, the default mask of ,99999999999999 is used. The complete set of number masks is listed in Table B.5 in the description of the NumberFormat() function.

Syntax:

```
LSNumberFormat(Number [, mask ])
```

Note

To display numbers in any of the U.S. formats, you can use the NumberFormat() function.

Example: The following displays a submitted form field in the default format for the current locale:

```
#LSNumberFormat(FORM.quantity)#
```

See also: DecimalFormat(), DollarFormat(), LSCurrencyFormat(), LSParseNumber(), NumberFormat()

LSParseCurrency()

Description: LSParseCurrency() converts a locale-specific number in string form into a valid number. LSParseCurrency() takes two parameters, the string to be converted and an

optional type. If a type is specified, its value must be `none`, `local`, or `international`. `Type` defaults to all types if not provided.

Syntax:

```
LSParseCurrency(String [, Type])
```

Example: The following example converts a user-supplied currency string into a number:

```
<CFSET sale_price = #LSParseCurrency(FORM.sale_price)#>
```

See also: `LSCurrencyFormat()`, `LSParseNumber()`

LSParseDateTime()

Description: `LSParseDateTime()` converts a locale-specific date in string form into a ColdFusion date/time object. `LSParseDateTime()` is the locale-specific version of the `ParseDateTime()` function. `LSParseDateTime()` takes a single parameter, the string to be converted.

Syntax:

```
LSParseDateTime(String)
```

Example: The following example converts a user-supplied string containing a date into a ColdFusion date/time object:

```
<CFSET ship_date = #LSParseDateTime(FORM.ship_date)#>
```

Note

For U.S. dates and times, you can use the simpler `ParseDateTime()` function.

Caution

Unlike the `ParseDateTime()` function, the `LSParseDateTime()` function does not support POP date/time fields. Passing a POP date/time field to `LSParseDateTime()` generates an error.

See also: `CreateDateTime()`, `ParseDateTime()`

LSParseEuroCurrency()

Description: The `LSParseEuroCurrency()` function converts a currency string that contains the Euro symbol or sign to a number. This function attempts conversion of the string based on all three default currency formats (`none`, `local`, and `international`). This function takes a single parameter, the string you want to convert.

Syntax:

```
LSParseEuroCurrency(currency-string)
```

Example: The following example converts a currency string that is currently formatted with the `international` Euro currency format into a simple number:

```
#LSParseEuroCurrency("EUR1974")#
```

See also: LSEuroCurrencyFormat(), LSParseCurrency()

Caution

For the LSParseEuroCurrency() function to be able to read the Euro currency symbol, the machine running the code must have Euro-enabled fonts installed.

LSParseNumber()

Description: LSParseNumber() converts a locale-specific number in string form into a valid number. LSParseNumber() takes a single parameter, the string to be converted.

Syntax:

```
LSParseNumber(String)
```

Example: The following example converts a user-supplied numeric string into a number:

```
<CFSET quantity = #LSParseNumber(FORM.quantity)#>
```

See also: LSCurrencyFormat(), LSParseCurrency(), Val()

LSTimeFormat()

Description: LSTimeFormat() displays the time portion of a date/time object in a locale-specific readable format. LSTimeFormat() is the locale-specific version of the TimeFormat() function. LSTimeFormat() takes two parameters: The first is the date/time object to be displayed, and the second is an optional mask value that enables you to control exactly how the data is formatted. If no mask is specified, a mask appropriate for the current locale is used. The complete set of date masks is listed in Table B.7 in the description of the TimeFormat() function.

Syntax:

```
LSTimeFormat(Date [, mask ])
```

Example: The following example displays the current time with the default formatting options for the current locale:

```
The time is: #LSTimeFormat(Now())#
```

Note

You can use the simpler TimeFormat() function for U.S. times.

See also: LSDateFormat(), LSNumberFormat(), TimeFormat()

SetLocale()

Description: SetLocale() sets the name of the locale to be used by any subsequent calls to the LS functions. SetLocale() also returns the name of the currently active locale so that it can be saved if necessary.

Syntax:

```
SetLocale(locale)
```

Example: The following example sets the locale to British English and saves the current locale to a local variable:

```
<CFSET previous_locale = #SetLocale("English (UK)")#>
```

See also: `GetLocale()`

LIST MANIPULATION FUNCTIONS

ColdFusion lists are an efficient way to manage groups of information. Lists are made up of elements, which are values separated by delimiting characters. The default delimiter is a comma, but you can change it to any character or string if required. Lists are actually simple two-dimensional arrays. For more complex or multidimensional lists, you should use arrays instead.

This list format is well suited for ColdFusion applications; it is both the format that HTML forms use to submit fields with multiple values and the format used by SQL to specify lists in SQL statements.

When using the list manipulation functions, remember the following:

- **List manipulation functions that add to, delete from, or change a list do not alter the original list passed to them**—Rather, they return an altered list to you for manipulation. If you do need to update the passed list itself, you must use `<CFSET>` to replace the list with the newly modified list.
- **All list functions accept as an optional last parameter a string with delimiters to be used in the processing of the list**—If this parameter is omitted, the default comma delimiter is used.
- **The number 1 is always the starting position in any list**—When referencing list functions, be sure to remember that lists always start at position 1, never 0.
- **When evaluating a list, be aware that ColdFusion will ignore any empty items in a list**—If you have a list defined as `"Laura, John, Sean, , ,Bryan"`, it will be evaluated as a four-element list, not a six-element list.

Note

All the ColdFusion list-manipulation functions have names that begin with the word `list`, making them easy to spot in your code.

Tip

Lists can be used in conjunction with the `<CFLOOP>` tag for processing.

ListAppend()

Description: ListAppend() adds an element to the end of a list and returns the new list with the appended element. ListAppend() takes two parameters; the first is the current list, and the second is the element to be appended.

Syntax:

```
ListAppend(List, Element)
```

Example: The following example appends John to an existing list of users and replaces the old list with the new one:

```
<CFSET Users = #ListAppend(Users, "John")#>
```

See also: ListInsertAt(), ListPrepend(), ListSetAt()

ListChangeDelims()

Description: ListChangeDelims() returns a passed list reformatted to use a different delimiter. ListChangeDelims() takes two parameters; the first is the list to be reformatted, and the second is the new delimiter character.

Syntax:

```
ListChangeDelims(List, Delimiter)
```

Example: The following example creates a new list containing the same elements as the original list but separated by plus signs:

```
<CFSET URLUsers = #ListChangeDelims(Users, "+")#>
```

Tip

The default list delimiter, a comma, is the delimiter used by SQL lists. If you are going to pass ColdFusion lists to SQL statements, you should use the default delimiter.

ListContains(), ListContainsNoCase()

Description: The ListContains() and ListContainsNoCase() functions search through a list to find the first element that contains the specified search text. If the search text is found, the position of the element containing the text is returned. If no match is found, 0 is returned. ListContains() performs a case-sensitive search; ListContainsNoCase() performs a non–case-sensitive search. Both functions take two parameters—the first parameter is the list to be searched, and the second parameter is the value for which to search.

Syntax:

```
ListContains(List, Value)
ListContainsNoCase(List, Value)
```

Example: The following example returns the position of the first element to contain the text cash (regardless of case):

```
Element #ListContainsNoCase(Payments, "cash")# contains the word "cash"
```

Note

`ListContains()` and `ListContainsNoCase()` find substrings within elements that match the specified search text. To perform a search for a matching element, use the `ListFind()` and `ListFindNoCase()` functions instead.

See also: `ListFind()`, `ListFindNoCase()`

`ListDeleteAt()`

Description: `ListDeleteAt()` deletes a specified element from a list. `ListDeleteAt()` takes two parameters; the first is the list to be processed, and the second is the position of the element to be deleted. `ListDeleteAt()` returns a modified list with the specified element deleted. The specified element position must exist; an error message is generated if you specify an element beyond the range of the list.

Syntax:

```
ListDeleteAt(List, Position)
```

Example: The following example deletes the second element in a list, but it first verifies that it exists:

```
<CFIF #ListLen(Users)# GTE 2>
 <CFSET Users = #ListDeleteAt(Users, 2)#>
</CFIF>
```

See also: `ListRest()`

`ListFind()`, `ListFindNoCase()`

Description: The `ListFind()` and `ListFindNoCase()` functions search through a list to find the first element that matches the specified search text. If a matching element is found, the position of that element is returned; if no match is found, `0` is returned. `ListFind()` performs a case-sensitive search, whereas `ListFindNoCase()` performs a non–case-sensitive search. Both functions take two parameters—the first parameter is the list to be searched, and the second parameter is the element text fo which to search.

Syntax:

```
ListFind(List, Value)
ListFindNoCase(List, Value)
```

Example: The following example returns the position of the first element whose value is `MI`:

```
MI is element #ListFind(States, "MI")#
```

Note

`ListFind()` and `ListFindNoCase()` find only elements that exactly match the specified search text. To perform a search for substrings within elements, use the `ListContains()` and `ListContainsNoCase()`functions.

See also: `ListContains()`, `ListContainsNoCase()`

ListFirst()

Description: ListFirst()returns the first element in a list. ListFirst() takes a single parameter, the list to be processed.

Syntax:

```
ListFirst(List)
```

Example: The following example returns the first selection from a field of book titles submitted by a user:

```
The first title you selected is #ListFirst(titles)#
```

See also: ListGetAt(), ListLast(), ListRest()

ListGetAt()

Description: ListGetAt() returns the list element at a specified position. ListGetAt() takes two parameters: The first is the list to process, and the second is the position of the desired element. The value passed as the position parameter must not be greater than the length of the list; otherwise, a ColdFusion error message is generated.

Syntax:

```
ListGetAt(List, Position)
```

Example: The following example returns the name of the fourth selection from a field of book titles submitted by a user:

```
The fourth title you selected is #ListGetAt(titles, 4)#
```

See also: ListFirst(), ListLast(), ListRest()

ListInsertAt()

Description: ListInsertAt() inserts a specified element into a list, shifting all elements after it one position to the right. ListInsertAt() takes three parameters; the first is the list to be processed, the second is the desired position for the new element, and the third is the value of the new element. The position parameter must be no greater than the number of elements in the list. A ColdFusion error message is generated if a greater value is provided.

Syntax:

```
ListInsertAt(List, Position, Value)
```

Example: The following example inserts John into the third position of an existing list of users and replaces the old list with the new one:

<CFSET Users = #ListInsertAt(Users, 3, “John”)#>

See also: ListAppend(), ListPrepend(), ListSetAt()

ListLast()

Description: ListLast() returns the last element in a list. ListLast() takes a single parameter, the list to be processed.

Syntax:

```
ListLast(List)
```

Example: The following example returns the last selection from a field of book titles submitted by a user:

The last title you selected is #ListLast(titles)#

See also: ListFirst(), ListGetAt(), ListRest()

ListLen()

Description: ListLen() returns the number of elements present in a list. ListLen() takes a single parameter: the list to be processed.

Syntax:

```
ListLen(List)
```

Example: The following example returns the number of books selected by a user:

```
You selected #ListLen(titles)# titles
```

ListPrepend()

Description: ListPrepend() inserts an element at the beginning of a list, pushing any other elements to the right. ListPrepend() returns the new list with the prepended element. ListPrepend() takes two parameters: The first is the current list, and the second is the element to be prepended.

Syntax:

```
ListPrepend(List, Element)
```

Example: The following example prepends John to an existing list of users and replaces the old list with the new one:

<CFSET Users = #ListPrepend(Users, "John")#>

See also: ListAppend(), ListInsertAt(), ListSetAt()

ListSort()

Description: ListSort()sorts the items in a list according to the specified sort type and order. The ListSort() function takes four parameters. The first, list, specifies the list you want to sort. The second, sort_type, specifies the type of sort you want to perform. Valid sort types are Numeric (sorts numerically), Text (sorts alphabetically), and TextNoCase (sorts alphabetically without regard to case). The third parameter, sort_order, is optional. You can specify either ASC (ascending) or DESC (descending). When no sort order is chosen,

ascending order is used. The fourth parameter, `delimiter`, specifies the character to use to delimit items in the list. This parameter is also optional. If no delimiter is provided, by default a comma is used.

Syntax:

```
ListSort(list, sort_type [, sort_order] [, delimiter ])
```

Example: The following example sorts a list with the `TextNoCase` sort type. By default, the list items are returned in ascending order, with a comma delimiting items in the list:

```
#ListSort(JohnList, "TextNoCase")#
```

ListQualify()

Description: The `ListQualify()` function returns the contents of a specified list with qualifying characters around each list item. The `ListQualify()` function takes four parameters. The first parameter is the list through which you want to parse. The second parameter is the qualifying character you want to have placed around each list item. The third parameter is the delimiter used in the list you are evaluating, and the fourth is whether you want to evaluate every list item or only list items comprised of alphabetic characters. Defining the elements parameter as either `ALL` or `CHAR` specifies this.

Syntax:

```
ListQualify(list, qualifier [, delimiters ] [, elements ])
```

Example: The following example looks through the elements of a list that is delimited with a comma, placing double quotation marks around all list items, regardless of whether the list item is numeric or comprised of alphabetic characters:

```
#ListQualify(JohnList,"""",",","ALL")#
```

ListRest()

Description: `ListRest()` returns a list containing all the elements after the first element. If the list contains only one element, an empty list (an empty string) is returned. `ListRest()` takes a single parameter: the list to be processed.

Syntax:

```
ListRest(List)
```

Example: The following example replaces a list with the list minus the first element:

```
<CFSET Users = #ListRest(Users)#>
```

See also: `ListDeleteAt()`

ListSetAt()

Description: `ListSetAt()` replaces the value of a specific element in a list with a new value. `ListSetAt()` takes four parameters: The first is the list to be processed, the second is the position of the element to be replaced, the third is the new value, and the fourth is the delimiter or set of delimiters to use. If more than one delimiter is specified, ColdFusion

defaults to the first delimiter in the list. If no delimiter is specified, the delimiter defaults to a comma. The value passed to the position parameter must be no greater than the number of elements in the list; otherwise, a ColdFusion error message is generated.

Syntax:

```
ListSetAt(List, Position, Value [, delimiters])
```

Example: The following searches for an element with the value of `"Honda"` and replaces it with the value `"Harley"`:

```
<CFIF ListFindNoCase(Users, "Honda") GT 0>
 <CFSET Users = ListSetAt(Users, ListFindNoCase(Users, "Honda"), "Harley")>
</CFIF>
```

See also: `ListAppend()`, `ListInsertAt()`, `ListPrepend()`

`ListValueCount()`, `ListValueCountNoCase()`

Description: `ListValueCount()` searches a list for a specific value and returns the number of occurrences it finds of the specified value. The `ListValueCount()` function is case sensitive. To search a list for a specific value regardless of case, use the `ListValueCountNoCase()` function. Both functions take three parameters. The first parameter is the list to search; the second parameter is the value for which to search the list. The third, optional parameter is the character used to delimit elements in the specified list. If the delimiter parameter is omitted, the default comma is used.

Syntax:

```
ListValueCountNoCase(list, value [, delimiters ])
```

Example: The following searches through a list of users to see how many have the first name Laura:

```
#ListValueCountNoCase(usersList, "Laura")#
```

ARRAY MANIPULATION FUNCTIONS

Arrays are special variables made up of collections of data. Array elements are accessed via their indexes into the array; to access the third element of a simple array, you would refer to `array[3]`, for example.

ColdFusion supports arrays with one to three dimensions. A one-dimensional array is similar to a list, whereas a two-dimensional array is similar to a grid. (In fact, under the hood, ColdFusion queries are essentially two-dimensional arrays.) Three-dimensional arrays are more like cubes.

Arrays are created using the `ArrayNew()` function. To create an array, you must specify the number of dimensions needed, between one and three. You don't need to specify how many elements will be stored in the array; ColdFusion automatically expands the array as necessary.

Note

Array elements can be added in any order. If you add an element `10` to an array that has only 5 elements, ColdFusion automatically creates elements 6–9 for you.

ArrayAppend()

Description: `ArrayAppend()` adds an element to the end of an array. `ArrayAppend()` takes two parameters: the array to append the element to and the data to be stored in that element. `ArrayAppend()` returns `TRUE` if the operation was successful.

Syntax:

```
ArrayAppend(Array, Value)
```

Example: The following example appends an element containing the word `January` to an array:

```
#ArrayAppend(Month, "January")#
```

This next example appends an element to a three-dimensional array, setting the value of element `[10][1]`:

```
#ArrayAppend(Users[10][1], "January")#
```

Note

You can set the values of explicit array elements using the `<CFSET>` tag.

See also: `ArrayInsertAt()`, `ArrayPrepend()`

ArrayAvg()

Description: `ArrayAvg()` returns the average numeric value in an array. `ArrayAvg()` takes a single parameter: the array to be checked.

Syntax:

```
ArrayAvg(Array)
```

Example: The following example reports the average cost of items in an array:

```
The average cost of each item in the list is #DollarFormat(ArrayAvg(items))#
```

Note

`ArrayAvg()` works only with arrays containing numeric data. Do not use this function with arrays that contain text data.

See also: `ArrayMin()`, `ArrayMax()`, `ArraySum()`

ArrayClear()

Description: ArrayClear() deletes all data from an array. ArrayClear() takes a single parameter: the array to be deleted. ArrayClear() returns TRUE if the operation was successful.

Syntax:

```
ArrayClear(Array)
```

Example: The following example empties an existing array:

```
<CFSET result = #ArrayClear(Items)#>
```

Note

ArrayClear() does not delete the actual array. Rather, it removes all the contents from it. The array itself remains and can be reused.

See also: ArrayDeleteAt(), ArrayIsEmpty()

ArrayDeleteAt()

Description: ArrayDeleteAt() deletes an element from an array at a specified position, pulling all remaining elements back one place. ArrayDeleteAt() takes two parameters: the array from which to delete the element and the position of the element to delete. ArrayDeleteAt() returns TRUE if the operation was successful.

Syntax:

```
ArrayDeleteAt(Array, Position)
```

Example: The following example deletes the ninth element from an array:

```
#ArrayDeleteAt(Items, 9)#
```

See also: ArrayClear(), ArrayInsertAt()

ArrayInsertAt()

Description: ArrayInsertAt() inserts an element into an array at a specified position, pushing all existing elements with an index greater than the index of the element inserted over one place. ArrayInsertAt() takes three parameters: the array into which to insert the element, the position at which to insert the element, and the data to be stored in that element. ArrayInsertAt() returns TRUE if the operation was successful.

Syntax:

```
ArrayInsertAt(Array, Position, Value)
```

Example: The following example inserts an element containing the word Alaska into the second position of an existing two-dimensional array; it then sets the abbreviation AK into the matching second dimension:

```
<CFSET result = #ArrayInsertAt(States[1], 2, "Alaska")#>
<CFSET States[2][2] = "AK">
```

See also: ArrayAppend(), ArrayDeleteAt(), ArrayPrepend()

ArrayIsEmpty()

Description: ArrayIsEmpty() checks whether an array has data. ArrayIsEmpty() takes a single parameter: the array to be checked. ArrayIsEmpty() returns TRUE if the array is empty and FALSE if not.

Syntax:

```
ArrayIsEmpty(Array)
```

Example: The following example reports whether an array is empty:

```
<CFOUTPUT>Array empty: #YesNoFormat(ArrayIsEmpty(Users))#</CFOUTPUT>
```

See also: ArrayClear(), ArrayLen(), IsArray()

ArrayLen()

Description: ArrayLen() returns the length of a specified array. ArrayLen() takes a single parameter, the array to be checked.

Syntax:

```
ArrayLen(Array)
```

Example: The following example reports the size of an array:

```
The items array has #ArrayLen(items)# elements
```

See also: ArrayIsEmpty(), ArrayResize()

ArrayMax()

Description: ArrayMax() returns the largest numeric value in an array. ArrayMax() takes a single parameter, the array to be checked.

Syntax:

```
ArrayMax(Array)
```

Example: The following example reports the cost of the most expensive item in an array:

```
The most expensive item in the list costs #DollarFormat(ArrayMax(items))#
```

Note

ArrayMax() works only with arrays containing numeric data. Do not use this function with arrays that contain text data.

See also: ArrayAvg(), ArrayMin(), ArraySum()

ArrayMin()

Description: ArrayMin() returns the smallest numeric value in an array. ArrayMin() takes a single parameter: the array to be checked.

Syntax:

```
ArrayMin(Array)
```

Example: The following example reports the cost of the least expensive item in an array:

```
The least expensive item in the list costs #DollarFormat(ArrayMin(items))#
```

Note

ArrayMin() works only with arrays containing numeric data. Do not use this function with arrays that contain text data.

See also: ArrayAvg(), ArrayMax(), ArraySum()

ArrayNew()

Description: ArrayNew() is used to create an array. ArrayNew() takes a single parameter: the number of dimensions needed. Valid dimensions are one through three. ArrayNew() returns the array itself.

Syntax:

```
ArrayNew(Dimensions)
```

Example: The following example creates a one-dimensional array:

```
<CFSET Users = #ArrayNew(1)#>
```

Note

After an array is created, ColdFusion automatically expands it as necessary. Use the ArrayResize() function to manually resize an array.

See also: IsArray(), ListToArray()

ArrayPrepend()

Description: ArrayPrepend() adds an element to the beginning of an array. ArrayPrepend() takes two parameters: the array into which to insert the element and the data to be stored in that element. ArrayPrepend() returns TRUE if the operation was successful.

Syntax:

```
ArrayPrepend(Array, Value)
```

Example: The following example inserts an element containing the word Alabama into the beginning of an array:

```
#ArrayPrepend(States, "Alabama")#
```

Note

You can set the values of explicit array elements using the <CFSET> tag.

See also: ArrayAppend(), ArrayInsertAt()

ArrayResize()

Description: ArrayResize() changes the size of an array, padding it with empty elements if necessary. ArrayResize() takes two parameters: the array to be resized and the size at which to resize it. ArrayResize() returns TRUE if the operation is successful.

Syntax:

```
ArrayResize(Array, Size)
```

Example: The following example creates an array and immediately resizes it to hold 100 elements:

```
<CFSET Users = #ArrayNew(1)#>
<CFSET result = #ArrayResize(Users, 100)#>
```

Tip

Dynamically expanding arrays is a slow operation. You can dramatically optimize ColdFusion's array processing by resizing the array to the anticipated size immediately after creating it with ArrayNew().

See also: ArrayLen(), ArraySet()

ArraySet()

Description: ArraySet() initializes one or more elements in a one-dimensional array with a specified value. ArraySet() takes four parameters—the array itself, the element starting and ending positions, and the value to use. ArraySet() returns TRUE if the operation is successful.

Syntax:

```
ArraySet(Array, Start, End, Value)
```

Example: The following example sets elements 1–100 with the value 0:

```
#ArraySet(OrderItems, 1, 100, 0)#
```

See also: ArrayResize(), ArraySort(), ArraySwap()

ArraySort()

Description: ArraySort() sorts the data in an array. ArraySort() takes three parameters: the array to be sorted, the sort type, and an optional sort order of either ascending or descending. If the sort order is omitted, the default order of ascending is used. ArraySort() supports three sort types, as listed in Table B.11.

TABLE B.11 `ArraySort()` SORT TYPES

Type	Description
`Numeric`	Sorts numerically
`Text`	Sorts text alphabetically, uppercase before lowercase
`TextNoCase`	Sorts text alphabetically; case is ignored

Syntax:

```
ArraySort(Array, Type [, Order])
```

Example: The following example sorts an array alphabetically using a non–case-sensitive sort (also known as a *dictionary sort*):

```
#ArraySort(Users, "textnocase")#
```

Note

`ArraySort()` sorts the actual passed array, not a copy of it.

See also: `ArraySet()`, `ArraySwap()`

ArraySum()

Description: `ArraySum()` returns the sum of all values in an array. `ArraySum()` takes a single parameter: the array to be checked.

Syntax:

```
ArraySum(Array)
```

Example: The following example reports the total cost of all items in an array:

```
The total cost of all item in the list is #DollarFormat(ArraySum(items))#
```

Note

`ArraySum()` works only with arrays containing numeric data. Do not use this function with arrays that contain text data.

See also: `ArrayAvg()`, `ArrayMin()`, `ArrayMax()`

ArraySwap

Description: `ArraySwap()` is used to swap the values in two array elements. `ArraySwap()` takes three parameters: the array itself and the positions of the two elements to be swapped. `ArraySwap()` returns `TRUE` if the operation is successful.

Syntax:

```
ArraySwap(Array, Position1, Position2)
```

Example: The following example swaps elements `10` and `11` in an array:

```
#ArraySwap(Users, 10, 11)#
```

See also: `ArraySet()`, `ArraySort()`

ArrayToList()

Description: `ArrayToList()` converts a one-dimensional ColdFusion array into a list. `ArrayToList()` takes two parameters: the array to be converted and an optional list delimiter. If no delimiter is specified, the default (comma) delimiter is used. `ArrayToList()` creates a new list.

Syntax:

```
ArrayToList (Array [, Delimiter])
```

Example: The following example converts an array of users into a list:

```
<CFSET UserList = #ArrayToList(UserArray)#>
```

See also: `ListToArray()`

IsArray()

Description: `IsArray()` checks whether a variable is a valid ColdFusion array; it also determines that an array has a specific number of dimensions. `IsArray()` takes two parameters: the variable to be checked and an optional number of dimensions to check for. `IsArray()` returns `TRUE` if the variable is an array and `FALSE` if not.

Syntax:

```
IsArray (Array [, Dimension])
```

Example: The following example checks whether a variable named `Users` is an array:

```
#IsArray(Users)#
```

This example checks whether `Users` is a three-dimensional array:

```
#IsArray(Users, 3)#
```

See also: `ArrayIsEmpty()`

ListToArray()

Description: `ListToArray()` converts a ColdFusion list to a one-dimensional array. `ListToArray()` takes two parameters: the list to be converted and an optional list delimiter. If no delimiter is specified, the default (comma) delimiter is used. `ListToArray()` creates a new array.

Syntax:

```
ListToArray(List [, Delimiter])
```

Example: The following example converts a list of users into an array:

```
<CFSET UserArray = #ListToArray(UserList)#>
```

Caution

When evaluating a list, be aware that ColdFusion will ignore any empty items in a list. If you have a list defined as `"Laura, John, Sean, , ,Bryan"`, it will be evaluated as a four-element list, not a six-element list.

See also: `ArrayToList()`

STRUCTURE MANIPULATION FUNCTIONS

ColdFusion *structures* are special data types that contain one or more other variables. Structures are a way to group related variables together.

Duplicate()

Description: The `Duplicate()` function returns a deep copy of a variable. After a variable is duplicated, the duplicate copy of the variable contains no reference to the original variable. The `Duplicate()` function takes a single parameter, the name of the variable you want to duplicate.

Syntax:

```
Duplicate(variable_name)
```

Example: The following example duplicates a variable `application.dsn` in to the request scope:

```
<CFLOCK SCOPE="APPLICATION" TYPE="READONLY" TIMEOUT="10">
<CFSET REQUEST.DSN=Duplicate(APPLICATION.DSN)>
</CFLOCK>
```

Note

The `Duplicate()` function is useful for making copies of complex data types, such as structures and queries.

Caution

You cannot duplicate COM, CORBA, or Java objects with the `Duplicate()` function. An attempt to do this causes ColdFusion to throw an exception error.

See also: `StructCopy()`

IsStruct()

Description: The `IsStruct()` function returns `TRUE` if the variable being evaluated is a structure. The `IsStruct()` function takes a single parameter, the variable to evaluate.

Syntax:

```
IsStruct(variable)
```

Example: The following example checks to see whether the variable `People` is a structure:

```
#IsStruct(People)#
```

StructClear()

Description: `StructClear()` deletes all data from a structure. `StructClear()` takes a single parameter: the structure to be cleared. `StructClear()` always returns `YES` if the operation is successful.

Syntax:

```
StructClear(Structure)
```

Example: The following example empties an existing structure:

```
<CFSET result = #StructClear(Items)#>
```

Note

`StructClear()` does not delete the actual structure. Rather, it removes all the contents from it. The structure itself remains and can be reused.

See also: `StructDelete()`, `StructIsEmpty()`

StructCopy()

Description: `StructCopy()` returns a clone of the specified structure, with all the keys and values of the specified structure intact. The `StructCopy()` function takes a single parameter, the structure to be copied.

Syntax:

```
StructCopy(structure)
```

Example: The following example produces a clone of the structure `People`:

```
StructCopy(People)
```

Note

On the surface, `StructCopy()` appears similar to the `Duplicate()` function. The main difference is that when using the `Duplicate()` function to copy a structure, you are creating a completely new copy of the specified structure, with no reference to the original. `StructCopy()` assigns any nested structures, objects, or query values to the new structure by making reference to the original.

See also: `Duplicate()`, `StructClear()`

StructCount()

Description: `StructCount()` returns the number of items in a specified structure. `StructCount()` takes a single parameter: the structure to be checked. ColdFusion will throw an error if the structure you are attempting to evaluate with `StructCount()` does not exist.

Syntax:

```
StructCount(Structure)
```

Example: The following example reports the number of elements in a structure:

```
The items structure has #StructCount(items)# elements
```

See also: `StructIsEmpty()`

StructDelete()

Description: `StructDelete()` deletes an item from a structure. `StructDelete()` takes three parameters: the structure, the name of the key to be deleted, and an optional flag that specifies how to handle requests to delete a key that does not exist. `StructDelete()` returns `YES` if the operation was successful and `NO` if not. If an attempt is made to delete a key that does not exist, and the `IndicateNotExisting` flag is not set to `TRUE`, the `StructDelete()` returns `YES`.

Syntax:

```
StructDelete(Structure, Key [, IndicateNotExisting])
```

Example: The following example deletes the `name` key from a `user` structure:

```
#StructDelete(user, name)#
```

See also: `StructClear()`, `StructKeyExists()`, `StructIsEmpty()`

StructFind()

Description: The `StructFind()` function searches through a structure to find the key that matches the specified search text. If a matching key is found, the value in that key is returned; if no match is found, an empty value is returned. This function takes two parameters: the first is the structure to be searched, and the second is the key for which to search.

Syntax:

```
StructFind(Structure, Key)
```

Example: The following example returns the username stored in a `user` structure:

```
User name is #StructFind(user, first_name)#
```

StructInsert()

Description: `StructInsert()` inserts an item into a structure. `StructInsert()` takes four parameters: the structure, the name of the key to be inserted, the value, and an optional flag that specifies whether a key can be overwritten. `StructInsert()` returns `YES` if the operation was successful and `NO` if not. Values can be overwritten unless `AllowOverwrite` is set to `FALSE`.

Syntax:

```
StructInsert(Structure, Key, Value [, AllowOverwrite])
```

Example: The following example inserts a key named `first_name` into a `user` structure:

```
#StructInsert(user, "first_name", "Ben")#
```

See also: `StructDelete()`

StructIsEmpty()

Description: `StructIsEmpty()` checks whether a structure has data. `StructIsEmpty()` takes a single parameter: the structure to be checked. `StructIsEmpty()` returns `TRUE` if the array is empty and `FALSE` if not.

Syntax:

```
StructIsEmpty(Structure)
```

Example: The following example reports whether a structure is empty:

```
<CFOUTPUT>Strucure empty: #YesNoFormat(StructIsEmpty(Users))#</CFOUTPUT>
```

See also: `StructClear()`, `StructCount()`, `StructKeyExists()`

StructKeyArray()

Description: The `StructKeyArray()` function returns the keys of a specified structure in an array. The `StructKeyArray()` function takes a single parameter, the structure to be evaluated.

Syntax:

```
StructKeyArray(structure)
```

Example: The following example returns, as an array, the keys of the specified structure:

```
#StructKeyArray(family)#
```

Note

The array the `StructKeyArray()` function returns is in no particular order. To sort the array, you use the `ArraySort()` function. Also, remember that if the structure you are attempting to evaluate with the `StructKeyArray()` function does not exist, ColdFusion throws an error.

See also: `StructClear()`, `StructDelete()`, `StructFind()`, `StructUpdate()`

StructKeyExists()

Description: `StructKeyExists()` checks whether a structure contains a specific key. `StructKeyExists()` takes two parameters: the structure to be checked and the key for which to look. `StructKeyExists()` returns `TRUE` if the key exists and `FALSE` if not.

Syntax:

```
StructKeyExists(Structure, Key)
```

Example: The following checks whether a key named `first_name` exists:

```
<CFIF #StructKeyExists(user, "first_name")#>
```

See also: `StructCount()`, `StructIsEmpty()`

StructKeyList()

Description: `StructKeyList()` returns a list of keys in the specified ColdFusion structure. The `StructKeyList()` function takes two parameters. The first parameter is the structure to be evaluated. The second, optional parameter is the delimiter you want to separate each item in the list returned.

Syntax:

```
StructKeyList(structure [,delimiter])
```

Example: The following retrieves a list of keys, delimited by a comma, from the specified structure:

```
#StructKeyList(MyStructure ,",")#
```

See also: `StructKeyArray()`, `StructClear()`

StructNew()

Description: `StructNew()` creates a new structure. `StructNew()` takes no parameters and returns the structure itself.

Syntax:

```
StructNew()
```

Example: The following example creates a simple structure:

```
<CFSET Orders = #StructNew()#>
```

StructUpdate()

Description: The `StructUpdate()` function updates the specified key in a given structure with a specified value. The function returns `YES` if the update is successful and throws an exception if an error is encountered. The `StructUpdate` function takes three parameters. The first is the structure you are attempting to update. The second is the key within the structure you are attempting to update, and the third is the value with which you want to update the specified key.

Syntax:

```
StructUpdate(structure, key, value)
```

Example: The following example updates the specified structure:

```
#StructUpdate(stFamily, "wife", "Laura")#
```

See also: `StructClear()`, `StructDelete()`, `StructFind()`

QUERY MANIPULATION FUNCTIONS

ColdFusion uses queries to return sets of data. Most queries are created with the <CFQUERY> tag, but other tags (<CFPOP> and <CFLDAP>) also return data in queries. ColdFusion also enables you to programmatically create your own queries using the QueryNew function and set query values using QuerySetCell.

IsQuery()

Description: IsQuery() checks whether a variable is a valid ColdFusion query. IsQuery() takes a single parameter: the variable to be checked. IsQuery() returns TRUE if the variable is a query and FALSE if not.

Syntax:

```
IsQuery (Query)
```

Example: The following example checks whether a variable named Users is a query:

```
<CFIF #IsQuery(Users)#>
```

Tip

IsQuery() is particularly useful within custom tags that expect queries as parameters. IsQuery() can be used to check that a valid value was passed before any processing occurs.

QueryAddColumn()

Description: QueryAddColumn() adds a new column to a specified query and populates that column with data from a one-dimensional array. The number of the column that was added is returned. If necessary, the contents of other query columns are padded to ensure that all columns retain the same number of rows. The QueryAddColumn() function takes three parameters. The first is the name of the query to which you want to add a column. The second is the name of the new column you are about to create, and the third is the name of the array that will be used to populate the query column.

Syntax:

```
QueryAddColumn(query, column-name, array-name)
```

Example: The following example creates a new column called UserName in the query GetUsers and updates that column with the data contained in the UserNameArray array:

```
#QueryAddColumn(qGetUsers, "UserName", UserNameArray)#>
```

Tip

When using the QueryAddColumn() function, remember that you cannot add columns to cached queries.

See also: QueryNew(), <CFQUERY>, QueryAddRow()

QueryAddRow()

Description: QueryAddRow() adds a row to an existing ColdFusion query. QueryAddRow() takes two parameters: the query to which to add a row and an optional number of rows to add. If the number of rows is omitted, the default number of 1 is used.

Syntax:

```
QueryAddRow(Query [, Number])
```

Example: The following example creates a new query called Users and adds 10 rows to it:

```
<CFSET Users = #QueryNew("FirstName, LastName")#>
<CFSET temp = #QueryAddRow(Users, 10)#>
```

See also: QueryNew(), QuerySetCell()

QueryNew()

Description: QueryNew() either returns an empty query with a user-defined set of columns, as specified with the columnlist argument, or—if an empty string is specified in the columnlist argument—returns an empty query with no columns.

Syntax:

```
QueryNew( columnlist)
```

Example: The following example creates a new query called Users with columns for FirstName and LastName:

```
<CFSET Users = #QueryNew("FirstName, LastName")#>
<CFSET temp = #QueryAddRow(Users, 10)#>
```

See also: QueryAddRow(), QuerySetCell()

QuerySetCell()

Description: QuerySetCell() is used to set the values of specific cells in a query. QuerySetCell() takes four parameters: the query name, the column name, the value, and an optional row number. If the row number is omitted, the cell in the last query row is set.

Syntax:

```
QuerySetCell(Query, Column, Value [, Row])
```

Example: The following example sets the FirstName column in the third row to the value Ben:

```
<CFSET temp = #QuerySetCell(Users, "FirstName", "John", 3)#>
```

Note

Query cells can also be set using the <CFSET> tag, treating the query as a two-dimensional array.

See also: QueryAddRow(), QueryNew()

SECURITY FUNCTIONS

ColdFusion supports advanced security contexts that let you create complete security systems to secure your applications. Security is managed and maintained using the ColdFusion Administrator. After security is established, you can make a call to the `<CFAUTHENTICATE>` tag to return security information. Use these security functions to interact with that security information.

AuthenticatedContext()

Description: `AuthenticatedContext()` returns the name of the currently valid security context. This function does not take any parameters.

Syntax:

```
AuthenticatedContext()
```

Example: The following example retrieves the currently valid security context and outputs the name of the context to the user:

```
The currently active security context is:
<CFOUTPUT>#AuthenticatedContext()#</CFOUTPUT>
```

See also: `IsAuthenticated()`, `AuthenticatedUser()`

AuthenticatedUser()

Description: `AuthenticatedUser()` returns the name of the currently authenticated user. This function does not take any parameters.

Syntax:

```
AuthenticatedUser()
```

Example: The following example retrieves the name of the currently authenticated user and outputs the name of that user:

```
The user currently authenticated is: <CFOUTPUT>#AuthenticatedUser()#</CFOUTPUT>
```

See also: `IsAuthenticated()`, `AuthenticatedContext()`

IsAuthenticated()

Description: `IsAuthenticated()` checks whether a user has been authenticated with the `<CFAUTHENTICATE>` tag. `IsAuthenticated()` takes two parameters: the user to be checked and the security context to be checked against. If no security context is specified, `IsAuthenticated()` checks to see whether the user has been authenticated by any security context. `IsAuthenticate()` returns `TRUE` if the user has been authenticated and `FALSE` if not.

Syntax:

```
IsAuthenticated(User, Context)
```

Example: The following example checks to see whether a user has been authenticated as an administrator:

```
<CFIF #IsAuthenticated("#user#", "administrator")#
```

Note

To use the IsAuthenticated() function, advanced security must be enabled in the ColdFusion Administrator and security contexts must already have been defined.

See also: `IsAuthorized()`

`IsAuthorized()`

Description: `IsAuthorized()` checks whether a user is authorized to perform specific actions. `IsAuthorized()` takes three parameters: the resource type to be checked, the resource to be checked, and an optional action. `IsAuthorized()` returns `TRUE` if the action is authorized and `FALSE` if not. Table B.12 lists the valid resource types.

Syntax:

```
IsAuthorized(ResourceType, ResourceName [, Action])
```

TABLE B.12 RESOURCE TYPES

Type
Application
CFML
File
Datasource
Component
Collection
CustomTag
UserObject

Example: The following example checks whether a user is authorized to use a specific data source:

```
<CFIF #IsAuthorized(Datasource, "OWS")#
```

Note

To use this function, advanced security must be enabled in the ColdFusion Administrator and security contexts must already have been defined.

See also: `IsAuthenticated()`

IsProtected()

Description: The IsProtected() function checks whether the resource that is being accessed is protected in the security context of the authenticated user. The IsProtected() function takes three parameters. The first, resourcetype, specifies the type of resource being accessed. The second, resourcename, specifies the name of the actual resource being protected. The third parameter, action, should not be specified if the resourcetype is specified as COMPONENT or CUSTOMTAG. For all other resource types, an action is required. Table B.13 outlines all resource types along with all possible actions for that specific resource type.

TABLE B.13 RESOURCE TYPES AND CORRESPONDING ACTIONS

Resource Type	Possible Actions
APPLICATION	ALL; USECLIENTVARIABLES
CFML	Actions for this tag are specified by RESOURCENAME
FILE	READ; WRITE
DATASOURCE	ALL; CONNECT; SELECT; INSERT; UPDATE; DELETE; SP (Stored Procedure)
COMPONENT	No actions exist for this resource type.
COLLECTION	DELETE; OPTIMIZE; PURGE; SEARCH; UPDATE
CUSTOMTAG	No actions exist for this resource type.
USEROBJECT	Actions for this resource type are specified from within the ColdFusion Administrator.

Syntax:

```
IsProtected(resourcetype, resourcename [, action])
```

See also: IsAuthorized()

SYSTEM FUNCTIONS

The ColdFusion system functions enable you to perform manipulation of file paths, create temporary files, and verify file existence.

DirectoryExists()

Description: DirectoryExists() checks for the existence of a specified directory and returns either YES or NO. DirectoryExists() takes a single parameter: the name of the directory for which to check. The directory name cannot be a relative path but must be specified as a fully qualified path.

Syntax:

```
DirectoryExists(Directory)
```

Example: The following example checks for the existence of a directory, creating it if it does not exist:

```
<CFIF #DirectoryExists("#directory#")# IS "No">
 <CFFILE ACTION="CREATE" DIRECTORY="#directory#">
</CFIF>
```

See also: `FileExists()`

`ExpandPath()`

Description: `ExpandPath()` converts a relative path into a fully qualified path. `ExpandPath()` takes a single parameter: the path to be converted.

Syntax:

```
ExpandPath(Path)
```

Example: The following example returns the full path of the server's default document:

```
#ExpandPath("index.cfm")#
```

See also: `GetTemplatePath()`

`FileExists()`

Description: `FileExists()` checks for the existence of a specified file and returns either `YES` or `NO`. `FileExists()` takes a single parameter: the name of the file for which to check. The filename cannot be a relative path but must be specified as a fully qualified path.

Syntax:

```
FileExists(File)
```

Example: The following example checks for the existence of an image file before using it in an `IMG` tag:

```
<CFIF #FileExists("C:\root\images\logo.gif")#>
 <IMG SRC="/images/logo.gif">
</CFIF>
```

Tip

Use the `ExpandPath()` function so you don't have to hard-code the filename passed to the `FileExists()` function; use `ExpandPath()` to convert the relative path to an actual filename.

See also: `DirectoryExists()`

`GetCurrentTemplatePath()`

Description: `GetCurrentTemplatePath()` returns the complete path of the template calling this function. The `GetCurrentTemplatePath()` function does not take any parameters.

Syntax:

```
GetCurrentTemplatePath()
```

Example: The following example, placed anywhere inside a CFML template, would output the complete path of the template running this code to the user:

```
<CFOUTPUT>#GetCurrentTemplatePath()#</CFOUTPUT>
```

See also: `GetBaseTemplatePath()`, `FileExists()`

GetDirectoryFromPath()

Description: `GetDirectoryFromPath()` extracts the drive and directory (with a trailing backslash) from a fully specified path. `GetDirectoryFromPath()` takes a single parameter: the path to be evaluated.

Syntax:

```
GetDirectoryFromPath(Path)
```

Example: The following example returns the directory portion of a current template's full file path:

```
#GetDirectoryFromPath(GetTemplatePath())#
```

See also: `GetFileFromPath()`

GetFileFromPath()

Description: `GetFileFromPath()` extracts the filename from a fully specified path. `GetFileFromPath()` takes a single parameter: the path to be evaluated.

Syntax:

```
GetFileFromPath(Path)
```

Example: The following example returns the filename portion of a temporary file:

```
#GetFileFromPath(GetTempFile(GetTempDirectory(), "CF"))#
```

See also: `GetDirectoryFromPath()`

GetMetricData()

Description: The `GetMetricData()` function returns a structure containing system and application metrics based on the mode specified. The `GetMetricData()` function takes a single parameter, the mode in which this function should run. Valid modes for the `GetMetricData()` function are `PERF_MONITOR` (returns data that would otherwise be displayed in the NT Performance Monitor), `SIMPLE_LOAD` (returns a value that is calculated internally, based on the server queue, and reflects the general load on the server), `PREV_REQ_TIME` (displays the time it took the server to process the last request), and `AVG_REQ_TIME` (returns the average time it takes the server to process a request).

Syntax:

```
GetMetricData(mode)
```

Example: The following example returns the general load on the server:

```
<CFOUTPUT>#GetMetricData(SIMPLE_LOAD)#</CFOUTPUT>
```

GetTempDirectory()

Description: `GetTempDirectory()` returns the full path of the operating system temporary directory with a trailing backslash. `GetTempDirectory()` takes no parameters.

Syntax:

```
GetTempDirectory()
```

Example: The following example returns the name of a temporary file beginning with the letters `CF` in the Windows temporary directory:

```
#GetTempFile(GetTempDirectory(), "CF")#
```

See also: `GetTempFile()`

GetTempFile()

Description: `GetTempFile()` creates and returns the full path to a temporary file for use by your application. The returned filename is guaranteed to be unique. `GetTempFile()` takes two parameters: The first is the directory where you want the temporary file created, and the second is a filename prefix of up to three characters. You can't omit the prefix, but you can pass an empty string (`""`).

Syntax:

```
GetTempFile(Directory, Prefix)
```

Example: The following example returns the name of a temporary file beginning with the letters `CF` in the Windows temporary directory:

```
#GetTempFile(GetTempDirectory(), "CF")#
```

Tip

To create a temporary file in the Windows temporary directory, pass the `GetTempDirectory()` function as the directory parameter.

See also: `GetTempDirectory()`

GetTemplatePath()

Description: `GetTemplatePath()` returns the fully qualified path of the base template being processed. `GetTemplatePath()` takes no parameters.

Syntax:

```
GetTemplatePath()
```

Example: The following example returns the full path of the base template being processed:

```
Processing: #GetTemplatePath()#
```

Note

`GetTemplatePath()` returns the path of the base template being processed. If you are using `GetTemplatePath()` in a template that is being included in a second template, the path of that second template is returned.

Caution

To maintain backward compatability, ColdFusion Server still processes the `GetTemplatePath()` function. However, you should be aware that this function is deprecated. The `GetBaseTemplatePath()` function supersedes this function and should be used in place of `GetTemplatePath()`.

`SetProfileString()`, `GetProfileString()`

Description: The `SetProfileString()` function sets the value of a profile entry in an initialization file. The `GetProfileString()` function retrieves the value of an entry in an initialization file.

Syntax:

```
SetProfileString(iniPath, section, entry, value)
GetProfileString(iniPath, section, entry)
```

CLIENT VARIABLE MANIPULATION FUNCTIONS

Client variables enable you to store client information so it is available between sessions. Client variables can be accessed just like any other ColdFusion variables; standard variable access tools, such as `<CFSET>`, can therefore be used to set variables. In addition, these functions provide special variable manipulation capabilities.

Note

For more information about client variables and how they are used, see Chapter 19, "Introducing the Web Application Framework."

`DeleteClientVariable()`

Description: `DeleteClientVariable()` deletes the client variables whose names are passed as parameters. Unlike other ColdFusion variables, client variables persist over time and must be deleted with this function. `DeleteClientVariable()` takes a single parameter: the

name of the variable to delete. `DeleteClientVariable()` returns `TRUE` if the variable was deleted. This function deletes a variable even if it did not previously exist. To ensure that a variable actually exists before using the `DeleteClientVariable()` function, use `IsDefined()` to test for the existence of the variable.

Syntax:

```
DeleteClientVariable(Variable)
```

Example: The following example deletes a variable named `login_name` and sets a local variable with the function's return value:

```
<CFSET DeleteSuccessful = #DeleteClientVariable("login_name")#>
```

GetClientVariablesList()

Description: `GetClientVariablesList()` returns a comma-delimited list of the read/write client variables available to the template. The standard read-only system client variables, listed in Table B.14, are not returned. `GetClientVariablesList()` takes no parameters.

Syntax:

```
GetClientVariablesList()
```

TABLE B.14 READ-ONLY CLIENT VARIABLES

Variable	Description
`CFID`	Unique ID assigned to this client.
`CFToken`	Unique security token used to verify the authenticity of a `CFID` value.
`URLToken`	Text to append to URLs; contains both `CFID` and `CFToken`. (Appended automatically to `<CFLOCATION>` URLs.)

Example: The following example retrieves the entire list of read/write client variables:

```
#ListLen(GetClientVariablesList())#
➥read-write client variables are currently active
```

Tip

The list of variables returned by the `GetClientVariablesList()` function is comma delimited, which makes it suitable for processing with the ColdFusion list functions.

EXPRESSION EVALUATION FUNCTIONS

ColdFusion enables you to perform *dynamic expression evaluation*. This is an advanced technique that allows you to build and evaluate expressions on-the-fly.

Dynamic expression evaluations are performed on string expressions. A string expression is just that—a string that contains an expression. The string `"1+2"` contains an expression that, when evaluated, returns `3`. String expressions can be as simple or as complex as necessary.

DE()

Description: DE() stands for delay evaluation. This function is designed for use with the IIF() and Evaluate() functions. It takes a string as a parameter and returns the same string enclosed within quotation marks; all double quotation marks are escaped. This enables you to pass a string to IIf() and Evaluate() without them being evaluated.

Syntax:

```
DE(String)
```

Example: The following example uses DE() to ensure that the string "A" is evaluated, instead of the variable "A".

```
#Evaluate(DE("A"))#
```

Evaluate()

Description: Evaluate() is used to evaluate string expressions. Evaluate() takes one or more string expressions as parameters and evaluates them from left to right.

Syntax:

```
Evaluate(String1, ...)
```

Example: The following example evaluates the variable "A":

```
#Evaluate("A")#.
```

IIf()

Description: IIf() evaluates a Boolean condition and evaluates one of two expressions depending on the results of that evaluation. If the Boolean condition returns TRUE, the first expression is evaluated; if the condition returns FALSE, the second expression is evaluated.

Syntax:

```
IIF(Boolean condition, Expression if TRUE, Expression if FALSE)
```

Example: The following example determines whether #cnt# has a value of 1; it evaluates "A" if it does and "B" if it does not:

```
#IIf("#cnt# IS 1", "A", "B")#
```

SetVariable()

Description: SetVariable() sets a specified variable to a passed value.

Syntax:

```
SetVariable(Variable, Value)
```

Example: The following example sets variable #cnt# to the value returned by the passed expression:

```
#SetVariable(#cnt#, "A")#
```

BIT AND SET MANIPULATION FUNCTIONS

ColdFusion provides a complete set of bit manipulation functions for use by advanced developers only. These functions enable you to manipulate the individual bits within a 32-bit integer.

The complete set of bit manipulation functions is listed in Table B.15. The description for each function is given in the C/C++ syntax.

Note

Any start, length, or position parameters passed to the bit manipulation functions must be in the range of `0–31`.

TABLE B.15 COLDFUSION BIT AND SET MANIPULATION FUNCTIONS

Function	Description
`BitAnd (x, y)`	x and y
`BitMaskClear (x, start, length)`	x with length bits starting from start cleared
`BitMaskRead (x, start, length)`	The value of the length bits starting from start
`BitMaskSet (x, mask, start, length)`	x with mask occupying the length bits starting from start
`BitNot (x)`	x
`BitOr (x, y)`	x | y
`BitSHLN (x, n)`	x << n
`BitSHRN (x, n)`	x >> n
`BitXor (x, y)`	x^y

CONVERSION FUNCTIONS

These functions are provided to enable you to easily convert data from one type to another.

`JavaCast()`

Description: The `JavaCast()` function is used to indicate that a ColdFusion variable be converted to be passed as an argument to an overloaded method of a Java object. The `JavaCast()` function should be used only to pass scalar and string arguments. This function takes two parameters. The first is the data type the variable should be converted to prior to being passed. Possible valid data types are Boolean, `int`, `long`, `double`, and `String`. The second parameter is the variable that should be converted.

Syntax:

```
JavaCast(type, variable)
```

Example: The following example converts the specified variable to the `int` data type:

```
#JavaCast("int", myNum)#
```

See also: `CreateObject()`, `CFOBJECT()`

JSStringFormat()

Description: The `JSStringFormat()` function formats a specified string so that it is safe to use with JavaScript. The `JSStringFormat()` function takes a single parameter, the string to be formatted. The function escapes any special JavaScript characters so you can put these characters in strings that you pass as JavaScript. The characters that are escaped by the `JSStringFormat()` function include the `'` (single quotation mark), `"` (double quotation mark), and newline characters.

Syntax:

```
JSStringFormat(string)
```

Example: The following example converts the specified string to a format that is safe to use with JavaScript:

```
#JSStringFormat("mystring")#
```

ToString()

Description: The `ToString()` function attempts to convert any value, including binary values, into a string. If the value specified cannot be converted into a string, the `ToString()` function throws an exception error. The `ToString()` function takes a single parameter, the value to be converted to a string.

Syntax:

```
ToString(any_value)
```

Example: The following example converts the specified value into a string:

```
#ToString(BinaryData)#
```

XMLFormat()

Description: The `XMLFormat()` function formats a specified string so that it is safe to use with XML. The `XMLFormat()` function takes a single parameter, the string to be formatted. The function escapes any special XML characters so you can put these characters in strings that you pass as XML. The characters that are escaped by the `XMLFormat()` function include `>` (greater than sign), `<` (less than sign), `'` (single quotation mark), `"` (double quotation mark), and `&` (ampersand).

Syntax:

```
XMLFormat(string)
```

Example: The following example converts the specified string to a format that is safe to use with XML:

```
#XMLFormat("mystring")#
```

MISCELLANEOUS FUNCTIONS

The following functions are listed here to give you access to some of the lesser-known, yet nonetheless important, functions available to you in ColdFusion 5.

CreateObject()

Description: The `CreateObject()` function enables you to create COM, CORBA, or Java objects. The `CreateObject()` function takes three parameters. The first parameter is the class, defined as the `ProgID` for the object you want to invoke. The second parameter, `context`, is optional. Valid context types are `InProc`, `Local`, and `Remote`. When no context type is specified, Registry settings are used. The third parameter, `server name`, is required when the context type is set to `Remote`. This parameter defines a valid server running the object you want to invoke.

Syntax:

```
CreateObject("COM", class, context, serverName)
```

CreateUUID()

Description: The `CreateUUID()` function returns a 35-character string representation of a unique 128-bit number. The `CreateUUID()` function takes no parameters. UUID in this instance stands for universally unique identifier.

Syntax:

```
CreateUUID()
```

Example: The following example creates and then outputs a UUID:

```
<CFOUTPUT>#CreateUUID()#</CFOUTPUT>
```

Note

The `CreateUUID()` function is useful when you need a unique identifier that can persist in a distributed environment.

GetBaseTagData()

Description: `GetBaseTagData()` is used within subtags. It returns an object containing data from a specified ancestor tag. `GetBaseTagData()` takes two parameters: the name of the tag whose data you want returned and an optional instance number. If no instance is specified, the default value of `1` is used.

Syntax:

```
GetBaseTagData(Tag [, InstanceNumber])
```

Example: The following example retrieves the data in a caller `<CFHTTP>` tag:

```
#GetBaseTagData(CFHTTP)#
```

Note

Not all tags contain data (for example, the `<CFIF>` tag). Passing a tag that contains no data to the `GetBaseTagData()` function causes an exception to be thrown.

See also: `GetBaseTagList()`

GetBaseTagList()

Description: `GetBaseTagList()` is used within subtags. It returns a comma-delimited list of base tag names. The returned list is in calling order, with the parent tag listed first.

Syntax:

```
GetBaseTagList()
```

Example: The following example displays the top-level calling tag:

```
<CFOUTPUT>The top level tag is #ListFirst(GetBaseTagList())#</CFOUTPUT>
```

See also: `GetBaseTagData()`

GetBaseTemplatePath()

Description: `GetBaseTemplatePath()` returns the full path of the base template. The `GetBaseTemplatePath()` function takes no parameters.

Syntax:

```
GetBaseTemplatePath()
```

Example: The following example displays the base template path information to the user:

```
<CFOUTPUT> #GetBaseTemplatePath()#</CFOUTPUT>
```

See also: `GetCurrentTemplatePath()`, `FileExists()`

GetFunctionList()

Description: The `GetFunctionList()` function returns a structure containing all built-in functions available in ColdFusion. The `GetFunctionList()` function takes no parameters.

Syntax:

```
GetFunctionList()
```

Example: The following returns a list of all functions available in ColdFusion:

```
<CFSET JohnsFunctions = GetFunctionList()>
<CFOUTPUT>#StructCount(JohnsFunctions)# functions<BR><BR></CFOUTPUT>
<CFLOOP COLLECTION="#JohnsFunctions#" ITEM="key">
     <CFOUTPUT>#key#<BR></CFOUTPUT>
</CFLOOP>
```

GetHTTPRequestData()

Description: The GetHttpRequestData() function retrieves the HTTP request headers and body and makes them available for use in a CFML template. The GetHttpRequestData() function takes no arguments and returns a structure containing the following variables: headers (returns all the HTTP request headers), content (returns, in string or binary format, the form data that was submitted by the client), method (returns the request method as a string), protocol (returns the server-based CGI variable Server_Protocol as a string).

Syntax:

```
GetHttpRequestData()
```

Example: The following example stores all information retrieved by the GetHttpRequestData() function in a variable:

```
<CFSET headerInfo = #GetHttpRequestData()#>
```

GetTickCount()

Description: GetTickCount() returns a tick count used to perform timing tests with millisecond accuracy. The value returned by GetTickCount is of no use other than to compare it to the results of another GetTickCount() call to check time spans.

Syntax:

```
GetTickCount()
```

Example: The following example tests how long a code block takes to execute:

```
<CFSET count1 = #GetTickCount()#>
...
<CFSET count2 = #GetTickCount()#>
<CFSET duration = count2-count1>
<CFOUTPUT>Code took #duration# milliseconds to execute</CFOUTPUT>
```

IsBinary()

Description: IsBinary() tests whether a specified value is binary. Returns TRUE if the specified value is binary and FALSE if it is not. The IsBinary() function takes a single parameter, the value to be evaluated.

Syntax:

```
IsBinary(value)
```

Example: The following checks whether a specified value is a binary value:

```
#IsBinary(myValue)#
```

See also: ToBinary(), ToBase64(), IsNumeric()

IsBoolean()

Description: IsBoolean() determines whether a value can be converted to a Boolean value. (Boolean values have two states only, ON and OFF or TRUE and FALSE.) IsBoolean() takes a

single parameter: the number, string, or expression to evaluate. When evaluating numbers, `IsBoolean()` treats `0` as `FALSE` and any non-zero value as `TRUE`.

Syntax:

```
IsBoolean(Value)
```

Example: The following example checks to see whether a value can be safely converted into a Boolean value before passing it to a formatting function:

```
<CFIF #IsBoolean(status)# IS "Yes">
 #YesNoFormat(status)#
</CFIF>
```

See also: `YesNoFormat()`

IsCustomFunction()

Description: `IsCustomFunction()` is used to check whether or not a specified function is a user-defined function. `IsCustomFunction()` takes a single parameter, the function to be verified to determine whether it is a user-defined function. It returns `Yes` if the function being evaluated is a user-defined function and `No` if it is not.

Syntax:

```
IsCustomFunction(function)
```

Example: The following example checks the function `UDF` to determine whether it is in fact a user-defined function:

```
#IsCustomFunction('UDF')#
```

IsDebugMode()

Description: `IsDebugMode()` checks whether a page is being sent back to the user in debug mode. `IsDebugMode()` returns `TRUE` if debug mode is on and `FALSE` if not. `IsDebugMode()` takes no parameters.

Syntax:

```
IsDebugMode()
```

Example: The following code writes debug data to a log file if it's in debug mode:

```
<CFIF #IsDebugMode()#>
 <CFFILE ACTION= APPEND" FILE="log.txt" OUTPUT="#debug_info#">
</CFIF>
```

IsDefined()

Description: `IsDefined()` determines whether a specified variable exists. `IsDefined()` returns `TRUE` if the specified variable exists and `FALSE` if not. `IsDefined()` takes a single parameter: the variable for which to check. This parameter can be passed as a fully qualified variable, with a preceding variable type designator. The variable name must be enclosed in

quotation marks; otherwise, ColdFusion checks to see whether the contents of the variable exist rather than the variable itself.

Syntax:

```
IsDefined(Parameter)
```

Example: The following example checks whether a variable of any type named USER_ID exists:

```
<CFIF #IsDefined("USER_ID")#>
```

The next example checks to see whether a CGI variable named USER_ID exists and ignores variables of other types:

```
<CFIF #IsDefined("CGI.USER_ID")# >
```

Note

IsDefined() is a little more complicated to use than ParameterExists(), but it does enable you to dynamically evaluate and redirect expressions. Because ParameterExists() is a deprecated function, you should always use IsDefined() in its place.

See also: Evaluate(), IsSimpleValue(), ParameterExists()

IsNumeric()

Description: IsNumeric() checks whether a specified value is numeric. IsNumeric() takes a single parameter: the value to be evaluated. IsNumeric() returns TRUE if the specified string can be converted to a number and FALSE if it can't.

Syntax:

```
IsNumeric(Value)
```

Example: The following example checks to ensure that a user entered a valid age (numeric characters only):

```
<CFIF #IsNumeric(age)# IS "No">
 You entered an invalid age!
</CFIF>
```

Note

Use the LSIsNumeric() function for international number support.

See also: InputBaseN(), LSIsNumeric(), Val()

IsSimpleValue()

Description: IsSimpleValue() checks whether a value is a string, number, TRUE/FALSE value, or DATE/TIME value. IsSimpleValue() takes a single parameter: the value to be checked. IsSimpleValue() returns TRUE if the value is a simple value and FALSE if not.

Syntax:

```
IsSimpleValue(Value)
```

Example: The following example checks to see that a description field is a simple value:

```
<CFIF #IsSimpleValue(Description)#>
```

See also: `Evaluate()`, `IsDefined()`, `ParameterExists()`

ParameterExists()

Description: `ParameterExists()` checks to see whether a specified variable exists. `ParameterExists()` returns `TRUE` if the specified variable exists and `FALSE` if not. `ParameterExists()` takes a single parameter: the variable for which to check. This parameter can be passed as a fully qualified variable, with a preceding variable type designator. Do not enclose the variable name in quotation marks.

Syntax:

```
ParameterExists(Parameter)
```

Example: The following example checks whether a variable of any type named `USER_ID` exists:

```
<CFIF ParameterExists(USER_ID) IS "Yes">
     <CFOUTPUT>The User ID is: #User_ID#</CFOUTPUT>
</CFIF>
```

The next example checks to see whether a CGI variable named `USER_ID` exists and ignores variables of other types:

```
<CFIF ParameterExists(CGI.USER_ID) IS "Yes">
     <CFOUTPUT>The User ID is: #CGI.USER_ID#</CFOUTPUT>
</CFIF>
```

Caution

The `ParameterExists()` function is supported only to maintain backward compatibility. The function itself is deprecated and should not be used. Use `IsDefined()` in place of `ParameterExists()` when writing new CFML code.

See also: `Evaluate()`, `IsDefined()`, `IsSimpleValue()`

PreserveSingleQuotes()

Description: `PreserveSingleQuotes()` instructs ColdFusion to not escape single quotation marks contained in values derived from dynamic parameters. `PreserveSingleQuotes()` takes a single parameter: the string to be preserved. This function is particularly useful when you are constructing SQL statements that will contain ColdFusion variables.

Syntax:

```
PreserveSingleQuotes(String)
```

Example: The following example uses `PreserveSingleQuotes()` to ensure that a dynamic parameter in a SQL statement is included correctly:

```
SELECT * FROM Customers
WHERE CustomerName IN ( #PreserveSingleQuotes(CustNames)#)
```

QuotedValueList(), ValueList()

Description: `QuotedValueList()` and `ValueList()` drive one query with the results of another. Both functions take a single parameter—the name of a query column—and return a list of all the values in that column. `QuotedValueList()` returns a list of values that are each enclosed within quotation marks and separated by commas. `ValueList()` returns the list separated by commas, but not enclosed in quotation marks.

Syntax:

```
QuotedValueList(Column)
ValueList(Column)
```

Example: The following example passes the results from one query to a second query:

```
SELECT * FROM Customers
WHERE CustomerType IN (#QuotedList(CustType.type)#)
```

Note

The `QuotedValueList()` and `ValueList()` functions are typically used only when constructing dynamic SQL statements.

Tip

The values returned by `QuotedValueList()` and `ValueList()` are both in the standard ColdFusion list format and can therefore be manipulated by the list functions.

Tip

As a general rule, you always should try to combine both the queries into a single SQL statement, unless you need to manipulate the values in the list. The time it takes to process one combined SQL statement is far less than the time it takes to process two simpler statements.

URLDecode()

Description: The `URLDecode()` function decodes a specified `URLEncoded` string. The `URLDecode()` function takes a single parameter, the `URLEncoded` string to be decoded.

Syntax:

```
URLDecode(urlEncodedString)
```

Example: The following example decodes a `URLEncoded` string:

```
#URLDecode(myEncodedString)#
```

See also: `URLEncodedFormat()`

URLEncodedFormat()

Description: URLEncodedFormat() encodes a string in a format that safely can be used within URLs. URLs can't contain spaces or any nonalphanumeric characters. The URLEncodedFormat() function replaces spaces with a plus sign; nonalphanumeric characters are replaced with equivalent hexadecimal escape sequences. URLEncodedFormat() takes a single parameter —the string to be encoded—and returns the encoded string.

Syntax:

```
URLEncodedValue(String)
```

Note

ColdFusion automatically decodes all URL parameters that are passed to a template.

Example: The following example creates a URL with a name parameter that safely can include any characters:

```
<A HREF="details.cfm?name=#URLEncodedFormat(name)#">Details</A>
```

WriteOutput()

Description: The WriteOutput() function appends text to the page output stream, regardless of the setting specified in a given CFSETTING block. The WriteOutput() function takes a single parameter, the string to output to the page.

Syntax:

```
WriteOutput(string)
```

Example: The following example writes the specified output to the page output stream:

```
#WriteOutput("Users Processed: Sean, Misty, Parker, Laura, Peggy, Delane")#
```

See also: <CFSETTING>