flexible search™

Logging Template Reference Manual

(Click on a link to access the desired section)

Version 2.0

SLICCWARE™

Publication Date -- January 24, 2003

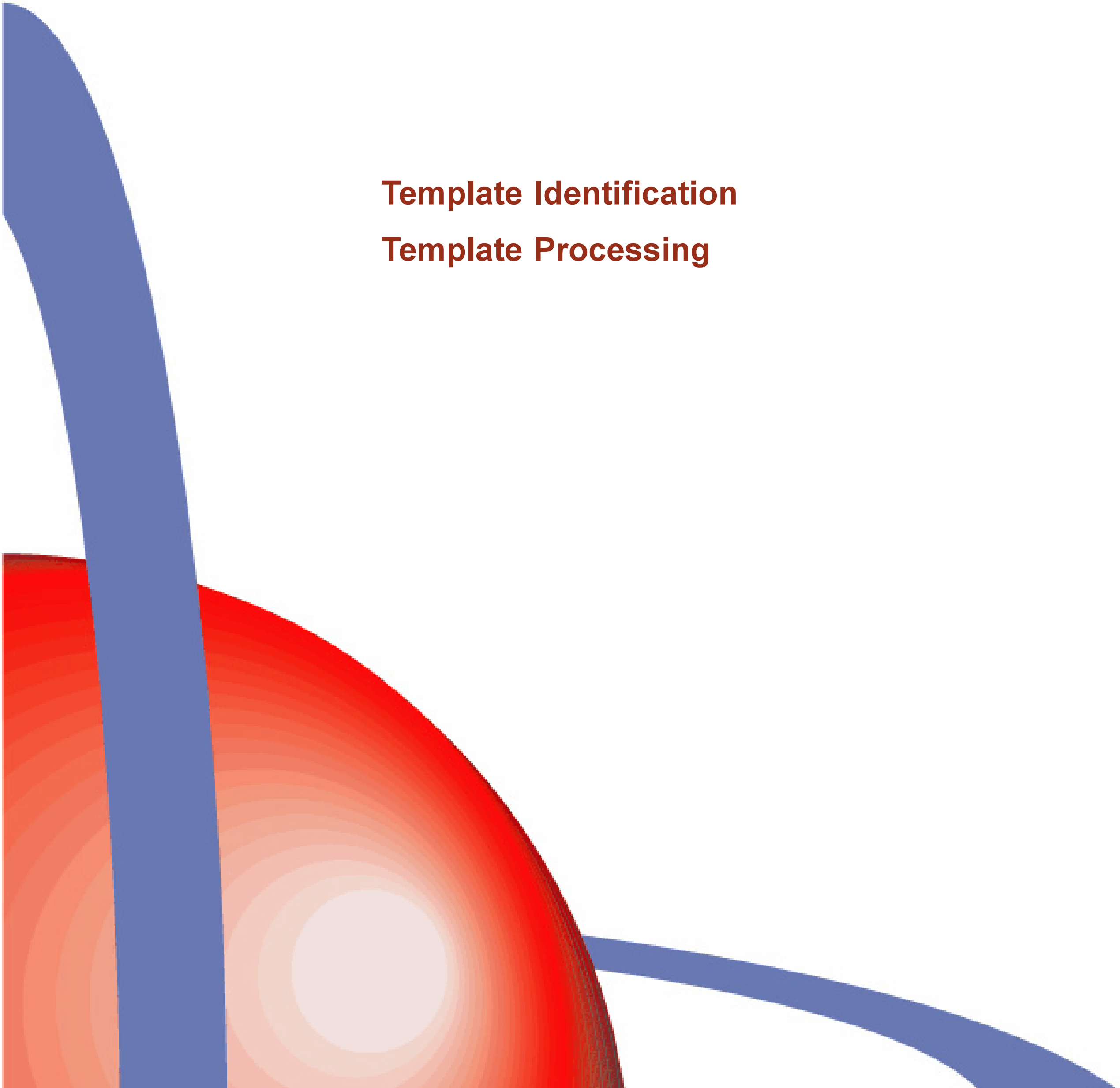
Table of Contents

Table of Contents (cont.)

Section 1 – Overview

Template Identification

Template Processing



Section 1.1 – Template Identification

A *Logging Template* is used to format an entry for the transaction log.

In processing the template, each character is output exactly as it appears within the template until a grave accent is encountered. A grave accent is the lower case character found in the upper left corner of the keyboard to the left of the one and exclamation point key, directly above the tab key. Grave accents are used to encapsulate control statements, expressions, variables, system values, and data field names that are to be processed by the template processing software.

When a grave accent is encountered special processing takes over. Once special processing has completed, normal processing continues at the new location, outputing each character exactly as it appears within the template until the next grave accent is encountered.

One of the functions supported by special processing is the setting of system variables. At the top of the *Logging Template*, a series of system values must be set. That is the only time throughout the entire *Logging Template* that any values are set. Three system values must be set:

__TYPE must be set to *LOG*;
__NAME is the name to be assigned to the template
__RESOURCE is the resource ID to be assigned to the display template

Nothing on the line where a system value is being set is copied to the transaction log, not even the terminating line-feed.

FORMAT

```
`__TYPE=LOG`
`__NAME=name`
`__RESOURCE=rsrc`
```

Where: **name** is the name assigned to the display template
rsrc is the ID assigned to the display template

EXAMPLES

Create a Display Template named BasicSearch with a resource ID of 803, used to display data stored within the search load with ID 2.

```
`__TYPE=LOG`
`__NAME=BasicTrace`
`__RESOURCE=804`
```

• • •

Flexible Search Template Processing supports displaying variables passed in from an HTML form or set within the *Logic Template*, as well as a large selection of system values. Variables are displayed exactly as entered into the system. To alter the format use a format statement.

Expressions involving variables are supported by *Flexible Search Template Processing*. Expressions can include multiple levels of parenthesis and any of a number of supported operators. An expressions is displayed as an unsigned integer. To display an expression in some other format use a format statement. For more information on expressions see Section .

A number of functions are supported by *Flexible Search Template Processing* for direct display or for use in expressions.

Flexible Search Template Processing also supports two constructs for controling the processing of the Logging template. These are:

the *IF .. ENDIF Conditional Construct*,
and the *IF .. ELSE .. ENDIF Multiple Choice Construct*

EXAMPLES

Section 2 – Processing Control

Conditional Construct

Multiple Choice Construct

Section 2.1 – Conditional Construct

Conditional Constructs are used to control logic flow and support conditional processing in the Logging Template. A *Conditional Construct* consists of an *IF (x)* statement enclosed within a pair of grave accents (where *x* is an expression to be evaluated), followed by one or more items that are to be processed if the expression evaluates to anything other than zero, and an *ENDIF* statement, also enclosed within a pair of grave accents.

When the *IF (x)* statement is encountered, the expression *x* is evaluated as a numeric expression. If the expression evaluates to zero, everything after the *IF (x)* statement up to the matching *ENDIF* statement is ignored. If the expression evaluates to anything other than zero, everything after the *IF (x)* statement up to the matching *ENDIF* statement is processed. When the matching *ENDIF* statement is encountered, processing continues as it had before the *IF (x)* statement was encountered.

Multiple levels of nested parenthesis are supported within the numerical expression *x*. Within each pair of parenthesis can be any number of operands connected by any of the operators described in section 5. Evaluation of the numerical expression *x* proceeds from the inner-most level of parenthesis outward, and then from left to right within each level. There is no precedence assigned to the operators.

Conditional Constructs may be nested to any level and may be contained within and contain any other construct.

FORMAT

Conditional processing within a Logging Template is accomplished by bracketing the section which is to be processed only when a certain condition is met by a *Conditional Construct*.

```
`IF ( exp )`

        text

`ENDIF`
```

Where: **exp** is the expression to be evaluated
text is the section of text to be processed only if **exp** evaluates to anything other than zero.

EXAMPLES

Use a Conditional Construct to only display the address if all 3, address, city and state are present.

```
    `IF (  Address  &&  City  &&  State  )`

        `Address` `City`, `State` `ZipCode`

    `ENDIF`
```

Section 2.2 – Multiple Choice Construct

Multiple Choice Constructs are used to control logic flow and support conditional processing in the Display Template. An *Multiple Choice Construct* consists of an *IF (x)* statement, enclosed within a pair of grave accents, (where *x* is an expression to be evaluated), followed by one or more items that are to be processed if the expression evaluates to anything other than zero, followed by an *ELSE* statement, also enclosed within a pair of grave accents, followed by one or more items that are to be processed if the expression evaluates to zero, and an *ENDIF* statement, enclosed within a pair of grave accents as well.

When an *IF (x)* statement is encountered, the expression following it is evaluated as a numeric expression. If the expression evaluates to zero, everything after the expression up to the matching *ELSE* statement is ignored. Everything between the *ELSE* and *ENDIF* statements is processed. If the expression evaluates to anything other than zero, everything after the expression up to the matching *ELSE* statement is processed. Everything between the *ELSE* and *ENDIF* statements is ignored. When the matching *ENDIF* statement is encountered, processing continues as it had before the *IF (x)* statement was encountered.

Multiple levels of nested parenthesis are supported within the numerical expression *x*. Within each pair of parenthesis can be any number of operands connected by any of the operators described in section 5. Evaluation of the numerical expression *x* proceeds from the inner-most level of parenthesis outward, and then from left to right within each level. There is no precedence assigned to the operators.

Multiple Choice Constructs may be nested to any level and amy be contained within and contain other constructs.

FORMAT

Command level alternative processing is done by bracketing the lines to be processed conditionally and the alternative lines by an *Multiple Choice Construct*.

```
`IF ( exp )`

      txtA

`ELSE`

      txtB

`ENDIF`
```

Where: **exp** is the expression to be evaluated
txtA is the section of text to be processed only if **exp** evaluates to anything other than zero.
txtB is the section of text to be processed only if **exp** evaluates to zero.

EXAMPLES

Use a Multiple Choice Construct to determine the format of an HTML display based upon the number of results returned.

```
`IF (  __COUNT  )`

    Displaying `__FIRST` through `__LAST`

    of `__COUNT` Items found

`ELSE`

    No Items found

`ENDIF`
```

Section 3 – Literals and Expressions

Numeric Literals

Text String Literals

Date-Time Literals

Dynamic Values

Dynamic Text Strings

Dynamic Date-Time Values

Section 3.1 – Numeric Literals

A literal is a value defined within the Display Template for use as function arguments, or within an expression. Three kinds of literals are supported: numeric literals, text string literals, and date-time literals.

A numeric literal is either a decimal constant or a hexadecimal constant. A decimal constant is expressed by a string of digits optionally preceded by a plus or minus sign. A hexadecimal constant is expressed by a zero followed by an “x” followed by from 1 to 8 hexadecimal digits (0-f). A numeric literal is always either enclosed within a pair of grave accents or as part of a larger expression enclosed within a pair of grave accents.

EXAMPLES

Compare the literal 2012 with the value in the variable “Value”.

```
`IF ( 2012 == Value )`
```

Compare the hexadecimal literal for 39 with the value in the variable “Flags”.

```
`IF( 0x0027 == Flags )`
```

Section 3.2 – Text String Literals

A literal is a value defined within the Display Template for use as function arguments, or within an expression. Three kinds of literals are supported: numeric literals, text string literals, and date-time literals.

A text string literal is a string of text enclosed within a pair of quotation marks. A text string literal must not contain any control characters such as a newline character, carriage return, or tab.

```
`IF ( _@FIND( KeywordString,  “politic” ) )`
```

If a quotation mark is to be contained within of the string of text,it must be preceded by a backslash as in the example below.

```
`IF ( _@FIND( KeywordString,  “\”Jim Crow\”” ) )`
```

Back-slashes within a text string literal, not found at the end of a line, and not preceding a quotation mark are treated as simple text, and not removed. That way a DOS pathname may be entered just as it would be on a command line.

```
`IF ( _@FIND( SearchPath,  “\test\data” ) )`
```

A back-slash at the end of a text string literal must be entered twice to avoid having it mistaken for an attempt to include the terminating quotation mark within the text string literal.

```
`IF ( _@FIND( SearchPath,  “\test\data\\” ) )`
```

Section 3.3 – Date-Time Literals

A literal is a value defined within the Display Template for use as function arguments, or within an expression. Three kinds of literals are supported: numeric literals, text string literals, and date-time literals.

A date-time literal is a recognizable date, time, date-time constant, or a time duration value -- enclosed within a pair of quotation marks.

“4:05 pm”	*actual time*
“July 4, 2001”	*actual date*
“4:00 pm July 4, 2001”	*actual date and time*
“1 day”	*a time duration*

A large number of date and time formats are recognized. Some of the recognizable formats for date constants are:

“January 5, 2001”	**“Jan 5, 2001”**	**“20010105”**	**“010105”**
“01/05/2001”	**“01/05/01”**	**“1/5/2001”**	**“1/5/01”**
“01-05-2001”	**“01-05-01”**	**“1-5-2001”**	**“1-5-01”**
“5 January 2001”	**“5 Jan 2001”**		
“05/01/2001”	**“05/01/01”**	**“5/1/2001”**	**“5/1/01”**
“05-01-2001”	**“05-01-01”**	**“5-1-2001”**	**“5-1-01”**

A large number of date and time formats are recognized. Some of the recognizable formats for time constants are:

“4:05:30 PM”	**“4:05:30 pm”**	**“4:05 PM”**	**“4:05 pm”**
“16:05:30”	**“16:05”**	**“160530”**	**“1605”**

Section 3.3 – Date-Time Literals (cont.)

Any recognized format for a date constant can be combined with any recognized format for a time constant to produce a date-time constant. The date constant can either precede or follow the time constant. The only exception is the numeric format in which the date must always precede the time literal with a blank seperating them.

"20010105 160530"	*numeric format*
"20010105 1605"	*numeric format*
"010105 1605"	*numeric format*

"January 5, 2001 4:05 pm"	**"05 Jan 2001 4:05 pm"**
"4:05 pm January 5, 2001"	**"4:05 pm 05 Jan 2001"**
"4:05:30 PM 01/05/2001"	**"05-01-01 16:05"**

A time duration value is a numeric value coupled with a time duration unit. For constructing a time duration value, a limitted number of time duration units are recognized. They are:

"minute"	**"min"**	**"minutes"**	**"mins"**
"hour"	**"hr"**	**"hours"**	**"hrs"**
"day"	**"da"**	**"days"**	**"das"**
"month"	**"mo"**	**"months"**	**"mos"**
"year"	**"yr"**	**"years"**	**"yrs"**

Section 3.4 – Dynamic Values

Dynamic data is data, which unlike literals, must be determined at run time. Three types of *dynamic data* exist: *dynamic values*, *dynamic text strings*, and *dynamic date-time values*.

A dynamic value may be an HTML variable, a value generating function, or a numeric expression. A dynamic value is always either enclosed within a pair of grave accents or as part of a larger expression enclosed within a pair of grave accents.

An HTML variable is a name to which a value has been assigned, either as part of a name-value pair within the query URL, or as the result of an assignement statement within the Logic Template.

To learn more about assigning values to HTML variables see section 2.1 of the Logic Template Reference Manual.

A value generating function is any numeric or string function which returns a numeric value. It is expressed by the function name, followed by the argument list enclosed in paranthesis.

```
`IF ( _@IMATCH( “News”, Source ) )`
              . . .              . . .
`ENDIF`
```

To learn more about value generating functions, and to learn how to use them, see sections 4.1 thru 4.13.

A numeric expression is any combination of *HTML variables*, *field values*, *value generating functions*, and *numeric literals*, separated by appropriate operators, and grouped by one or more pairs of parenthesis.

Evaluation of the expression proceeds from the inner-most level of parenthesis outward, and then from left to right within each level. There is no precedence given to any operator over any other.

```
`( 123 + MyValue)`
`( ( ( BaseValue + Increment ) * 3 ) - 1 )`
```

To learn more about what operators are supported within numeric expressions and how to use them, see section 5.

Section 3.5 – Dynamic Text Strings

Dynamic data is data, which unlike literals, must be determined at run time. Three types of *dynamic data* exist: *dynamic values*, *dynamic text strings*, and *dynamic date-time values*.

A dynamic text string may be an HTML variable, or a string generating function. A dynamic text string is always either enclosed within a pair of grave accents or as part of a larger expression enclosed within a pair of grave accents.

An HTML variable is a name to which a value has been assigned, either as part of a name-value pair within the query URL, or as the result of an assignement statement within the Logic Template.

To learn more about assigning text strings to HTML variables see section 2.1 of the Logic Template Reference Manual.

A text generating function is any function which returns a text string. It is expressed by the function name, followed by the argument list enclosed in paranthesis. String generating functions may be nested as in the second example below.

```
`_@SUBSTR( MyPath, 16, 255 )`
`_@CAT( “/data/test/”, _@SUBSTR( MyPath, 16, 255 ) )`
```

To learn more about text generating functions, and to learn how to use them, see sections 4.14 thru 4.18.

Section 3.6 – Dynamic Date-Time Values

Dynamic data is data, which unlike literals, must be determined at run time. Three types of *dynamic data* exist: *dynamic values*, *dynamic text strings*, and *dynamic date-time values*.

A dynamic date-time value may be an HTML variable, a field date-time value, a relative date value, a relative time value, a relative date-time value, a simple date-time expression, or a complex date-time expression.

A dynamic date-time value can be used within any numerical expression where a date-time variable can be used. If a dynamic date-time value is to be displayed, it must be done using a Format command. Displaying it directly would result in it being displayed as a meaningless unsign integer value.

“today”	*relative date*
“this hour”	*relative time*
“now”	*relative date-time*

Simple Expressions
“today - 3 days”
“1 hour + 30 mins”

define 12:00 noon last February 4th
“last year + 34 days + 12 hours”

Complex Expressions

Use NewDate if present, and OldDate if not,
then subtract 3 days to obtain the start date
((NewDate |& OldDate) - “3 days”)

Display todays date within a Display Template
`Format( “today”, “DATE” )`

An HTML variable is a name to which a value has been assigned, either as part of a name-value pair within the query URL, or as the result of an assignement statement within the Logic Template.

To learn more about assigning dates and times to HTML variables see section 2.1 of the Logic Template Reference Manual.

A relative date value is any one of a number of supported strings enclosed within a pair of quotation marks. The result returned is relative to the current system date.

“today”	*relative date*

The following relative date values are recognized:

“yesterday”	*the current date minus one day*
“today”	*the current date*
“tomorrow”	*the current date plus one day*
“last day”	*the current date minus one day*
“this day”	*the current date*
“next day	*the current date plus one day*
“last week”	*Sunday of last week*
“this week”	*Sunday of the current week*
“next week”	*Sunday of next week*
“last month”	*the first of last month*
“this month”	*the first of this month*
“next month”	*the first of next month*
“last year”	*January first of last year*
“this year”	*January first of this year*
“next year	*January first of next year*

A relative time value is any one of a number of supported strings enclosed within a pair of quotation marks. The result returned is relative to the current system time.

“this hour”	*relative time*

The following relative time values are recognized:

“last minute”	*the current date and time to the minute minus one minute*
“this minute”	*the current date and time to the minute*
“next minute”	*the current date and time to the minute plus one minute*
“last hour”	*the current date and time to the hour minus one hour*
“this hour”	*the current date and time to the hour*
“next hour”	*the current date and time to the hour plus one hour*

A relative date-time value is any one of a number of supported strings enclosed within a pair of quotation marks. The result returned is relative to the current system date and time.

“now” *relative date-time*

Only the following relative date-time value is recognized:

“now” *the current date and time*

A simple date-time expression can be either a combination of one or more time duration values connected by plus or minus signs; or a combination of one or more time duration values plus a relative date value or a relative time value or a relative date-time value, connected by plus or minus signs. A simple date-time expression is enclosed within quotation marks. No individual components of the expresion are independently enclosed within quotation marks.

Evaluation of the expression proceeds from left to right. There is no precedence given to any operator over any other.

“today - 3 days”
“1 hour + 30 mins”

A simple expression to define
12:00 noon last February 4th
“last year + 34 days + 12 hours”

A complex date-time expression is any combination of simple date-time expressions date, time, or date-time constants; date-time extractions; relative date, time or date-time values; and time duration values, and reasonable numeric functions; grouped by one or more pairs of parenthesis.

Evaluation of the expression proceeds from the inner-most level of parenthesis outward, and then from left to right within each level. There is no precedence given to any operator over any other.

Use NewDate if present, and OldDate if not,
then subtract 3 days to obtain the start date
((NewDate |& OldDate) - “3 days”)

To learn more about what operators are supported within complex date-time expressions and how to use them, see section 5.

Section 4 – Functions

NUMERIC FUNCTIONS

- _@MIN
- _@FIRST
- _@MAX
- _@ISSET

STRING FUNCTIONS

- _@STRLEN
- _@IMATCH
- _@STRICMP
- _@IFIND
- _@IRFIND
- _@CAT
- _@LOWER
- _@MATCH
- _@STRCMP
- _@FIND
- _@RFIND
- _@SUBSTR
- _@CAPS
- _@UPPER

_@MIN is used to compare two or more values and return the smallest of them.

_@MIN is a value generating numeric function.

FORMAT

_@MIN(val_1, val_2 [, val_2 [...]])

Where: **val_1** is the first value to be compared
val_2 is the second value to be compared
val_3 is the third value to be compared
... continue for addition values

EXAMPLES

Section 4.2 – Numeric Function – _@MAX

_@MAX is used to compare two or more values and return the smallest of them.

_@MAX is a value generating numeric function.

FORMAT

_@MAX(val_1, val_2 [, val_2 [...]])

Where: **val_1** is the first value to be compared
val_2 is the second value to be compared
val_3 is the third value to be compared
... continue for addition values

EXAMPLES

_@FIRST is used to return the first in a list of values that is non-zero.

_@FIRST is a value generating numeric function.

FORMAT

_@FIRST(val_1, val_2 [, val_2 [...]])

Where:		
	val_1	is the first value to be tested
	val_2	is the second value to be tested
	val_3	is the third value to be tested
	...	continue for addition values

EXAMPLES

_@ISSET is used to determine the state of an attribute within a selection mask. If the identified attribute is set within the attribute selection mask, a one is returned. If the identified attribute is not set within the attribute selection mask, a zero is returned.

An attribute selection mask is created using the *Load Selection Mask* command and is used to support filtering of results using attribute matching. The function, *_@ISSET,* can be used to determine whether or not the option associated with the attribute should be set when reflecting the user's selections back to him in the next form.

_@ISSET is a value generating numeric function.

For more information on inputing attribute data from selection lists within HTML forms see section 6.2 of the *Display Template Reference Manua*l.

For more information on the *Load Selection Mask* command see Section 2.2 of the *Logic Template Reference Manual*.

For more information of Filtering by attributes see sections 9.25 through 9.30 of the *Logic Template Reference Manual*.

FORMAT

_@ISSET(MaskID, AttrID)

Where:	**MaskID**	is the ID of the selection mask being tested
	AttrID	is ID of the individual attribute being tested

EXAMPLES

_@STRLEN returns the number of characters in a text string.

_@STRLEN is a value generating text function.

FORMAT

_@STRLEN(string)

Where: **string** is the text string literal or the name of the text or string data field to be sized

EXAMPLES

_@MATCH is used to compare two text strings. The two text strings are passed as arguments to the function. They are compared on a character by character basis.

Wild cards are supported in the comparison. A question mark (?) matches any character, and an asterisk (*) matches any string of characters.

If the two text strings are identical, the function returns 1.

If the two text strings are not identical, the function returns zero.

_@MATCH is a value generating text function.

FORMAT

_@MATCH(string_1, string_2)

Where: **string_1** is the first text literal or the name of the first text or string data field to be tested
string_2 is the second text literal or the name of the second text or string data field to be tested

EXAMPLES

_@IMATCH is used to compare two text strings. The two text strings are passed as arguments to the function. They are compared on a character by character basis without regard to case. That is the two strings “HELLO” and “hello” would be considered identical.

Wild cards are supported in the comparison. A question mark (?) matches any character, and an asterisk (*) matches any string of characters.

If the two text strings when converted to the same case (all upper or all lower) are identical, the function returns 1.

If the two text strings when converted to the same case (all upper or all lower) are not identical, the function returns 0.

_@IMATCH is a value generating text function.

FORMAT

_@IMATCH(string_1, string_2)

Where: **string_1** is the first text literal or the name
of the first text or string data field to be tested
string_2 is the second text literal or the name
of the second text or string data field to
be tested

EXAMPLES

_@STRCMP is used to compare two text strings just as *strcmp* is used in the *C* programming language. The two text strings are passed as arguments to the function. They are compared on a character by character basis.

Wild cards are supported in the comparison. A question mark (?) matches any character, and an asterisk (*) matches any string of characters.

If the two text strings are identical, the function returns a zero.

If the two text strings are not identical a result of -1 or 1 is returned. The determination of which is made in the following way:

> If the first text string is shorter than the second, but every character in the first text string is an exact match to the corresponding character in the second, a -1 is returned.
>
> If the second text string is shorter than the first, but every character in the second text string is an exact match to the corresponding character in the first, a 1 is returned.
>
> If the text strings do not match, and the first character within the first text string not to match the corresponding character in the second has a lower intrinsic value than the corresponding character in the second, a -1 is returned.
>
> If the text strings do not match, and the first character within the first text string not to match the corresponding character in the second has a higher intrinsic value than the corresponding character in the second, a 1 is returned.

_@STRCMP is a value generating text function.

FORMAT

_@STRCMP(string_1, string_2)

Where: **string_1** is the first text string literal or the name of the first text or string data field to be tested
string_2 is the second text string literal or the name of the second text or string data field to be tested

_@STRICMP is used to compare two text strings just as *strcasecmp* is used in the *C* programming language. The two text strings are passed as arguments to the function. They are compared on a character by character basis without regard to case. That is the two strings "HELLO" and "hello" would be considered identical.

Wild cards are supported in the comparison. A question mark (?) matches any character, and an asterisk (*) matches any string of characters.

If the two text strings are identical, the function returns a zero.

If the two text strings are not identical a result of -1 or 1 is returned. The determination of which is made in the following way:

> If the first text string is shorter than the second, but every character in the first text string is an exact match to the corresponding character in the second, a -1 is returned.
>
> If the second text string is shorter than the first, but every character in the second text string is an exact match to the corresponding character in the first, a 1 is returned.
>
> If the text strings do not match, and the first character within the first text string not to match the corresponding character in the second has a lower intrinsic value after being converted to lower case than the corresponding character in the second after being converted to lower case, a -1 is returned.
>
> If the text strings do not match, and the first character within the first text string not to match the corresponding character in the second has a higher intrinsic value after being converted to lower case than the corresponding character in the second after being converted to lower case, a 1 is returned.

_@STRICMP is a value generating text function.

FORMAT

_@STRICMP(string_1, string_2)

Where: **string_1** is the first text literal or the name
of the first text or string data field to be tested
string_2 is the second text literal or the name
of the second text/string data field to be tested

_@FIND is used to find if and where one text string is contained within the other. The first text string is searched for the first occurence of the second text string within it. If the second text string is found within the first text string, the location of the first occurence is returned. If not a zero is returned.

A one returned indicates the occurence of the second text string within the first text string begins with the *first* character of the first text string.

The comparison is case-sensitive.

ie: “Politics” would *not* match “politics”

If the text string “Hello!hello!hello!Hello” were searched for the text string “hello”, the result would be 7.

_@FIND is a value generating text function.

FORMAT

_@FIND(string_1, string_2)

Where: **string_1** is the text literal or the name of the text or string data field to be searched
string_2 is the text literal or the name of the text or string data field to be found

EXAMPLES

Section 4.11 – Text Function – _@IFIND

_@IFIND is used to find if and where one text string is contained within the other. The first text string is searched for the first occurence of the second text string within it. If the second text string is found within the first text string, the location of the *first* occurence is returned. If not a zero is returned.

A one returned indicates the occurence of the second text string within the first text string begins with the first character of the first text string.

The comparison is *not* case-sensitive.

ie: “Politics” would match “politics”

If the string “Hello!hello!hello!Hello” were searched for the string “hello”, the result would be 1.

_@IFIND is a value generating text function.

FORMAT

_@IFIND(string_1, string_2)

Where: **string_1** is the text literal or the name
of the text or string data field to be searched
string_2 is the text literal or the name
of the text or string data field to be found

EXAMPLES

_@RFIND is used to find if and where one text string is contained within the other. The first text string is searched for the first occurence of the second text string within it. If the second text string is found within the first text string, the location of the *last* occurence is returned. If not a zero is returned.

A one returned indicates the occurence of the second text string within the first text string begins with the first character of the first text string.

The comparison is case-sensitive.

ie: "Politics" would *not* match "politics"

If the string "Hello!hello!hello!Hello" were searched for the string "hello", the result would be 13.

_@RFIND is a value generating text function.

FORMAT

_@RFIND(string_1, string_2)

Where: **string_1** is the text literal or the name
of the text or string data field to be searched
string_2 is the text literal or the name
of the text or string data field to be found

EXAMPLES

_@IRFIND is used to find if and where one text string is contained within the other. The first text string is searched for the first occurence of the second text string within it. If the second text string is found within the first text string, the location of the *last* occurence is returned. If not a zero is returned.

A one returned indicates the occurence of the second text string within the first text string begins with the first character of the first text string.

The comparison is *not* case-sensitive.

ie: “Politics” would match “politics”

If the string “Hello!hello!hello!Hello” were searched for the string “hello”, the result would be 19.

_@IRFIND is a value generating text function.

FORMAT

_@FIND(string_1, string_2)

Where: **string_1** is the text literal or the name
of the text or string data field to be searched
string_2 is the text literal or the name
of the text or string data field to be found

EXAMPLES

_@SUBSTR is used to extract a text string from within another text string. The original text string, the location within the text string to begin copying, and the number of characters to copy are passed as arguments to the function.

If a value of 1 is passed for the location to begin copying, copying begins with the first character. If a value of 2 is passed for the location to begin copying, copying begins with the second character. And, so on.

If the location to begin copying is identified as zero or exceeds the number of characters in the original text string, nothing is copied.

If the number of characters to copy exceeds the number of characters remaining in the original text string, only the remaining characters in the text string are copied.

_@SUBSTR is a text generating text function.

FORMAT

_@SUBSTR(string, start, length)

Where:	**string**	is the text literal or the name of the text or string data field from which to extract the substring
	start	is the location within the text string containing the first character to be copied to the new text string. A value of 1 indicates start with the first character in the original text string.
	length	is the number of characters to be copied from the original text string to the substring

EXAMPLES

Section 4.15 – Text Function – _@CAT

_@CAT is used to combine two or more text strings into a single text string. The individual text strings to be concatenated are passed to the function, as arguments, in the order to be processed. The text strings being concatenated are joined end-to-end. No blanks are inserted or removed from between the individual text strings.

Within the Display Template, _@CAT is not required to concatenate text strings. To concatenate two text strings, they need only be placed side by side with no intervening blanks.

_@CAT is a text generating text function.

FORMAT

_@CAT(string_1, string_2 [, string_3 [, . . .]])

Where: **string_1** is the first text literal or the name of the first text or string data field to be included
string_2 is the second text literal or the name of the second text or string data field to be included
string_3 is the third text literal or the name of the third text or string data field to be included
. . . continue in like manor to add text literals or text or string data fields

EXAMPLES

_@CAPS returns a copy of the text string passed as an argument, with all words capitalized.

_@CAPS is a text generating text function.

FORMAT

_@CAPS(string)

Where: **string** is the text literal or the name of the text or string data field to be capitalized

EXAMPLES

_@LOWER returns a copy of the text string passed as an argument, with all characters coverted to lower case.

_@LOWER is a text generating text function.

FORMAT

_@LOWER(string)

Where: **string** is the text literal or the name of the text or string data field to be converted to lower case

EXAMPLES

_@UPPER returns a copy of the text string passed as an argument, with all characters coverted to upper case.

_@UPPER is a text generating text function.

FORMAT

_@UPPER(string)

Where: **string** is the text literal or the name of the text or string data field to be converted to upper case

EXAMPLES

```
<!-- ##  Display the title in all uppercase  ## -->
          ` _@UPPER(  _ArticleTitle )`<br>
```

Section 5 – Operators

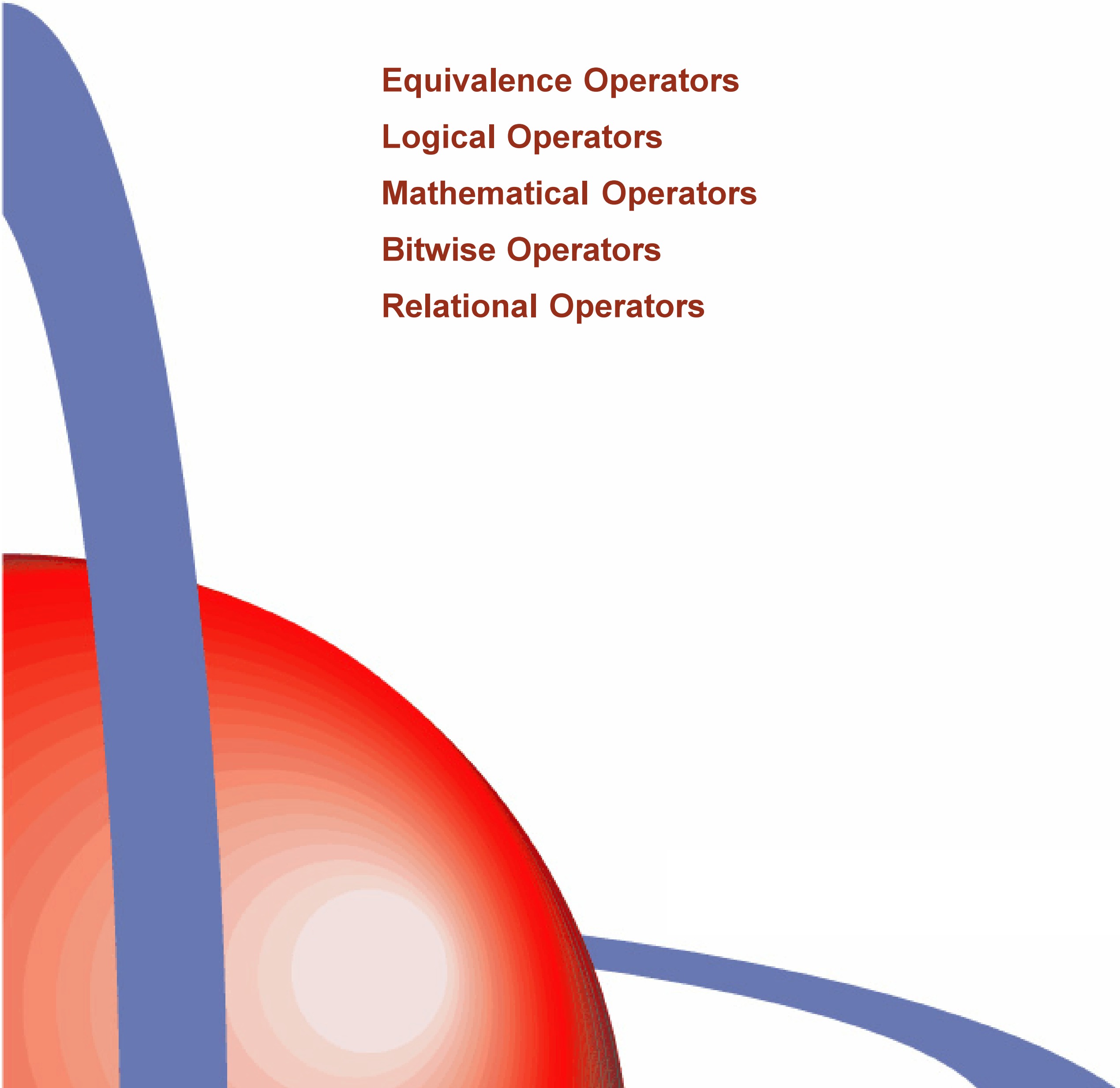
Equivalence Operators

Logical Operators

Mathematical Operators

Bitwise Operators

Relational Operators



Section 5.1 – Equivalence Operators

Operators are an integral part of any expression. What operators are supported determines the power of the software in processing expressions. Flexible Search supports a wide range of operators. Among them are equivalence operators, logical operators, mathematical operators, bitwise operators, and relational operators.

Equivalence operators are operators that compare two operands. Six types of equivalence operators are supported:

"=" or "=="
the binary operator *EQUAL TO*
returns 1 if operands are equal

"!=" or "<>"
the binary operator *NOT EQUAL TO*
returns 1 if operands are not equal

"<"
the binary operator *LESS THAN*
returns 1 if the first operand is
less than the second operand

">"
the binary operator *GREATER THAN*
returns 1 if the first operand is
greater than the second operand

"!<" or ">="
the binary operator *NOT LESS THAN*
returns 1 if the first operand is
not less than the second operand

"!>" or "<="
the binary operator *NOT GREATER THAN*
returns 1 if the first operand is
not greater than the second operand

Logical operators are operators that operate on the operands as logical values of *FALSE* or *TRUE*, that is zero or non-zero. Three types of logical operators are supported:

“!” the unary negation operator *NOT*
returns 0 for non-zero operand
and 1 for operand of 0

“||” the binary operator *INCLUSIVE OR*
returns 1 if either operand is non-zero
otherwise it returns 0

"&&" the binary operator *RESTRICTIVE AND*
returns 1 if both operands are non-zero
otherwise it returns 0

Section 5.3 – Mathematical Operators

Mathematical operators are operators that operate on the operands as mathematical values to produce mathematical results. Six types of mathematical operators are supported:

“-” the unary negation operator minus
returns the operand multiplied by minus one

“-” the binary operator for subtraction
returns the 1st operand minus the 2nd operand

"+" the binary operator for addition
returns the 1st operand plus the 2nd operand

“*” the binary operator for multiplication
returns the 1st operand
multiplied by the 2nd operand

"/" the binary operator for integer division
returns the result of the 1st operand divided by
the 2nd operand and discards the remainder

“%” the binary operator for modular division
returns the remainder of the 1st operand divided
by the 2nd operand and discards the result

Bitwise operators are operators that operate on the operands as collections of bits. Six types of bitwise operators are supported:

“~” the unary negation operator *NOT*
returns the operand with all bits inverted

“<<” the binary operator *LEFT SHIFT*
returns the 1st operand with the individual bits shifted, not rotated, to the left by the amount in the 2nd operand. Bits shifted off are lost. Bits shifted in from the right are 0.

“>>" the binary operator *RIGHT SHIFT*
returns the 1st operand with the individual bits shifted, not rotated, to the right by the amount in the 2nd operand. Bits shifted off are lost. Bits shifted in from the left are 0.

“&” the binary operator *RESTRICTIVE AND*
returns for each pair of corresponding bits in the 1st and 2nd operand, a 1 if both bits are 1 and a 0 otherwise.

“|" the binary operator *INCLUSIVE OR*
returns for each pair of corresponding bits in the 1st and 2nd operand, a 1 if either bits is 1 and a 0 otherwise.

“^” the binary operator inclusive *OR*
returns for each pair of corresponding bits in the 1st and 2nd operand, a 0 if the two bits are identical and a 1 otherwise.

Section 5.5 – Relational Operators

Relational operators are operators compare the two operands and return one of the operands as a result. Three types of relational operators are supported:

“&<” the binary operator *MAX*
returns the larger of the two operands.
A list of values seperated by the *MAX*
operator returns the largest value in the list.

“&>” the binary operator *MIN*
returns the smaller of the two operands.
A list of values seperated by the *MIN*
operator returns the smallest value in
the list.

“&|” the binary operator *FIRST*
returns the 1st operand if the value of the
1st operand is other than 0, and the 2nd
operand otherwise.
A list of values seperated by the *FIRST*
operator returns the first non-zero value
in the list.

Section 6 – FormatString Statement

Numeric Formats

Date-Time Formats

Geo-Spatial Formats

Section 6.1 – Numeric Formats

The *FormatString* statement can be used to display numeric data in any of five supported numeric formats:

SIGNED — used to display numeric data which may be either positive or negative. Positive values are displayed unsigned while negative values are displayed with a leading minus sign.

UNSIGNED — used to display numeric data that can only be positive.

HEX — used to display numeric data as a hexadecimal value.

FIXED POINT — used to display fixed point numeric data.

CURRENCY — used to display numeric data as a currency value.

The entire *FormatString* statement, including parenthesis and arguments, must be enclosed within a pair of grave accents.

FORMATS

FormatString(value, "SIGNED")

Where: **value** is the numeric data field, variable, or expression to be formatted

FormatString(value, "UNSIGNED" [, size])

Where: **value** is the numeric data field, variable, or expression to be formatted
size is the number of digits to be displayed

FormatString(value, "HEX")

Where: **value** is the numeric data field, variable, or expression to be formatted

FormatString(value, "FIXED POINT", size [, count])

Where: **value** is the numeric data field, variable, or expression to be formatted
size is the number of decimal places implied within the value to be displayed
count is the number of digits to be displayed to the right of the decimal point

FormatString(value, "CURRENCY", units[, count])

Where: **value** is the numeric data field, variable, or expression to be formatted
units is the modular value used to determine what portion of the value to be displayed on either side of the decimal point. For a US dollar or mexican peso, it is 100 if cents or centavos are the storage unit.
count is the number of digits to be displayed to the right of the decimal point

EXAMPLES

Section 6.2 – Date-Time Formats

The *FormatString* statement can be used to display numeric data in any of ten supported *Date-Time Formats*:

DATE — used to display a date in conventional month-day-year format. ie: Jan 1, 2000.

TIME — used to display time in conventional hour-minute-am/pm format. ie: 11:00 pm.

DATE TIME — used to display date-time in conventional month-day-year-hour-minute-am/pm format. ie: Jan 1, 2000 11:00 pm.

TIME DATE — used to display date-time in conventional hour-minute-am/pm-month-day-year format. ie: 11:00 pm Jan 1, 2000.

12HR TIME — used to display time in conventional hour-minute-am/pm format. ie: 11:00 pm.

24HR TIME — used to display time in 24-hour-minute format. ie: 23:00.

12HR TIME DATE — used to display date-time in conventional hour-minute-am/pm-month-day-year format. ie: 11:00 pm Jan 1, 2000.

24HR TIME DATE — used to display date-time in 24-hour-minute-am/pm-month-day-year format. ie: 23:00 Jan 1, 2000.

DATE 12HR TIME — used to display date-time in conventional month-day-year-hour-minute-am/pm format. ie: Jan 1, 2000 11:00 pm.

DATE 24HR TIME — used to display date-time in month-day-year-24-hour-minute format.
ie: Jan 1, 2000 23:00.

The entire *FormatString* statement, including parenthesis and arguments, must be enclosed within a pair of grave accents.

FORMATS

FormatString(data, "DATE")

Where: **data** is the date-time variable, expression or numeric data field to be formatted

FormatString(data, "TIME")

Where: **data** is the date-time variable, expression or numeric data field to be formatted

FormatString(data, "DATE TIME")

Where: **data** is the date-time variable, expression or numeric data field to be formatted

FormatString(data, "TIME DATE")

Where: **data** is the date-time variable, expression or numeric data field to be formatted

FormatString(data, "12 HR TIME")

Where: **data** is the date-time variable, expression or numeric data field to be formatted

FormatString(data, "24 HR TIME")

Where: **data** is the date-time variable, expression or numeric data field to be formatted

FormatString(data, "12 HR TIME DATE")

Where: **data** is the date-time variable, expression or numeric data field to be formatted

FormatString(data, “24 HR TIME DATE”)

Where: **data** is the date-time variable, expression or numeric data field to be formatted

FormatString(data, “DATE 12 HR TIME”)

Where: **data** is the date-time variable, expression or numeric data field to be formatted

FormatString(data, “DATE 24 HR TIME”)

Where: **data** is the date-time variable, expression or numeric data field to be formatted

EXAMPLES

The *FormatString* statement can be used to display numeric data in any of five supported *Geo-spatial Formats*:

GEO DEGREES used to display a latitude or longitude in degrees with a possible decimal fraction.

GEO MINUTES used to display a latitude or longitude in degrees and minutes.

GEO SECONDS used to display a latitude or longitude in degrees, minutes, and seconds.

MILES used to display a distance used in geospatial searching, filtering or sorting in miles.

KILOS used to display a distance used in geospatial searching, filtering or sorting in kilometers.

The entire *FormatString* statement, including parenthesis and arguments, must be enclosed within a pair of grave accents.

FORMATS

FormatString(coord, "GEO DEGREES" [, count])

Where: **coord** is the longitude or latitude in internal geo-spatial coordinate units

count is the number of digits to be displayed to the right of the decimal point

FormatString(coord, "GEO MINUTES" [, size])

Where: **coord** is the longitude or latitude in internal geo-spatial coordinate units

FormatString(coord, "GEO SECONDS")

Where: **coord** is the longitude or latitude in internal geo-spatial coordinate units

FormatString(dist, "MILES" [, count])

Where: **dist** is the numeric data field, variable, or expression to be formatted
count is the number of digits to be displayed to the right of the decimal point

FormatString(dist, "KILOS" [, count])

Where: **dist** is the numeric data field, variable, or expression to be formatted
count is the number of digits to be displayed to the right of the decimal point

EXAMPLES

Section 7 – System Values

Basic Processing Values

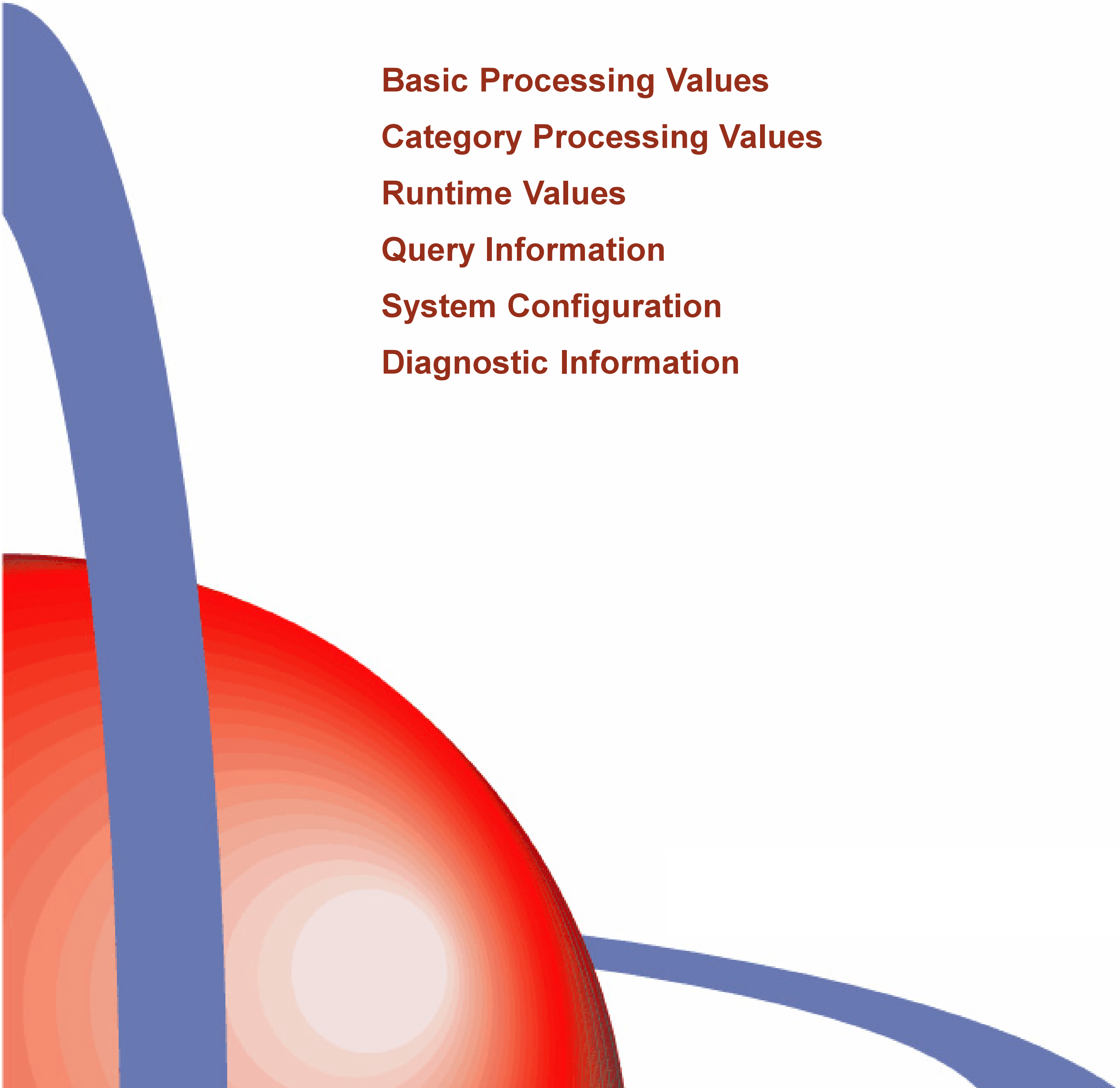
Category Processing Values

Runtime Values

Query Information

System Configuration

Diagnostic Information



Section 7.1 – Basic Processing Values

System values are values used to define necessary constants, control processing, return information about the system or log activity through time. Some system values can be set within the *logic template,* however, most system values are read-only.

Each system value is identified by a name in full caps preceded by two underscores.

BASIC PROCESSING VALUES

__REQUEST_CNT

Number of items requested to be displayed on each page of the search results.

__TOTAL

The total number of items found matching the search criteria.

__COUNT

Number of items to be displayed on the current page of the search results.

__FIRST

The ordinal number (1 thru __TOTAL) of the first item to be displayed on the current page of the search results.

__PRIOR

The ordinal number (1 thru __TOTAL) of the first item to be displayed on the previous page of the search results.

__NEXT

The ordinal number (1 thru __TOTAL) of the first item to be displayed on the next page of the search results.

__LAST

The ordinal number (1 thru __TOTAL) of the last item to be displayed on the current page of the search results.

__ORDINAL

The ordinal number (1 thru __COUNT) of the current item being processed.

__ODD

TRUE (1) if the value of __ORDINAL is odd. FALSE (0) if it is even.

CATEGORY PROCESSING VALUES

__DISTRIBUTED_TOTAL
: Total number of individual items found, among all the categories, matching the search criteria.

__CATEGORY_TOTAL
: Total number of categories found having items matching the search criteria.

__FIRST_CATEGORY
: Ordinal number (1 thru __CATEGORY_TOTAL) of the first category to be displayed.

__CATEGORY_COUNT
: Number of categories being displayed on the current page of the results.

__RUNNING_COUNT
: Total number of items displayed, up to this point, from all categories.

__CATEGORY
: Keyword (or base value if mapped to a range) which defines the category.

__CATEGORY_DELTA
: For category lists mapped to a range, the size of an individual subrange.

__CATEGORY_SIZE
: Number of items found within the category matching the search criteria.

RUNTIME VALUES

__CURRENT_DATE
Current local date.

__CURRENT_TIME
Current local time.

__CURRENT_DT
Current local date and time.

__TIME_STAMP
Current date and time in timestamp format.

__RANDOM_NUMBER
A simulated random number for use in generating uncacheable ad URLs among other things.

__THREAD
Thread performing the request.

__KEYWORD_SET
A search request of a keyword index has been made.

__RANGE_SET
A search request of a range index has been made.

__TIMELINE_SET
A search request of a timeline index has been made.

__SPATIAL_SET
A search request of a spatial index has been made.

__GEO_SPATIAL_SET
A search request of a geo-spatial index has been made.

__PACKAGE_SET
A search request of a package index has been made.

__HAS_INPUT
A search request of an index has been made.

__SEARCH_PERFORMED
A search was performed against at least one index.

__EARLY_EXIT
TRUE (1) if the recently exited loop did not run to completion. FALSE (0) if it did run to completion.

__PARMS
URL-encoded name-value pairs for all variables whose names do not begin with an underscore.

QUERY INFORMATION

__CLIENT_IP

IP address of the client machine sending the query.

__CLIENT_PORT

Port address of the client machine sending the query.

__REFERING_ITEM

Refering web page as extracted from the HTTP header.

__USER_AGENT

User agent as extracted from the HTTP header.

__LANGUAGE

Language identified by the HTTP header.

__ACCESS_MODE

The format of the query received:

1	Simple socket request
2	HTTP get operation
3	HTTP post operation
4	Simple echo test

__QUERY

The query received from the client.

__STATUS

The status of the current request.

__TRANSFER_SIZE

Number of bytes sent in response to query.

SYSTEM CONFIGURATION

__SERVER

The IP address or DNS name of the server machine processing the template.

__PORT

The port being used by the server to service insecure queries.

__SSL_PORT

The port being used by the server to service secure queries using the SSL security protocol.

__LOGIC

Resource ID of the Logic Template used to process the request.

__DISPLAY

Resource ID of the Display Template used to display the results.

__LOG

Resource ID of the Logging Template used to generate the transaction log.

DIAGNOSTIC INFORMATION

__START_DT
Date and time at which application was launched.

__MEM_ALLOCATED
Number of 8K blocks allocated from the system for use in loading indexes and processing requests.

__MEM_AVAILABLE
Number of allocated 8K blocks not currently in use..

__MEM_PARTITIONED
Number of allocated 8K blocks partitioned into smaller allocation units.

__CONNECT_COUNT
Number of queries received since the status tabulations were last reset.

__DISCONNECT_COUNT
Number of queries experiencing which timed out during transmission since the status tabulations were last reset.

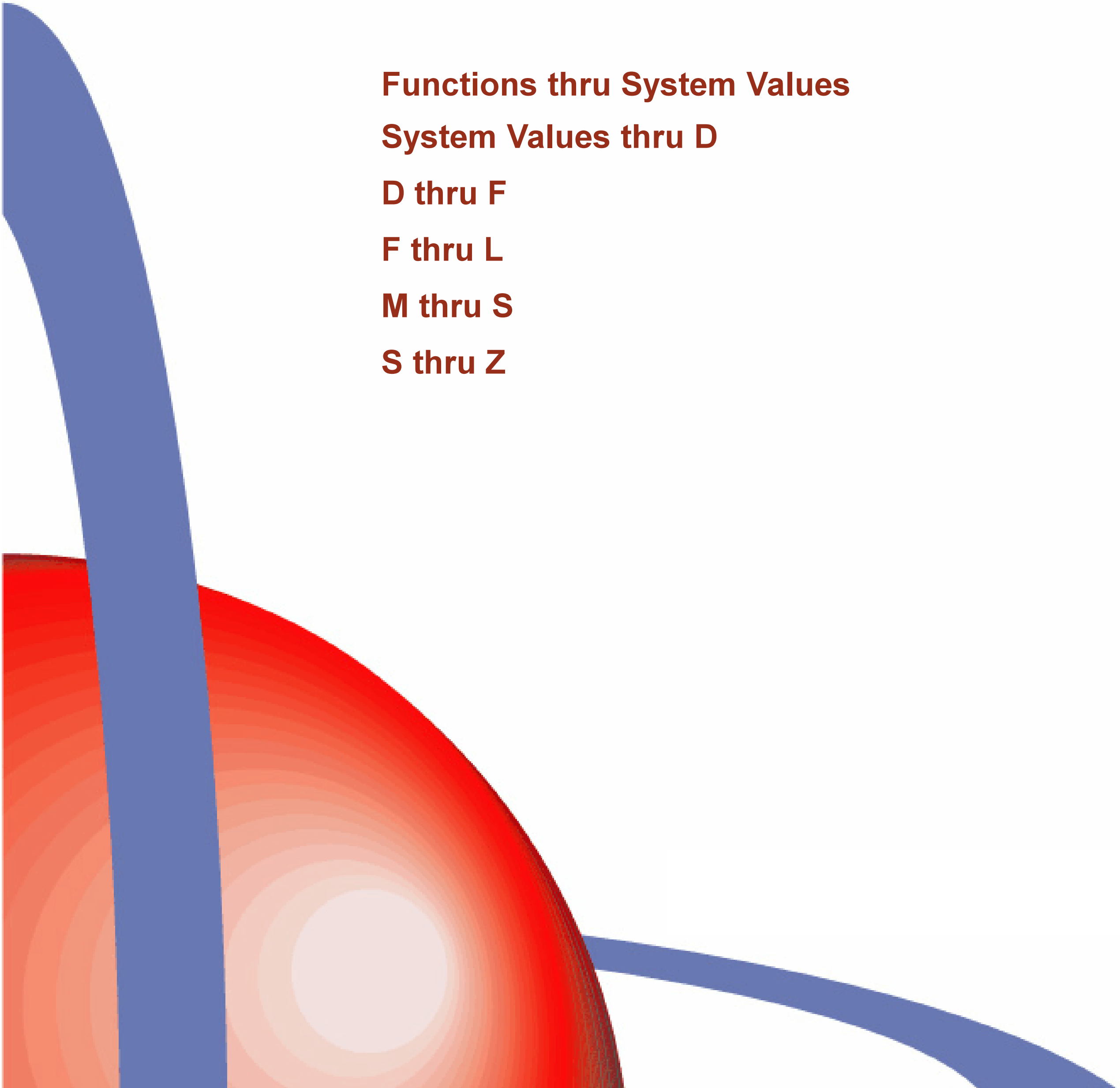
__SUCCESS_COUNT
Number of queries returning results since the status tabulations were last reset.

__FAILURE_COUNT
Number of queries returning no results since the status tabulations were last reset.

__ERROR_COUNT
Number of queries experiencing processing errors since the status tabulations were last reset.

__STATUS_RESET_DT
Date and time at which the tabulation values were last reset.

INDEX



Functions

System Values