Information Builders

Functions Reference

Version 7 Release 6.1 and Higher

DN3501785.0408

Contents

Preface

This documentation describes how to use DataMigrator-supplied functions to perform complex calculations and manipulate data in your procedures.

How This Manual Is Organized

This manual includes the following chapters:

	Chapter/Appendix	Contents
1	Functions Overview	Introduces functions and explains the different types of available functions.
2	Character Functions	Describes character functions which manipulate alphanumeric fields and character strings.
3	Variable Length Character Functions	Describes variable-length character functions which manipulate alphanumeric fields and character strings.
4	Data Source and Decoding Functions	Describes data source and decoding functions which search for data source records, retrieve data source records or values, and assign values based on the value of an input field.
5	Date and Date-Time Functions	Describes date and time functions which manipulate date and time values.
6	Format Conversion Functions	Describes format conversion functions which convert fields from one format to another.
7	Numeric Functions	Describes numeric functions which perform calculations on numeric constants and fields.
8	System Functions	Describes system functions which call the operating system to obtain information about the operating environment or to use a system service.

	Chapter/Appendix	Contents
9	SQL Character Functions	Describes SQL character functions which manipulate alphanumeric fields and character strings.
10	SQL Date and Time Functions	Describes SQL date and time functions which manipulate date and time values.
11	SQL Data Type Conversion Functions	Describes SQL format conversion functions which convert fields from one format to another.
12	SQL Numeric Functions	Describes SQL numeric functions which perform calculations on numeric constants and fields.
13	SQL Miscellaneous Functions	Describes miscellaneous SQL functions which perform conversions, tests and manipulations.
14	SQL Operators	Describes SQL operators which used to evaluate expressions.

Documentation Conventions

The following table lists and describes the conventions that apply in this manual.

Convention	Description
`THIS TYPEFACE` or *`this typeface`*	Denotes syntax that you must enter exactly as shown.
this typeface	Represents a placeholder (or variable), a cross-reference, or an important term.
underscore	Indicates a default setting.
this typeface	Highlights a file name or command. It may also indicate a button, menu item, or dialog box option you can click or select.
Key + Key	Indicates keys that you must press simultaneously.
`{ }`	Indicates two or three choices; type one of them, not the braces.

Convention	Description
[ ]	Indicates a group of optional parameters. None is required, but you may select one of them. Type only the parameter in the brackets, not the brackets.
|	Separates mutually exclusive choices in syntax. Type one of them, not the symbol.
. . .	Indicates that you can enter a parameter multiple times. Type only the parameter, not the ellipsis points (...).
. . .	Indicates that there are (or could be) intervening or additional commands.

Related Publications

To view a current listing of our publications and to place an order, visit our Technical Documentation Library, *http://documentation.informationbuilders.com*. You can also contact the Publications Order Department at (800) 969-4636.

Customer Support

Do you have any questions about this product?

Join the Focal Point community. Focal Point is our online developer center and more than a message board. It is an interactive network of more than 3,000 developers from almost every profession and industry, collaborating on solutions and sharing tips and techniques, *http://forums.informationbuilders.com/eve/forums*.

You can also access support services electronically, 24 hours a day, with InfoResponse Online. InfoResponse Online is accessible through our World Wide Web site, *http://www.informationbuilders.com*. It connects you to the tracking system and known-problem database at the Information Builders support center. Registered users can open, update, and view the status of cases in the tracking system and read descriptions of reported software issues. New users can register immediately for this service. The technical support section of www.informationbuilders.com also provides usage techniques, diagnostic tips, and answers to frequently asked questions.

Call Information Builders Customer Support Service (CSS) at (800) 736-6130 or (212) 736-6130. Customer Support Consultants are available Monday through Friday between 8:00 a.m. and 8:00 p.m. EST to address all your WebFOCUS Managed Reporting questions. Information Builders consultants can also give you general guidance regarding product capabilities and documentation. Please be ready to provide your six-digit site code number (xxxx.xx) when you call.

To learn about the full range of available support services, ask your Information Builders representative about InfoResponse Online, or call (800) 969-INFO.

Information You Should Have

To help our consultants answer your questions effectively, be prepared to provide the following information when you call:

- Your six-digit site code (xxxx.xx).
- Your iWay Software configuration:
  - The iWay Software version and release. You can find your server version and release using the Version option in the Web Console. (Note: the MVS and VM servers do not use the Web Console.)
  - The communications protocol (for example, TCP/IP or LU6.2), including vendor and release.
- The stored procedure (preferably with line numbers) or SQL statements being used in server access.
- The database server release level.
- The database name and release level.
- The Master File and Access File.
- The exact nature of the problem:
  - Are the results or the format incorrect? Are the text or calculations missing or misplaced?
  - The error message and return code, if applicable.
  - Is this related to any other problem?
- Has the procedure or query ever worked in its present form? Has it been changed recently? How often does the problem occur?
- What release of the operating system are you using? Has it, your security system, communications protocol, or front-end software changed?

- Is this problem reproducible? If so, how?
- Have you tried to reproduce your problem in the simplest form possible? For example, if you are having problems joining two data sources, have you tried executing a query containing just the code to access the data source?
- Do you have a trace file?
- How is the problem affecting your business? Is it halting development or production? Do you just have questions about functionality or documentation?

User Feedback

In an effort to produce effective documentation, the Documentation Services staff welcomes your opinions regarding this manual. Please use the Reader Comments form at the end of this manual to communicate suggestions for improving this publication or to alert us to corrections. You can also use the Documentation Feedback form on our Web site, *http://documentation.informationbuilders.com/feedback.asp*.

Thank you, in advance, for your comments.

iWay Software Training and Professional Services

Interested in training? Our Education Department offers a wide variety of training courses for iWay Software and other Information Builders products.

For information on course descriptions, locations, and dates, or to register for classes, visit our World Wide Web site, *http://www.iwaysoftware.com*, or call (800) 969-INFO to speak to an Education Representative.

Interested in technical assistance for your implementation? Our Professional Services department provides expert design, systems architecture, implementation, and project management services for all your business integration projects. For information, visit our World Wide Web site, *http://www.iwaysoftware.com*.

1 Functions Overview

Functions provide a convenient way to perform certain calculations and manipulations. They operate on one or more arguments and return a single value that is assigned to an *output_format*. The returned value can be stored in a field, assigned to a Dialogue Manager variable, used in an expression or other processing, or used in a selection or validation test. These functions can be used in source and target objects.

Topics:

Function Arguments

All function arguments except the last one are *input arguments*. The formats for these argument are described with each function. Unless specified, every input argument can be provided as one of the following:

- A literal (that is, a number for numeric formats or a character string enclosed in single quotation marks for alphanumeric formats).
- A field of the correct format.
- A variable assigned by a Dialogue Manager command.
- An expression result evaluated in the correct format.

The *output_format* is the last function argument. With few exceptions, it is a required argument whose only goal is to provide *a format* for the function's output. It is *not* a field to put the result in. The format can be provided as either:

- A character string enclosed in single quotation marks.
- A field name whose format is to be used.

This field is the one to which the result of the expression evaluation is assigned. If the output_format is alphanumeric, its size should be large enough to fit the function output and avoid truncation; excessive size causes the output to be padded with blanks.

Function Categories

Functions are grouped into the following areas:

- *Character Functions*
- *Variable Length Character Functions*
- *Data Source and Decoding Functions*
- *Date and Date-Time Functions*
  - *Standard Date Functions*
  - *Date-Time Functions*
  - *Legacy Date Functions*
- *Format Conversion Functions*
- *Numeric Functions*
- *System Functions*

Character Chart for ASCII and EBCDIC

This chart shows the primary printable characters in the ASCII and EBCDIC character sets and their decimal equivalents. Extended ASCII codes (above 127) are not included.

Decimal	ASCII		EBCDIC	
33	!	exclamation point		
34	"	quotation mark		
35	#	number sign		
36	$	dollar sign		
37	%	percent		
38	&	ampersand		
39	'	apostrophe		
40	(	left parenthesis		
41	)	right parenthesis		
42	*	asterisk		
43	+	plus sign		
44	,	comma		
45	-	hyphen		
46	.	period		
47	/	slash		
48	0	0		
49	1	1		
50	2	2		
51	3	3		
52	4	4		

Decimal	ASCII		EBCDIC	
53	5	5		
54	6	6		
55	7	7		
56	8	8		
57	9	9		
58	:	colon		
59	;	semicolon		
60	<	less-than sign		
61	=	equal sign		
62	>	greater-than sign		
63	?	question mark		
64	@	at sign		
65	A	A		
66	B	B		
67	C	C		
68	D	D		
69	E	E		
70	F	F		
71	G	G		
72	H	H		
73	I	I		
74	J	J	¢	cent sign

Decimal	ASCII		EBCDIC	
75	K	K	.	period
76	L	L	<	less-than sign
77	M	M	(	left parenthesis
78	N	N	+	plus sign
79	O	O	|	logical or
80	P	P	&	ampersand
81	Q	Q		
82	R	R		
83	S	S		
84	T	T		
85	U	U		
86	V	V		
87	W	W		
88	X	X		
89	Y	Y		
90	Z	Z	!	exclamation point
91	[	opening bracket	$	dollar sign
92	\	back slant	*	asterisk
93	]	closing bracket	)	right parenthesis
94	^	caret	;	semicolon
95	_	underscore	¬	logical not
96	`	grave accent	-	hyphen

Decimal	ASCII		EBCDIC	
97	a	a	/	slash
98	b	b		
99	c	c		
100	d	d		
101	e	e		
102	f	f		
103	g	g		
104	h	h		
105	i	i		
106	j	j		
107	k	k	,	comma
108	l	l	%	percent
109	m	m	_	underscore
110	n	n	>	greater-than sign
111	o	o	?	question mark
112	p	p		
113	q	q		
114	r	r		
115	s	s		
116	t	t		
117	u	u		
118	v	v		

Decimal	ASCII		EBCDIC	
119	w	w		
120	x	x		
121	y	y		
122	z	z	:	colon
123	{	opening brace	#	number sign
124	|	vertical line	@	at sign
125	}	closing brace	'	apostrophe
126	~	tilde	=	equal sign
127			"	quotation mark
129			a	a
130			b	b
131			c	c
132			d	d
133			e	e
134			f	f
135			g	g
136			h	h
137			i	i
145			j	j
146			k	k
147			l	l
148			m	m

Decimal	ASCII		EBCDIC	
149			n	n
150			o	o
151			p	p
152			q	q
153			r	r
162			s	s
163			t	t
164			u	u
165			v	v
166			w	w
167			x	x
168			y	y
169			z	z
185			`	grave accent
193			A	A
194			B	B
195			C	C
196			D	D
197			E	E
198			F	F
199			G	G
200			H	H

Decimal	ASCII		EBCDIC	
201			I	I
209			J	J
210			K	K
211			L	L
212			M	M
213			N	N
214			O	O
215			P	P
216			Q	Q
217			R	R
226			S	S
227			T	T
228			U	U
229			V	V
230			W	W
231			X	X
232			Y	Y
233			Z	Z
240			0	0
241			1	1
242			2	2
243			3	3

Decimal	ASCII		EBCDIC	
244			4	4
245			5	5
246			6	6
247			7	7
248			8	8
249			9	9

2 Character Functions

Character functions manipulate alphanumeric fields and character strings.

Topics:

- LCWORD: Converting a Character String to Mixed Case

ARGLEN: Measuring the Length of a Character String

How to:

Measure the Length of a Character String

The ARGLEN function measures the length of a character string within a field, excluding trailing spaces. The field format in a synonym specifies the full size of a field, including trailing spaces.

Syntax: How to Measure the Length of a Character String

```
ARGLEN(length, source_string, output_format)
```

where:

length

Integer

Is the number of characters in the source_string field.

source_string

Alphanumeric

Is a character string whose length, without trailing blanks, is to be returned.

output_format

Integer

Example: Measuring the Length of a Character String

ARGLEN determines the length of the character string in LAST_NAME and stores the result in a column with the format I3:

```
ARGLEN(15, LAST_NAME, 'I3')
```

For SMITH, the result is 5.

For BLACKWOOD, the result is 9.

BITSON: Determining If a Bit is On or Off

How to:

Determine If a Bit is On or Off

The BITSON function evaluates an individual bit within a character string to determine whether it is on or off. If the bit is on, BITSON returns a value of 1; if the bit is OFF, it returns a value of 0. This function is useful in interpreting multi-punch data, where each punch conveys an item of information.

Syntax: How to Determine If a Bit is On or Off

```
BITSON(bitnumber, source_string, output_format)
```

where:

bitnumber

Integer

Is the number of the bit to be evaluated, counted from the left-most bit in the character string.

source_string

Alphanumeric

Is the character string, in multiple 8-bit blocks.

output_format

Integer

Example: Determining If a Bit is On or Off

BITSON evaluates the 24th bit of LAST_NAME:

```
BITSON(24, LAST_NAME, 'I1')
```

For SMITH, the result is 1.

For CROSS, the result is 9.

BITVAL: Evaluating a Bit String as a Binary Integer

How to:

Evaluate a Bit String as a Binary Integer

The BITVAL function evaluates a string of bits within a character string. The bit string can be any group of bits within the character string and can cross byte and word boundaries. The function evaluates the subset of bits in the string as an integer value.

Syntax: How to Evaluate a Bit String as a Binary Integer

```
BITVAL(source_string, startbit, number, output_format)
```

where:

source_string

Alphanumeric

Is the character string to be evaluated.

startbit

Integer

Is the first bit number in the source_string, counting from the left-most bit. If this argument is less than or equal to 0, the function returns zero.

number

Integer

Is the number of bits in the subset of bits. If this argument is less than or equal to 0, the function returns a value of zero.

output_format

Integer

Example: Evaluating a Bit String as a Binary Integer

BITVAL evaluates the bits 12 through 20 of LAST_NAME and stores the result in a column with the format I5:

```
BITVAL(LAST_NAME, 12, 9, 'I5')
```

For SMITH, the result is 332.

For JONES, the result is 365.

BYTVAL: Translating a Character to a Decimal Value

How to:

Translate a Character to a Decimal Value

The BYTVAL function translates a character to the ASCII or EBCDIC decimal value that represents it, depending on the operating system.

Syntax: How to Translate a Character to a Decimal Value

```
BYTVAL(character, output_format)
```

where:

character

Alphanumeric

Is the character to be translated. If you supply more than one character, the function evaluates the first one.

output_format

Integer

Example: Translating a Character to a Decimal Value

BYTVAL translates the first character of LAST_NAME into its ASCII decimal value and and stores the result in a column with the format I3.

```
BYTVAL(LAST_NAME,'I3')
```

For SMITH, the result is 83.

For JONES the result is 74.

CHKFMT: Checking the Format of a Character String

How to:

Check the Format of a Character String

The CHKFMT function checks a character string for incorrect characters or character types. It compares each character string to a second string, called a mask, comparing each character in the first string to the corresponding character in the mask. If all characters in the character string match the characters or character types in the mask, CHKFMT returns the value 0. Otherwise, CHKFMT returns a value equal to the position of the first character in the character string not matching the mask.

If the mask is shorter than the character string, the function checks only the portion of the character string corresponding to the mask. For example, if you are using a four-character mask to test a nine-character string, only the first four characters are checked; the rest are returned as a no match, with CHKFMT giving the first non-matching position as the result.

Syntax: How to Check the Format of a Character String

```
CHKFMT(numchar, source_string, 'mask', output_format)
```

where:

numchar

Integer

Is the number of characters being compared to the mask.

source_string

Alphanumeric

Is the character string to be checked.

'mask'

Alphanumeric

Is the mask, which contains the comparison characters enclosed in single quotation marks. Some characters in the mask are generic and represent character types. If a character in the string is compared to one of these characters and is the same type, it matches.

Generic characters are:

`A` is any letter between A and Z (uppercase or lowercase).

`9` is any digit between 0-9.

`X` is any letter between A-Z or any digit between 0-9.

`$` is any character.

Any other character in the mask represents only that character. For example, if the third character in the mask is B, the third character in the string must be B to match.

output_format

Integer

Example: Checking the Format of a Character String

CHKFMT examines EMP_ID for nine numeric characters starting with 11 and stores the result in a column with the format I3.

```
CHKFMT(9, EMP_ID, '119999999', 'I3')
```

For 071382660, the result is 1.

For 119265415, the result is 0.

For 23764317, the result is 2.

CTRAN: Translating One Character to Another

How to:

Translate One Character to Another

The CTRAN function translates a character within a character string to another character based on its decimal value. This function is especially useful for changing replacement characters to unavailable characters, or to characters that are difficult to input or unavailable on your keyboard. It can also be used for inputting characters that are difficult to enter when responding to a Dialogue Manager -PROMPT command, such as a comma or apostrophe. It eliminates the need to enclose entries in single quotation marks.

To use CTRAN, you need to know the decimal equivalent of the characters in internal machine representation. Note that the coding chart for conversion is platform dependent, hence the latter determines whether ASCII or EBCDIC coding is used. Printable EBCDIC or ASCII characters and their decimal equivalents are listed in *Character Chart for ASCII and EBCDIC* on page 17.

Syntax: How to Translate One Character to Another

```
CTRAN(length, source_string, decimal, decvalue, output_format)
```

where:

length

Integer

Is the number of characters in the source_string field.

source_string

Alphanumeric

Is the character string to be translated.

decimal

Integer

Is the ASCII or EBCDIC decimal value of the character to be translated.

decvalue

Integer

Is the ASCII or EBCDIC decimal value of the character to be used as a substitute for decimal.

output_format

Alphanumeric

Example: **Translating Spaces to Underscores on an ASCII Platform**

CTRAN translates the spaces in ADDRESS_LN3 (ASCII decimal value of 32) to underscores (ASCII decimal value of 95) and stores the result in a column with the format A20.

```
CTRAN(20, PRODNAME, 32, 95, 'A20')
```

For RUTHERFORD NJ 07073, the result is RUTHERFORD_NJ_07073_.

For NEW YORK NY 10039, the result is NEW_YORK_NY_10039___.

CTRFLD: Centering a Character String

How to:

Center a Character String

The CTRFLD function centers a character string within a field. The number of leading spaces is equal to or one less than the number of trailing spaces.

CTRFLD is useful for centering the contents of a field and its report column, or a heading that consists only of an embedded field. HEADING CENTER centers each field value including trailing spaces. To center the field value without the trailing spaces, first center the value within the field using CTRFLD.

Syntax: How to Center a Character String

```
CTRFLD(source_string, length, output_format)
```

where:

source_string

Alphanumeric

Is the alphanumeric constant enclosed in single quotation marks, or a field or variable that contains the character string.

length

Integer

Is the number of characters in the source_string and the output_format. This argument must be greater than 0. A length less than 0 can cause unpredictable results.

output_format

Alphanumeric

Example: Centering a Column

CTRFLD centers LAST_NAME and stores the result in a column with the format A12:

```
CTRFLD(LAST_NAME, 12, 'A12')
```

DCTRAN: Translating A Single-Byte or Double-Byte Character to Another

How to:

Translate a Single-Byte or Double-Byte Character to Another

The DCTRAN function translates a single-byte or double-byte character within a character string to another character based on its decimal value. To use DCTRAN, you need to know the decimal equivalent of the characters in internal machine representation.

To use DCTRAN, you need to know the decimal equivalent of the characters in internal machine representation.

Syntax: How to Translate a Single-Byte or Double-Byte Character to Another

```
DCTRAN(length, source_string, inhexchar, outhexchar, output_format)
```

where:

length

Double

Is the number of characters in the source_string field.

source_string

Alphanumeric

Is the character string to be translated.

decimal

Double

Is the ASCII or EBCDIC decimal value of the character to be translated.

decvalue

Double

Is the ASCII or EBCDIC decimal value of the character to be used as a substitute for *inhexchar*.

output_format

Alphanumeric

Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.

Example: Using DCTRAN to Translate Double-Byte Characters

In the following:

DCTRAN(8, 'A゗A本B語', 177, 70, A8)

For A゗A本B語, the result is AFA本B語.

DEDIT: Extracting or Adding Characters

How to:

Extract or Add DBCS or SBCS Characters

If your configuration uses a DBCS code page, you can use the DEDIT function to extract characters form or add characters to a string.

DEDIT works by comparing the characters in a mask to the characters in a source field. When it encounters a nine (9) in the mask, DEDIT copies the corresponding character from the source field to the new field. When it encounters a dollar sign ($) in the mask, DEDIT ignores the corresponding character in the source field. When it encounters any other character in the mask, DEDIT copies that character to the corresponding position in the new field.

Syntax: How to Extract or Add DBCS or SBCS Characters

```
DEDIT(inlength, source_string, mask_length, mask, outfield)
```

where:

inlength

Integer

Is the number of *bytes* in *source_string*. The string can have a mixture of DBCS and SBCS characters; therefore, the number of bytes represents the maximum number of characters possible in the source string.

source_string

Alphanumeric

Is the string to edit enclosed in single quotation marks, or the field containing the string.

mask_length

Integer

Is the number of *characters* in mask.

mask

Alphanumeric

Is the string of mask characters.

Each nine (9) in the mask causes the corresponding character from the source field to be copied to the new field.

Each dollar sign ($) in the mask causes the corresponding character in the source field to be ignored.

Any other character in the mask is copied to the new field.

outfield

Alphanumeric

Is the field to which the result is returned, or the format of the output value enclosed in single quotation marks.

Example: **Adding and Extracting DBCS Characters**

The following example copies alternate characters from the source string to the new field, starting with the first character in the source string, and then adds several new characters at the end of the extracted string:

```
DEDIT( 15, 'あaいiうuえeおo', 16, '9$9$9$9$9$-かきくけこ', 'A30')
The result is あいうえお-かきくけこ.
```

The following example copies alternate characters from the source string to the new field, starting with the second character in the source string, and then adds several new characters at the end of the extracted string:

```
DEDIT( 15, 'あaいiうuえeおo', 16, '$9$9$9$9$9-ABCDE', 'A20')
The result is aiueo-ABCDE.
```

DSTRIP: Removing a Single-Byte or Double-Byte Character From a String

How to:

Remove a Single-Byte or Double-Byte Character From a String

The DSTRIP function removes all occurrences of a specific single-byte or double-byte character from a string. The resulting character string has the same length as the original string but is padded on the right with spaces.

Syntax: How to Remove a Single-Byte or Double-Byte Character From a String

```
DSTRIP(length, source_string, char, output_format)
```

where:

length

Double

Is the number of characters in *source_string* and *outfield*.

source_string

Alphanumeric

Is the string from which the character will be removed.

char

Alphanumeric

Is the character to be removed from the string. If more than one character is provided, the left-most character will be used as the strip character.

Note: To remove single quotation marks, use two consecutive quotation marks. You must then enclose this character combination in single quotation marks.

output_format

Alphanumeric

Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.

Example: Removing a Double-Byte Character From a String

In the following:

DSTRIP(9, 'A日A本B語', '日', A9)

For A日A本B語, the result is AA本B語.

DSUBSTR: Extracting a Substring

How to:

Extract a Substring

If your configuration uses a DBCS code page, you can use the DSUBSTR function to extract a substring based on its length and position in the parent string.

Syntax: How to Extract a Substring

`DSUBSTR(inlength, parent, start, end, sublength, outfield)`

where:

inlength

Integer

Is the length of the parent string in *bytes*, or a field that contains the length. The string can have a mixture of DBCS and SBCS characters; therefore, the number of bytes represents the maximum number of characters possible in the parent string.

parent

Alphanumeric

Is the parent string enclosed in single quotation marks, or the field containing the parent string.

start

Integer

Is the starting position (in number of *characters*) of the substring in the parent string. If this argument is less than one or greater than *end*, the function returns spaces.

end

Integer

Is the ending position (in number of *characters*) of the substring. If this argument is less than *start* or greater than *inlength*, the function returns spaces.

sublength

Integer

Is the length of the substring in characters (normally *end* - *start* + 1). If *sublength* is longer than *end* - *start* +1, the substring is padded with trailing spaces. If it is shorter, the substring is truncated. This value should be the declared length of *outfield*. Only *sublength* characters will be processed.

outfield

Alphanumeric

Is the field to which the result is returned, or the format of the output value enclosed in single quotation marks.

Example: Extracting a Substring

The following example extracts the 3-character substring in positions 4 through 6 from a 15-byte string of characters:

```
DSUBSTR( 15, 'あaいiうuえeおo', 4, 6, 3, 'A10')
The result is iうu.
```

EDIT: Extracting or Adding Characters

How to:

Extract or Add Characters

The EDIT function extracts characters from the source string and adds characters to the output string, according to the mask. It can extract a substring from different parts of the source string. It can also insert characters from the source string into an output string. For example, it can extract the first two characters and the last two characters of a string to form a single output string.

EDIT compares the characters in a *mask* to the characters in a *source_string*. When it encounters a 9 in the mask, EDIT copies the corresponding character from the source string to the output string. When it encounters a dollar sign $ in the mask, EDIT ignores the corresponding character in the source string. When it encounters any other character in the mask, EDIT copies that character to the corresponding position in the output string. This process ends when the mask is exhausted.

Syntax: How to Extract or Add Characters

```
EDIT(source_string, 'mask')
```

where:

source_string

Alphanumeric

Is a character string from which to pick characters. Each 9 in the mask represents one digit so the size of the source_string must be at least as large as the number of 9's that appear in the mask.

mask

Alphanumeric

Is a string of mask characters.

The length of the mask without characters $ determines the length of the output string.

Note:

- EDIT does not require an output_format argument because the result is obviously alphanumeric and its size is determined from the value of the mask argument.
- EDIT can also convert the format of a field as described in *EDIT: Converting the Format of a Field* on page 145, which can be combined with extracting and adding characters.

Example: Extracting Characters from a Column

EDIT extracts the first initials from the FNAME column.

```
EDIT(FNAME, '9$$$$$$$$$')
```

For GREGORY, the result is G.

For STEVEN, the result is S.

GETTOK: Extracting a Substring (Token)

How to:

Extract a Substring (Token)

The GETTOK function parses a source character string, finding substrings called tokens. A specific character, called a delimiter, occurs in the string and separates the string into tokens. GETTOK returns the token whose number is the *token_number*. It ignores leading and trailing blanks in the source character string.

For example, suppose you want to extract the fourth word from a sentence. In this case use the space character for a delimiter and the number 4 for *token_number*. GETTOK divides the sentence into words using this delimiter, then extracts the fourth word. If the string is not divided by the delimiter, use the PARAG function for this purpose. See *PARAG: Dividing Text Into Smaller Lines* on page 51.

Syntax: How to Extract a Substring (Token)

```
GETTOK(source_string, length, token_number, 'delim', outlen, output_format)
```

where:

source_string

Alphanumeric

Is the source character string from which to extract the token.

length

Integer

Is the number of characters in the source_string. If this argument is less than or equal to 0, the function returns spaces.

token_number

Integer

Is the token number to extract. If this argument is positive, the tokens are counted from left to right. If this argument is negative, the tokens are counted from right to left. For example, -2 extracts the second token from the right. If this argument is 0, the function returns spaces. Leading and trailing null tokens are ignored.

'delim'

Alphanumeric

Is one character delimiter. If you specify more than one character, only the first character is used.

outlen

Integer

Is the size of the token extracted. If this argument is less than or equal to 0, the function returns spaces. If the token is longer than this argument, it is truncated; if it is shorter, it is padded with trailing spaces.

output_format

Alphanumeric

Note that the delimiter is not included in the extracted token.

Example: Extracting a Token From a Column

GETTOK extracts the last token from ADDRESS_LN3 and stores the result in a column with the format A10:

```
GETTOK(ADDRESS_LN3, 20, -1, ' ', 10, 'A10')
```

In this case, the last token will be the zip code.

For RUTHERFORD NJ 07073, the result is 07073.

For NEW YORK NY 10039, the result is 10039.

JPTRANS: Converting Japanese Specific Characters

How to:

Convert Japanese Specific Characters

The JPTRANS function converts Japanese specific characters.

Syntax: How to Convert Japanese Specific Characters

```
JPTRANS ('type_of_conversion', length, source_string, 'output_format')
```

where:

type_of_conversion

Is one of the following options indicating the type of conversion you want to apply to Japanese specific characters. These are the single component input types:

Conversion Type	Description
'UPCASE'	Converts Zenkaku(Fullwidth) alphabets to Zenkaku uppercase.
'LOCASE'	Converts Zenkaku alphabets to Zenkaku lowercase.
'HNZNALPHA'	Converts alphanumerics from Hankaku (Halfwidth) to Zenkaku.
'HNZNSIGN'	Converts ASCII symbols from Hankaku to Zenkaku.
'HNZNKANA'	Converts Katakana from Hankaku to Zenkaku.
'HNZNSPACE'	Converts space (blank) from Hankaku to Zenkaku.
'ZNHNALPHA'	Converts alphanumerics from Zenkaku to Hankaku.
'ZNHNSIGN'	Converts ASCII symbols from Zenkaku to Hankaku.
'ZNHNKANA'	Converts Katakana from Zenkaku to Hankaku.
'ZNHNSPACE'	Converts space from Zenkaku to Hankaku.
'HIRAKATA'	Converts Hiragana to Zenkaku Katakana.
'KATAHIRA'	Converts Zenkaku Katakana to Hiragana.
'930TO939'	Converts codepage from 930 to 939.

Conversion Type	Description
'939TO930'	Converts codepage from 939 to 930.

length

Integer

Is the number of characters in the source string.

source_string

Alphanumeric

Is the string to convert.

output_format

Alphanumeric

Is the name of the field that contains the output, or the format enclosed in single quotation marks.

Example: Using the JPTRANS Function

For ａｂｃ, the result is ＡＢＣ.

```
JPTRANS('UPCASE', 20, Alpha_DBCS_Field, 'A20')
```

For ＡＢＣ, the result is ａｂｃ.

```
JPTRANS('LOCASE', 20, Alpha_DBCS_Field, 'A20')
```

For AaBbCc123, the result is ＡａＢｂＣｃ１２３.

```
JPTRANS('HNZNALPHA', 20, Alpha_SBCS_Field, 'A20')
```

For !@$%,.?, the result is ！＠＄％、。？

```
JPTRANS('HNZNSIGN', 20, Symbol_SBCS_Field, 'A20')
```

For 「ﾍﾞｰｽﾎﾞｰﾙ。」, the result is 「ベースボール。」

```
JPTRANS('HNZNKANA', 20, Hankaku_Katakana_Field, 'A20')
```

For ｱｲｳ, the result is ア　イ　ウ

```
JPTRANS('HNZNSPACE', 20, Hankaku_Katakana_Field, 'A20')
```

For Ａ ａ Ｂ ｂ Ｃ ｃ １ ２ ３, the result is AaBbCc123.

```
JPTRANS('ZNHNALPHA', 20, Alpha_DBCS_Field, 'A20')
```

For ！ @ ＄ ％、 。 ？, the result is !@$%,.?

```
JPTRANS('ZNHNSIGN', 20, Symbol_DBCS_Field, 'A20')
```

For 「ベースボール。」, the result is 「ﾍﾞｰｽﾎﾞｰﾙ。」

```
JPTRANS('ZNHNKANA', 20, Zenkaku_Katakana_Field, 'A20')
```

For ア　イ　ウ, the result is ア イ ウ

```
JPTRANS('ZNHNSPACE', 20, Zenkaku_Katakana_Field, 'A20')
```

For あいう, the result is アイウ

```
JPTRANS('HIRAKATA', 20, Hiragana_Field, 'A20')
```

For アイウ, the result is あいう

```
JPTRANS('KATAHIRA', 20, Zenkaku_Katakana_Field, 'A20')
```

In the following, codepoints 0x62 0x63 0x64 are converted to 0x81 0x82 0x83, respectively:

```
JPTRANS('930TO939', 20, CP930_Field, 'A20')
```

In the following, codepoints 0x59 0x62 0x63 are converted to 0x81 0x82 0x83, respectively:

```
JPTRANS('939TO930', 20, CP939_Field, 'A20')
```

Reference: **Usage Notes for the JPTRANS Function**

- HNZNSIGN and ZNHNSIGN focus on the conversion of symbols.

  Many symbols have a one-to-one relation between Japanese Fullwidth characters and ASCII symbols, whereas some characters have one-to-many relations. For example, the Japanese punctuation character (U+3001) and Fullwidth comma , (U+FF0C) will be converted to the same comma , (U+002C). We have the following EXTRA rule for those special cases.

  HNZNSIGN:

  - Double Quote " (U+0022) -> Fullwidth Right Double Quote ” (U+201D)
  - Single Quote ' (U+0027) -> Fullwidth Right Single Quote ’ (U+2019)
  - Comma , (U+002C) -> Fullwidth Ideographic Comma (U+3001)

- Full Stop . (U+002E) -> Fullwidth Ideographic Full Stop ? (U+3002)
- Backslash \ (U+005C) -> Fullwidth Backslash \ (U+FF3C)
- Halfwidth Left Corner Bracket (U+FF62) -> Fullwidth Left Corner Bracket (U+300C)
- Halfwidth Right Corner Bracket (U+FF63) -> Fullwidth Right Corner Braket (U+300D)
- Halfwidth Katakana Middle Dot ? (U+FF65) -> Fullwidth Middle Dot · (U+30FB)

ZNHNSIGN:

- Fullwidth Right Double Quote ” (U+201D) -> Double Quote " (U+0022)
- Fullwidth Left Double Quote “ (U+201C) -> Double Quote " (U+0022)
- Fullwidth Quotation " (U+FF02) -> Double Quote " (U+0022)
- Fullwidth Right Single Quote ’ (U+2019) -> Single Quote ' (U+0027)
- Fullwidth Left Single Quote ‘ (U+2018) -> Single Quote ' (U+0027)
- Fullwidth Single Quote ' (U+FF07) -> Single Quote ' (U+0027)
- Fullwidth Ideographic Comma (U+3001) -> Comma , (U+002C)
- Fullwidth Comma , (U+FF0C) -> Comma , (U+002C)
- Fullwidth Ideographic Full Stop ? (U+3002) -> Full Stop . (U+002E)
- Fullwidth Full Stop . (U+FF0E) -> Full Stop . (U+002E)
- Fullwidth Yen Sign ¥ (U+FFE5) -> Yen Sign ¥ (U+00A5)
- Fullwidth Backslash \ (U+FF3C) -> Backslash \ (U+005C)
- Fullwidth Left Corner Bracket (U+300C) -> Halfwidth Left Corner Bracket (U+FF62)
- Fullwidth Right Corner Bracket (U+300D) -> Halfwidth Right Corner Bracket (U+FF63)
- Fullwidth Middle Dot · (U+30FB) -> Halfwidth Katakana Middle Dot · (U+FF65)

- HNZNKANA and ZNHNKANA focus on the conversion of Katakana

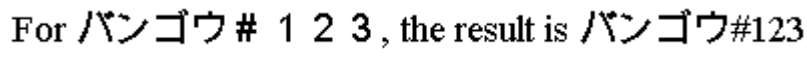
  They convert not only letters but also punctuation symbols on the following list:

  - Fullwidth Ideographic Comma (U+3001) <-> Halfwidth Ideographic Comma (U+FF64)
  - Fullwidth Ideographic Full Stop (U+3002) <-> Halfwidth Ideographic Full Stop (U+FF61)
  - Fullwidth Left Corner Bracket (U+300C) <-> Halfwidth Left Corner Braket (U+FF62)
  - Fullwidth Right Corner Bracket (U+300D) <-> Halfwidth Right Corner Bracket (U+FF63)

- Fullwidth Middle Dot · (U+30FB) <-> Halfwidth Katakana Middle Dot · (U+FF65)
- Fullwidth Prolonged Sound (U+30FC) <-> Halfwidth Prolonged Sound (U+FF70)

- JPTRANS can be nested for multiple conversions.

  For example, text data may contain fullwidth numbers and fullwidth symbols. In some situations, they should be cleaned up for ASCII numbers and symbols.


  ```
  JPTRANS('ZNHNALPHA', 20, JPTRANS('ZNHNSIGN', 20, Symbol_DBCS_Field,
  'A20'), 'A20')
  ```

- HNZNSPACE and ZNHNSPACE focus on the conversion of a space (blank character).

  Currently only conversion between U+0020 and U+3000 is supported.

LCWORD: Converting a Character String to Mixed Case

How to:

Convert a Character String to Mixed Case

The LCWORD function converts the letters in a character string to mixed case. It converts every alphanumeric character to lowercase except the first letter of each word and the first letter after a single or double quotation mark. For example, O'CONNOR is converted to O'Connor and JACK'S to Jack'S.

LCWORD skips numeric and special characters in the source string and continues to convert the following alphabetic characters. The result of LCWORD is a string in which the initial uppercase characters of all words are followed by lowercase characters.

Syntax: How to Convert a Character String to Mixed Case

```
LCWORD(length, source_string, output_format)
```

where:

length

Integer

Is the number of characters in the source_string and output_format.

source_string

Alphanumeric

Is the character string to be converted.

output_format
Alphanumeric

Example: Converting a Character String to Mixed-Case

LCWORD converts LAST_NAME to mixed-case and stores the result in a column with the format A15:

```
LCWORD(15, LAST_NAME, 'A15')
```

For STEVENS, the result is Stevens.

For SMITH, the result is Smith.

LJUST: Left-justifying a Character String

How to:

Left-justify a Character String

The LJUST function left-justifies a character string.

Syntax: How to Left-justify a Character String

```
LJUST(length, source_string, output_format)
```

where:

length
Integer

Is the number of characters in the source_string and output_format.

string
Alphanumeric

Is the character string to be justified.

output_format
Alphanumeric

Example: Left-justifying a Column

LJUST left-justifies FNAME and stores the result in a column with the format A25:

```
LJUST(15, FNAME, 'A25')
```

LOCASE: Converting Text to Lowercase

How to:

Convert Text to Lowercase

The LOCASE function converts alphanumeric text to lowercase.

Syntax: How to Convert Text to Lowercase

```
LOCASE(length, source_string, output_format)
```

where:

length

Integer

Is the number of characters in the source_string and output_format. The length must be greater than 0.

source_string

Alphanumeric

Is a string to convert.

output_format

Alphanumeric

Example: Converting a Field to Lowercase

LOCASE converts LAST_NAME to lowercase and stores the result in a column with the format A15:

```
LOCASE(15, LAST_NAME, 'A15')
```

For SMITH, the result is smith.

For JONES, the result is jones.

OVRLAY: Overlaying a Character String

How to:

Overlay a Character String

The OVRLAY function overlays a base character string with a substring. The function enables you to edit part of an alphanumeric field without replacing the entire field.

Syntax: How to Overlay a Character String

```
OVRLAY(source_string, length, substring, sublen, position, output_format)
```

where:

source_string

Alphanumeric

Is the base character string.

length

Integer

Is the number of characters in the source_string and output_format. If this argument is less than or equal to 0, unpredictable results occur.

substring

Alphanumeric

Is the substring that will overlay the source_string.

sublen

Integer

Is the number of characters in the substring. If this argument is less than or equal to 0, the function returns spaces.

position

Integer

Is the position in the source_string at which the overlay begins. If this argument is less than or equal to 0, the function returns spaces. If this argument is larger than length, the function returns the base string.

output_format

Alphanumeric

Note that if the overlaid string is longer than the output field, the string is truncated to fit the field.

Example: Replacing Characters in a Character String

OVRLAY replaces the last three characters of EMP_ID with CURR_JOBCODE to create a new identification code and stores the result in a column with the format A9:

```
OVRLAY(EMP_ID, 9, CURR_JOBCODE, 3, 7, 'A9')
```

For EMP_ID of 326179357 with CURR_JOBCODE of B04, the result is 26179B04.

For EMP_ID of 818692173 with CURR_JOBCODE of A17, the result is 818692A17.

PARAG: Dividing Text Into Smaller Lines

How to:

Divide Text Into Smaller Lines

The PARAG function divides a character string into substrings by marking them with a delimiter. It scans a specific number of characters from the beginning of the string and replaces the last space in the group scanned with the delimiter, thus creating a first substring, also known as a token. It then scans the next group of characters in the line, starting from the delimiter, and replaces its last space with a second delimiter, creating a second token. It repeats this process until it reaches the end of the line.

Once each token is marked off by the delimiter, you can use the function GETTOK to place the tokens into different fields (see *GETTOK: Extracting a Substring (Token)* on page 41). If PARAG does not find any spaces in the group it scans, it replaces the first character after the group with the delimiter. Therefore, make sure that any group of characters has at least one space. The number of characters scanned is provided as the maximum token size.

For example, if you have a field called 'subtitle' which contains a large amount of text consisting of words separated by spaces, you can cut the field into roughly equal substrings by specifying a maximum token size to divide the field. If the field is 350 characters long, divide it into three substring by specifying a maximum token size of 120 characters. This technique enables you to print lines of text in paragraph form.

Tip: If you divide the lines evenly, you may create more sub-lines than you intend. For example, suppose you divide 120-character text lines into two lines of 60 characters maximum, but one line is divided so that the first sub-line is 50 characters and the second is 55. This leaves room for a third sub-line of 15 characters. To correct this, insert a space (using weak concatenation) at the beginning of the extra sub-line, then append this sub-line (using strong concatenation) to the end of the one before it. Note that the sub-line will be longer than 60 characters.

Syntax: How to Divide Text Into Smaller Lines

```
PARAG(length, source_string, 'delimiter', max_token_size, output_format)
```

where:

length

Integer

Is the number of characters in the source_string and output_format.

source_string

Alphanumeric

Is a string to divide into tokens.

delimiter

Alphanumeric

Is the delimiter character. Choose a character that does not appear in the text.

max_token_size

Integer

Is the upper limit for the size of each token.

output_format

Alphanumeric

Example: **Dividing Text Into Smaller Lines**

PARAG divides ADDRESS_LN2 into smaller lines of not more than ten characters, using a comma as the delimiter. The result is stored in a column with the format A20:

```
PARAG(20, ADDRESS_LN2, ',', 10, 'A20')
```

For 147-15 NORTHERN BLD, the result is 147-15,NORTHERN,BLD.

For 13 LINDEN AVE., the result is 13 LINDEN,AVE.

PATTERN: Generating a Pattern From an Input String

How to:

Generate a Pattern From an Input String

The PATTERN function examines a source string and produces a pattern that indicates the sequence of numbers, uppercase letters, and lowercase letters in the input string. In the output pattern, any character from the input that represented a digit becomes the character '9', any character that represents an uppercase letter becomes 'A', and any character that represents a lowercase letter becomes 'a'. Special characters remain unchanged. An unprintable character becomes the character 'X'. This function is useful for examining data to make sure that it follows a standard pattern.

Syntax: How to Generate a Pattern From an Input String

```
PATTERN (length, infield,  outfield)
```

where:

length

Numeric

Is the length of *infield*.

infield

Alphanumeric

Is a field containing the input string, or a literal string enclosed in single quotation marks.

outfield

Alphanumeric

Is the name of the field to contain the result or the format of the field enclosed in single quotation marks.

Example: Producing a Pattern From Alphanumeric Data

The following 19 records are stored in a fixed format sequential file (with LRECL 14) named TESTFILE:

```
212-736-6250
212 736 4433
123-45-6789
800-969-INFO
10121-2898
10121
2 Penn Plaza
917-339-6380
917-339-4350
(212) 736-6250
(212) 736-4433
212-736-6250
212-736-6250
212-736-6250
(212) 736 5533
(212) 736 5533
(212) 736 5533
10121 Æ
800-969-INFO
```

The Master File is:

```
FILENAME=TESTFILE, SUFFIX=FIX     ,
  SEGMENT=TESTFILE, SEGTYPE=S0, $
    FIELDNAME=TESTFLD, USAGE=A14, ACTUAL=A14, $
```

The following request generates a pattern for each instance of TESTFLD and displays them by the pattern that was generated. It shows the count of each pattern and its percentage of the total count. The PRINT command shows which values of TESTFLD generated each pattern.

```
CMS FILEDEF TESTFILE DISK Atestfile.ftm
DEFINE FILE TESTFILE
   PATTERN/A14 = PATTERN (14, TESTFLD, 'A14' ) ;
END

TABLE FILE TESTFILE
  SUM CNT.PATTERN AS 'COUNT' PCT.CNT.PATTERN AS 'PERCENT'

   BY PATTERN
 PRINT TESTFLD
   BY PATTERN
ON TABLE COLUMN-TOTAL
END
```

Note that the next to last line produced a pattern from an input string that contained an unprintable character, so that character was changed to X. Otherwise, each numeric digit generated a 9 in the output string, each uppercase letter generated the character 'A', and each lowercase letter generated the character 'a'. The output is:

```
PATTERN          COUNT  PERCENT  TESTFLD
-------          -----  -------  -------
(999) 999 9999       3    15.79  (212) 736 5533
                                 (212) 736 5533
                                 (212) 736 5533
(999) 999-9999       2    10.53  (212) 736-6250
                                 (212) 736-4433
9 Aaaa Aaaaa         1     5.26  2 Penn Plaza
999 999 9999         1     5.26  212 736 4433
999-99-9999          1     5.26  123-45-6789
999-999-AAAA         2    10.53  800-969-INFO
                                 800-969-INFO
999-999-9999         6    31.58  212-736-6250
                                 917-339-6380
                                 917-339-4350
                                 212-736-6250
                                 212-736-6250
                                 212-736-6250
99999                1     5.26  10121
99999 X              1     5.26  10121 Æ
99999-9999           1     5.26  10121-2898

TOTAL               19   100.00
```

POSIT: Finding the Beginning of a Substring

How to:

Find the Beginning of a Substring

The POSIT function finds the starting position of a substring within a source string. For example, the starting position of the substring DUCT in the string PRODUCTION is 4. If the substring is not in the source string, the function returns the value 0.

Syntax: How to Find the Beginning of a Substring

```
POSIT(source_string, length, substring, sublength, output_format)
```

where:

source_string

Alphanumeric

Is the string to parse.

length

Integer

Is the number of character in the source_string. If this argument is less than or equal to 0, the function returns a 0.

substring

Alphanumeric

Is the substring whose position you want to find.

sublength

Integer

Is the number of characters in substring. If this argument is less than or equal to 0, or if it is greater than length, the function returns a 0.

output_format

Integer

Example: Finding the Position of a Letter

POSIT determines the position of the first capital letter I in LAST_NAME and stores the result in a column with the format I2:

```
POSIT(LAST_NAME, 15, 'I', 1, 'I2')
```

For STEVENS, the result is 0.

For SMITH, the result is 3.

For IRVING, the result is 1.

REVERSE: Reversing a Character String

How to:

Reverse a Character String

The REVERSE function reverses the characters in a character string.

Syntax: How to Reverse a Character String

```
REVERSE(length, source_string, output_format)
```

where:

length

Integer

Is the number of characters in the source_string and output_format.

source_string

Alphanumeric

Is the string to reverse characters.

output_format

Alphanumeric

Example: Reversing the Characters in a String

Reverse reverses the characters in PRODCAT and stores the result in a column with the format A15:

```
REVERSE(15, PRODCAT, 'A15')
```

For VCRs, the result is sRCV.

For DVD, the result is DVD.

RJUST: Right-justifying a Character String

How to:

Right-justify a Character String

The RJUST function right-justifies a character string. All trailing blacks become leading blanks. This is useful when you display alphanumeric fields containing numbers.

Syntax: How to Right-justify a Character String

```
RJUST(length, source_string, output_format)
```

where:

length

Integer

Is the number of characters in the source_string and output_format. Their lengths must be the same to avoid justification problems.

source_string

Alphanumeric

Is the string to right justify.

output_format

Alphanumeric

Example: Right-justifying a Column

RJUST right-justifies LAST_NAME and stores the result in a column with the format A15:

```
RJUST(15, LAST_NAME, 'A15')
```

SOUNDEX: Comparing Character Strings Phonetically

How to:

Compare Character Strings Phonetically

The SOUNDEX function analyzes a character string phonetically, without regard to spelling. It converts a source string to 4-character code. The first character must be the first character in the string. The last three characters represent the next three significant sounds in the source string.

Syntax: How to Compare Character Strings Phonetically

```
SOUNDEX(length, source_string, output_format)
```

where:

length

Alphanumeric

Is the string whose numeric characters specify the number of characters in the source_string. It should be two characters long because the number must be from 1 to 99; a number greater than 99 causes the function to return asterisks (***).

source_string

Alphanumeric

Is the string to analyze.

output_format

Alphanumeric

Example: Comparing Character Strings Phonetically

SOUNDEX analyzes LAST_NAME phonetically and stores the result in a column with the format A4.

```
SOUNDEX('15', LAST_NAME, 'A4')
```

SPELLNM: Spelling Out a Dollar Amount

How to:

Spell Out a Dollar Amount

The SPELLNM function spells out an alphanumeric string or numeric value containing two decimal places as dollars and cents. For example, the value 32.50 is spelled out as THIRTY TWO DOLLARS AND FIFTY CENTS.

Syntax: How to Spell Out a Dollar Amount

```
SPELLNM(outlength, number, output_format)
```

where:

outlength

Integer

Is the number of characters in the output_format.

If you know the maximum value of the number, use the following table to determine the value of the outlength:

If the **number** is less than...	...**outlength** should be
$10	37
$100	45
$1.000	59
$10,000	74
$100,000	82
$1,000,000	96

number

Alphanumeric or any numeric format (9.2)

Is the number to be spelled out. This value must contain two decimal places.

output_format

Alphanumeric

Example: Spelling Out a Dollar Amount

SPELLNM spells out the values in CURR_SAL and stores the result in a column with the format A82:

```
SPELLNM(82, CURR_SAL, 'A82')
```

For $13,200.00, the result is THIRTEEN THOUSAND TWO HUNDRED DOLLARS AND NO CENTS.

For $18,480.00, the result is EIGHTEEN THOUSAND FOUR HUNDRED EIGHTY DOLLARS AND NO CENTS.

SQUEEZ: Reducing Multiple Spaces to a Single Space

How to:

Reduce Multiple Spaces to a Single Space

The SQUEEZ function reduces multiple contiguous spaces within a character string to a single space. The resulting character string has the same length as the original string but is padded on the right with spaces.

Syntax: How to Reduce Multiple Spaces to a Single Space

```
SQUEEZ(length, source_string, output_format)
```

where:

length

Integer

Is the number of characters in the source_string and output_format.

source_string

Alphanumeric

Is the string to squeeze.

output_format
Alphanumeric

Example: Reducing Multiple Spaces to a Single Space

SQUEEZ reduces multiple spaces in NAME to a single blank and stores the result in a column with the format A30:

```
SQUEEZ(30, NAME, 'A30')
```

For MARY SMITH, the result is MARY SMITH.

For DIANE JONES, the result is DIANE JONES.

For JOHN MCCOY, the result is JOHN MCCOY.

STRIP: Removing a Character From a String

How to:

Remove a Character From a String

The STRIP function removes all occurrences of a specific character from a string. The resulting character string has the same length as the original string but is padded on the right with spaces.

Syntax: How to Remove a Character From a String

```
STRIP(length, source_string, char, output_format)
```

where:

length
Integer

Is the number of characters in the source_string and output_format.

source_string
Alphanumeric

Is the string from which the character will be removed.

char

Alphanumeric

Is the character to be removed from the string. If more than one character is provided, the left-most character will be used as the strip character.

Note: To remove single quotation marks, use two consecutive quotation marks. You must then enclose this character combination in single quotation marks.

output_format

Alphanumeric

Example: Removing Occurrences of a Character From a String

STRIP removes all occurrences of a period (.) from DIRECTOR and stores the result in a field with the format A17:

```
STRIP(17, DIRECTOR, '.', 'A17')
```

For ZEMECKIS R., the result is ZEMECKIS R.

For BROOKS J.L., the result is BROOKS JL.

STRREP: Replacing Character Strings

How to:

Replace Character Strings

Reference:

Usage Notes for STRREP Function

The STRREP function enables you to replace all instances of a specified string within a given input string. It also supports replacement by null strings.

Syntax: How to Replace Character Strings

```
STRREP(inlength, instring, searchlength, searchstring, replength,
repstring, outlength, outstring)
```

where:

inlength

Numeric

Is the number of characters in the input string.

instring

Alphanumeric

Is the input string.

searchlength

Numeric

Is the number of characters in the (shorter length) string to be replaced.

searchstring

Alphanumeric

Is the character string to be replaced.

replength

Numeric

Is the number of characters in the replacement string. Must be zero (0) or greater.

repstring

Alphanumeric

Is the replacement string (alphanumeric). Ignored if replength is zero (0).

outlength

Numeric

Is the number of characters in the resulting output string. Must be 1 or greater.

outstring

Alphanumeric

Is the resulting output string after all replacements and padding.

Example: Replacing Commas and Dollar Signs

In the following example, STRREP finds and replaces commas and dollar signs that appear in the CS_ALPHA field, first replacing commas with null strings to produce CS_NOCOMMAS (removing the commas) and then replacing the dollar signs ($) with (USD) in the right-most CURR_SAL column:

```
TABLE FILE EMPLOYEE
SUM CURR_SAL NOPRINT
COMPUTE CS_ALPHA/A15=FTOA(CURR_SAL,'(D12.2M)',CS_ALPHA);
        CS_NOCOMMAS/A14=STRREP(15,CS_ALPHA,1,',',0,'X',14,CS_NOCOMMAS);
        CS_USD/A17=STRREP(14,CS_NOCOMMAS,1,'$',4,'USD ',17,CS_USD);
        NOPRINT
        CS_USD/R AS CURR_SAL
BY LAST_NAME
END
```

The output is:

```
LAST_NAME   CS_ALPHA             CS_NOCOMMAS                CURR_SAL
---------   --------             -----------        -----------------
BANNING          $29,700.00           $29700.00          USD 29700.00
BLACKWOOD        $21,780.00           $21780.00          USD 21780.00
CROSS            $27,062.00           $27062.00          USD 27062.00
GREENSPAN         $9,000.00            $9000.00          USD  9000.00
IRVING           $26,862.00           $26862.00          USD 26862.00
JONES            $18,480.00           $18480.00          USD 18480.00
MCCOY            $18,480.00           $18480.00          USD 18480.00
MCKNIGHT         $16,100.00           $16100.00          USD 16100.00
ROMANS           $21,120.00           $21120.00          USD 21120.00
SMITH            $22,700.00           $22700.00          USD 22700.00
STEVENS          $11,000.00           $11000.00          USD 11000.00
```

Reference: Usage Notes for STRREP Function

The maximum string length is 4095.

SUBSTR: Extracting a Substring

How to:

Extract a Substring

The SUBSTR function extracts a substring based on where it begins and its length in the source string.

Syntax: How to Extract a Substring

```
SUBSTR(length, source_string, start, end, sublength, output_format)
```

where:

length

Integer

Is the number of characters in the source_string field.

source_string

Alphanumeric

Is the string from which to extract a substring.

start

Integer

Is the starting position of the substring in the source_string. If *start* is less than 1 or greater than length, the function returns spaces.

end

Integer

Is the ending position of the substring. If *end* is less than start or greater than length, the function returns spaces.

sublength

Integer

Is the number of characters in the substring (normally *end - start + 1*). If *sublength* is longer than *end - start +1*, the substring is padded with trailing spaces. If it is shorter, the substring is truncated. This value should be the declared length of the output_format. Only sublength characters will be processed.

output_format

Alphanumeric

Example: Extracting a String

SUBSTR extracts the first three characters from LAST_NAME, and stores the results in a column with the format A3:

```
SUBSTR(15, LAST_NAME, 1, 3, 3, 'A3')
```

For BANNING, the result is BAN.

For MCKNIGHT, the result is MCK.

TRIM: Removing Leading and Trailing Occurrences

How to:

Remove Leading and Trailing Occurrences

The TRIM function removes leading and/or trailing occurrences of a pattern within a character string.

Syntax: How to Remove Leading and Trailing Occurrences

```
TRIM('trim_where', source_string, length, pattern, sublength, output_format)
```

where:

trim_where

Alphanumeric

Is one of the following, which indicates where to remove the pattern:
'L' removes leading occurrences.
'T' removes trailing occurrences.
'B' removes both leading and trailing occurrences.

source_string

Alphanumeric

Is the string to trim.

length

Integer

Is the number of characters in the source_string.

pattern

Alphanumeric

Is the character string pattern to remove.

sublength

Integer

Is the number of characters in the pattern.

output_format

Alphanumeric

Example: Removing Leading Occurrences

TRIM removes leading occurrences of the characters BR from DIRECTOR and stores the result in a column with the format A17:

```
TRIM('L', DIRECTOR, 17, 'BR', 2, 'A17')
```

For BROOKS R., the result is OOKS R.

For ABRAHAMS J., the result is ABRAHAMS J.

UPCASE: Converting Text to Uppercase

How to:

Convert Text to Uppercase

The UPCASE function converts a character string to uppercase. It is useful for sorting on a field that contains both mixed case and uppercase values. Sorting on a mixed case field produces incorrect results because the sorting sequence in EBCDIC always places lowercase letters before uppercase letters, while the ASCII sorting sequence always places uppercase letters before lowercase. To obtain correct results, define a new field with all of the values in uppercase, and sort on that.

Syntax: How to Convert Text to Uppercase

```
UPCASE(length, source_string, output_format)
```

where:

length

Integer

Is the number of characters in the source_string and output_format.

source_string

Alphanumeric

Is the string to convert.

output_format

Alphanumeric type A*n*V or A*n*

If the format of the output_format is A*n*V, then the length returned is equal to the smaller of the source_string length and the upper_limit length.

Example: Converting a Mixed-Case Field to Uppercase

UPCASE converts LAST_NAME_MIXED to uppercase and stores the result in a column with the format A15:

```
UPCASE(15, LAST_NAME_MIXED, 'A15')
```

For Banning, the result is BANNING.

For McKnight, the result is MCKNIGHT.

3 Variable Length Character Functions

The character format A*n*V is supported in synonyms for FOCUS, XFOCUS, Fusion, and relational data sources. This format is used to represent the VARCHAR (variable length character) data types supported by relational database management systems.

Topics:

Overview

For relational data sources, A*n*V keeps track of the actual length of a VARCHAR column. This information is especially valuable when the value is used to populate a VARCHAR column in a different RDBMS. It affects whether trailing blanks are retained in string concatenation and, for Oracle, string comparisons (the other relational engines ignore trailing blanks in string comparisons).

In a FOCUS, Fusion, or XFOCUS data source, A*n*V does not provide true variable length character support. It is a fixed-length character field with an extra two leading bytes to contain the actual length of the data stored in the field. This length is stored as a short integer value occupying two bytes. Because of the two bytes of overhead and the additional processing required to strip them, A*n*V format is *not* recommended for use with non-relational data sources.

A*n*V fields can be used as arguments to all Information Builders-supplied functions that expect alphanumeric arguments. An A*n*V input parameter is treated as an A*n* parameter and is padded with blanks to its declared size (*n*). If the last parameter specifies an A*n*V format, the function result is converted to type A*n*V with actual length set equal to its size.

The six functions described in this topic are designed to work specifically with the A*n*V data type parameters.

LENV: Returning the Length of an Alphanumeric Field

How to:

Return the Length of an Alphanumeric Field

The LENV function returns the actual length of an A*n*V input field or the size of an A*n* field.

Syntax: How to Return the Length of an Alphanumeric Field

```
LENV(source_string, output_format)
```

where:

source_string

Alphanumeric of type A*n* or A*n*V.

If it is an An format field, the function returns its size n. For a character string enclosed in quotation marks or a variable, the size of the string or variable is returned. For a field of A*n*V format, its length, taken from the length-in-bytes of the field, is returned.

output_format

Integer

Example: Finding the Length of an A*n*V Field

LENV returns the length of TITLEV and stores the result in a column with the format I2:

```
LENV(TITLEV, 'I2')
```

For ALICE IN WONDERLAND, the result is 19.

For SLEEPING BEAUTY, the result is 15.

LOCASV: Creating a Variable Length Lowercase String

How to:

Create a Variable Length Lowercase String

The LOCASV function converts alphabetic characters in the source_string to lowercase and is similar to LOCASE. LOCASV returns A*n*V output whose actual length is the lesser of the actual length of the A*n*V source string and the value of the input parameter upper_limit.

Syntax: How to Create a Variable Length Lowercase String

```
LOCASV(upper_limit, source_string, output_format)
```

where:

upper_limit

Integer

Is the limit for the length of the source_string.

source_string

Alphanumeric of type A*n* or A*n*V

Is the string to convert. If it is a field of type A*n*V, its length is taken from the length-in-bytes of the field. If the upper_limit is smaller than the length, the source_string is truncated to this upper limit.

output_format

Alphanumeric of type A*n*V or A*n*

If the format of the output_format is A*n*V, the actual length returned is equal to the smaller of the source_string length and the upper_limit.

Example: Creating a Variable Length Lowercase String

LOCASV converts LAST_NAME to lowercase and specifies a length limit of five characters. The results are stored in a column with the format A15V:

```
LOCASV(5, LAST_NAME, 'A15V')
```

For SMITH, the result is smith.

For JONES, the result is jones.

POSITV: Finding the Beginning of a Variable Length Substring

How to:

Find the Beginning of a Variable Length Substring

The POSITV function is similar to POSIT. However, the lengths of its A*n*V parameters are based on the actual lengths of those parameters in comparison with two other parameters that specify their sizes.

Syntax: How to Find the Beginning of a Variable Length Substring

```
POSITV(source_string, upper_limit, substring, sub_limit, output_format)
```

where:

source_string

Alphanumeric of type A*n* or A*n*V

Is the character string that contains the substring whose position you want to find. If it is a field of A*n*V format, its length is taken from the length-in-bytes of the field. If the upper_limit is smaller than the length, the source_string is truncated to this upper limit.

upper_limit

Integer

Is a limit for the length of the source_string.

substring

Alphanumeric of type A*n* or A*n*V

Is the substring whose position you want to locate. If it is a field of type AnV, its length is taken from the length-in-bytes of the field. If the sub_limit is smaller than the length, the source_string is truncated to this sub_limit.

sub_limit

Integer

Is the limit for the length of the substring.

output_format

Integer

Example: Finding the Starting Position of a Variable Length Pattern

POSITV finds the starting position of a comma in TITLEV, which would indicate a trailing definite or indefinite article in a movie title (such as ", THE" in SMURFS, THE). LENV is used to determine the length of title. The result is stored in a column with the format I4

```
POSITV(TITLEV,LENV(TITLEV,'I4'), ',', 1,'I4')
```

For “SMURFS, THE”, the result is 7.

For “SHAGGY DOG, THE”, the result is 11.

SUBSTV: Extracting a Variable Length Substring

How to:

Extract a Variable Length Substring

The SUBSTV function extracts a substring from a string and is similar to SUBSTR. However, the end position for the string is calculated from the starting position and the substring length. Therefore, it has fewer parameters than SUBSTR. Also, the actual length of the output field, if it is an A*n*V field, is determined based on the substring length.

Syntax: How to Extract a Variable Length Substring

```
SUBSTV(upper_limit, source_string, start, sub_limit, output_format)
```

where:

upper_limit

Integer

Is the limit for the length of the source_string.

source_string

Alphanumeric of type A*n* or A*n*V

Is the character string that contains the substring you want to extract. If it is a field of type A*n*V, its length is taken from the length-in-bytes of the field. If the upper_limit is smaller than the length, the source_string is truncated to the upper_limit. The final length value determined by this comparison is referred to as *p_length* (see *output_format* for related information).

start

Integer

Is the starting position of the substring in the source_string. Note that the starting position can exceed the source_string length.

sub_limit

Integer

Is the limit for the length of the substring. The end position of the substring is *end* = *start* + *sub_limit* - *1*. Note that the ending position can exceed the source_string length depending on the provided values for *start* and *sub_limit*.

output_format

Alphanumeric of type A*n*V or A*n*

If the format of output_format is A*n*V, the actual length, *outlen*, is computed as follows from the values for end, start, and p_length (see *source_string* for related information):

If *end* > *p_length* or *end* < *start*, then *outlen* = *0*; otherwise, *outlen* = *end* - *start* + *1*.

Example: Extracting a Variable Length Substring

SUBSTV extracts the first three characters from the TITLEV and stores the result in a column with the format A20V:

```
SUBSTV(39, TITLEV, 1, 3, 'A20V')
```

For SMURFS, the result is SMU.

For SHAGGY DOG, the result is SHA.

TRIMV: Removing Characters From a String

How to:

Remove Characters From a String

The TRIMV function removes a pattern from a string and is similar to TRIM. However, both the source string and the pattern of characters to remove can have an A*n*V format.

Note: The TRIMV function is useful for converting an A*n* field to an A*n*V field (with the length-in-bytes containing the actual length of the data up to the last non-blank character).

Syntax: How to Remove Characters From a String

```
TRIMV(trim_where, source_string, upper_limit, pattern, pattern_limit,
output_format)
```

where:

trim_where

Alphanumeric

Is one of the following, which indicates where to remove the pattern:
'L' removes leading occurrences.
'T' removes trailing occurrences.
'B' removes both leading and trailing occurrences.

source_string

Alphanumeric of type A*n*V or A*n*

Is the string to truncate. If it is a field of type A*n*V, its length is taken from the length-in-bytes of the field. If the upper_limit is smaller than the length, the source_string is truncated to the upper_limit.

upper_limit

Integer

Is the limit value for the length of the source_string.

pattern

Alphanumeric of type A*n*V or A*n*

Is the pattern to remove. If it is a field of type A*n*V, its length is taken from the length-in-bytes of the field. If the pattern_limit is smaller than the length, the pattern is truncated to the pattern_limit.

pattern_limit

Integer

Is the limit for the length of the *pattern*.

output_format

Alphanumeric type A*n*V or A*n*

If the format is A*n*V, the length is set equal to the number of characters left after trimming.

Example: Creating an A*n*V Field by Removing Trailing Blanks

TRIMV removes trailing blanks from TITLE and stores the result in a column with the format A39V:

```
TRIMV('T', TITLE, 39, ' ', 1, 'A39V')
```

UPCASV: Creating a Variable Length Uppercase String

How to:

Create a Variable Length Uppercase String

The UPCASV function converts alphabetic characters in the source_string to uppercase and is similar to UPCASE. UPCASV can return A*n*V output whose actual length is the lesser of the actual length of the A*n*V source_string and an input parameter that specifies the upper limit.

Syntax: How to Create a Variable Length Uppercase String

```
UPCASV(upper_limit, source_string, output_format)
```

where:

upper_limit

Integer

Is the limit for the length of the *source_string*.

source_string

Alphanumeric of type A*n*V or A*n*

Is the string to convert. If it is a field of type AnV, its length is taken from the length-in-bytes of the field. If the *upper_limit* is smaller than the length, the source_string is truncated to the upper_limit.

output_format

Alphanumeric of type A*n*V or A*n*

If the format of the output_format is A*n*V, then the length returned is equal to the smaller of the source_string length and the upper_limit.

Example: Creating a Variable Length Uppercase String

UPCASEV converts LAST_NAME_MIXED to uppercase and stores the result in a column with the format A15V:

```
UPCASEV(15, LAST_NAME_MIXED, 'A15V5')
```

For Banning, the result is BANNING.

For McKnight, the result is MCKNIGHT.

4 Data Source and Decoding Functions

Data source and decoding functions search for data source records, retrieve data source records or values, and assign values based on the value of an input field.

Topics:

DB_LOOKUP: Retrieving Data Source Values

How to:

Retrieve a Value From a Lookup Data Source

Reference:

Usage Notes for DB_LOOKUP

The DB_LOOKUP function enables you to retrieve a value from one data source when running a request against another data source, without joining or combining the two data sources.

DB_LOOKUP compares pairs of fields from the source and lookup data sources to locate matching records and retrieve the value to return to the request. You can specify as many pairs as needed to get to the lookup record that has the value you want to retrieve. If your field list pairs do not lead to a unique lookup record, the first matching lookup record retrieved is used.

DB_LOOKUP can be called in a DEFINE command, TABLE COMPUTE command, MODIFY COMPUTE command, or DataMigrator flow.

There are no restrictions on the source file. The lookup file can be any non-FOCUS data source that is supported as the cross referenced file in a cluster join. The lookup fields used to find the matching record are subject to the rules regarding cross-referenced join fields for the lookup data source. A fixed format sequential file can be the lookup file if it is sorted in the same order as the source file.

Syntax: How to Retrieve a Value From a Lookup Data Source

```
DB_LOOKUP(look_mf, srcfld1, lookfld1, srcfld2, lookfld2, ..., returnfld);
```

where:

look_mf

Alphanumeric

Is the lookup Master File.

srcfld1, srcfld2 ...

Alphanumeric

Are fields from the source file used to locate a matching record in the lookup file.

lookfld1, lookfld2 ...

Alphanumeric

Are columns from the lookup file that share values with the source fields. Only columns in the table or file can be used; columns created with DEFINE cannot be used. For multi-segment synonyms only columns in the top segment can be used.

returnfld

Alphanumeric

Is the name of a column in the lookup file whose value is returned from the matching lookup record. Only columns in the table or file can be used; columns created with DEFINE cannot be used.

Example: Retrieving a Value From a LOOKUP Table

DB_LOOKUP takes the value for STORE_CODE and retrieves the STORENAME associated with it.

DB_LOOKUP(dmcomp,STORE_CODE,STORE_CODE,STORENAME) For 1003CA the result is Audio Expert For 1004MD the result is City Video For 2010AZ the result is eMart

Reference: Usage Notes for DB_LOOKUP

- The maximum number of pairs that can be used to match records is 63.
- If the lookup file is a fixed format sequential file, it must be sorted and retrieved in the same order as the source file. The sequential file's key field must be the first lookup field specified in the DB_LOOKUP request. If it is not, no records will match.

  In addition, if a DB_LOOKUP request against a sequential file is issued in a DEFINE FILE command, you must clear the DEFINE FILE command at the end of the TABLE request that references it or the lookup file will remain open. It will not be reusable until closed and may cause problems when you exit WebFOCUS or FOCUS. Other types of lookup files can be reused without clearing the DEFINE. They will be cleared automatically when all DEFINE fields are cleared.
- If the lookup field has the MISSING=ON attribute in its Master File and the DEFINE or COMPUTE command specifies MISSING ON, the missing value is returned when the lookup field is missing. Without MISSING ON in both places, the missing value is converted to a default value (blank for an alphanumeric field, zero for a numeric field).
- Source records display on the report output even if they lack a matching record in the lookup file.
- Only real fields in the lookup Master File are valid as lookup and return fields.

- If there are multiple rows in the lookup table where the source field is equal to the lookup field, the first value of the return field is returned.

DECODE: Decoding Values

How to:

Decode Values

The DECODE function assigns values based on the coded value of an input field. DECODE is useful for giving a coded value in a field a more meaningful value. For example, the field GENDER may have the code F for female employees and M for male employees for efficient storage (for example, one character instead of six for female). DECODE expands (decodes) these values to ensure correct interpretation on a report.

You can supply codes and decoding values (results) for DECODE directly in the parentheses following the fieldname.

Syntax: How to Decode Values

```
DECODE fieldname(code1 result1 code2 result2...[ELSE default])
DECODE fieldname(filename ...[ELSE default])
```

where:

fieldname

Alphanumeric or Any Numeric Format

Is the name of the input field.

code

Alphanumeric or Any Numeric Format

Is the coded value that DECODE compares with the current value of the field name. If the value has embedded blanks, commas, or other special characters, it must be enclosed in single quotation marks. When DECODE finds the specified value, it returns the corresponding result. When the code is compared to the value of the fieldname, the code and field name must be in the same format.

result

Alphanumeric or Any Numeric Format

Is the returned value that corresponds to the code. If the result has embedded blanks or commas, or contains a negative number, it must be enclosed in single quotation marks. Do not use double quotation marks (").

If the result is presented in alphanumeric format, it must be a non-null, non-blank string. The format of the result must correspond to the datatype of the expression.

default

Alphanumeric or Any Numeric Format

Is the value that is returned as a result for non-matching codes. The format of the default value does not differ from that of the result. If you omit a default value, DECODE returns a blank or zero to non-matching codes.

filename

Alphanumeric

Is the name of the file in which code/result pairs are stored. Every record in the file must contain the pair.

You can use up to 40 lines to define the code and result pairs for any given DECODE function, or 39 lines if you also use an ELSE keyword. Use either a comma or blank to separate the code from the result, or one pair from another.

Note: An output_format is not used with DECODE.

Example: **Supplying Values Using the DECODE Function**

DECODE returns the state abbreviation for PLANT.

```
DECODE PLANT(BOS 'MA' DAL 'TX' LA 'CA')
```

For BOS, the result is MA.

For DAL, the result is TX.

For LA, the result is CA.

FIND: Verifying the Existence of a Value in an Indexed Field

How to:

Verify the Existence of a Value in an Indexed Field

The FIND function determines if an incoming data value is in an indexed FOCUS data source field. The function sets a temporary field to a non-zero value if the incoming value is in the data source field, and to 0 if it is not. A value greater than zero confirms the presence of the data value, not the number of instances in the data source field.

You can also use FIND in a VALIDATE command to determine if a transaction field value exists in another FOCUS data source. If the field value is not in that data source, the function returns a value of 0, causing the validation test to fail and the request to reject the transaction.

You can use any number of FINDs in a COMPUTE or VALIDATE command. However, more FINDs increase processing time and require more buffer space in memory.

Limit: FIND does not work on files with different DBA passwords. FIND only works on FOCUS files.

The opposite of FIND is NOT FIND. The NOT FIND function sets a temporary field to *1* if the incoming value is not in the data source and to *0* if the incoming value is in the data source.

Syntax: How to Verify the Existence of a Value in an Indexed Field

```
FIND(fieldname [AS dbfield] IN file);
```

where:

fieldname

Alphanumeric

Is the name of the field that contains the incoming data value.

AS *dbfield*

Alphanumeric

Is the name of the data source field whose values are compared to the values in the incoming field. This field must be indexed. If the incoming field and the data source field have the same name, omit this phrase.

file

Alphanumeric

Is the name of the indexed FOCUS data source.

Note:

- FIND does not use an output_format.
- Do not include a space between FIND and the left parenthesis.

Example: Verifying the Existence of a Value in an Indexed Field

FIND determines if a supplied value in EMP_ID is in the EDUCFILE data source.

```
FIND(EMP_ID IN EDUCFILE)
```

LAST: Retrieving the Preceding Value

How to:

Retrieve the Preceding Value

The LAST function retrieves the preceding value for a field.

The effect of LAST depends on whether it appears in an extract or load transformation:

- In an extract transformation the LAST value applies to the previous record retrieved from the data source before sorting takes place.
- In a load transformation, the LAST value applies to the record in the previous record loaded.

Syntax: How to Retrieve the Preceding Value

```
LAST fieldname
```

where:

fieldname

Alphanumeric or Numeric

Is the field name.

Note: LAST does not use an output_format.

Example: Retrieving the Preceding Value

LAST retrieves the previous value of DEPARTMENT:

```
LAST DEPARTMENT
```

LOOKUP: Retrieving a Value From a Cross-referenced Data Source

How to:

Retrieve a Value From a Cross-referenced Data Source

The LOOKUP function retrieves a data value from a cross-referenced FOCUS data source in a MODIFY request. You can retrieve data from a data source cross-referenced statically in a synonym or a data source joined dynamically to another by the JOIN command. LOOKUP retrieves a value, but does not activate the field. LOOKUP is required because a MODIFY request, unlike a TABLE request, cannot read cross-referenced data sources freely.

LOOKUP allows a request to use the retrieved data in a computation or message, but it does not allow you to modify a cross-referenced data source.

LOOKUP can read a cross-referenced segment that is linked directly to a segment in the host data source (the host segment). This means that the cross-referenced segment must have a segment type of KU, KM, DKU, or DKM (but not KL or KLU) or must contain the cross-referenced field specified by the JOIN command. Because LOOKUP retrieves a single cross-referenced value, it is best used with unique cross-referenced segments.

The cross-referenced segment contains two fields used by LOOKUP:

- The field containing the retrieved value. Alternatively, you can retrieve all the fields in a segment at one time. The field, or your decision to retrieve all the fields, is specified in LOOKUP.

  For example, LOOKUP retrieves all the fields from the segment:

  ```
  RTN = LOOKUP(SEG.DATE_ATTEND);
  ```

- The cross-referenced field. This field shares values with a field in the host segment called the host field. These two fields link the host segment to the cross-referenced segment. LOOKUP uses the cross-referenced field, which is indexed, to locate a specific segment instance.

When using LOOKUP, the MODIFY request reads a transaction value for the host field. It then searches the cross-referenced segment for an instance containing this value in the cross-referenced field:

- If there are no instances of the value, the function sets a return variable to 0. If you use the field specified by LOOKUP in the request, the field assumes a value of blank if alphanumeric and 0 if numeric.
- If there are instances of the value, the function sets the return variable to 1 and retrieves the value of the specified field from the first instance it finds. There can be more than one if the cross-referenced segment type is KM or DKM, or if you specified the ALL keyword in the JOIN command.

Syntax: How to Retrieve a Value From a Cross-referenced Data Source

```
LOOKUP(field)
```

where:

field

Alphanumeric

Is the name of the field to retrieve in the cross-referenced file. If the field name also exists in the host data source, you must qualify it here. Do not include a space between LOOKUP and the left parenthesis.

Note: LOOKUP does not use an output_format.

Example: Using the LOOKUP Function

LOOKUP finds the enrollment date from DATE_ENROLL. The result can then be used to validate an expression.

```
LOOKUP(DATE_ENROLL)
```

5 Date and Date-Time Functions

Date and Date-Time functions deal with date fields using three types of formats:

- *Standard Date Functions*
- *Date-Time Functions*
- *Legacy Date Functions*

All but three date functions deal with only one date format. The exceptions are **DATECVT**, **HCNVRT**, and **HDATE**, which convert one date type into another.

Overview

The following explains the difference between all three types of date formats:

- **Standard date** functions are for use with standard date formats, or just date formats. A date format refers to internally stored data that is capable of holding date components, such as century, year, quarter, month, and day. It does not include time components. A synonym does not specify an internal data type or length for a date format; instead, it specifies display date components, such as D (day), M (month), Q (quarter), Y (2-digit year), or YY (4-digit year). For example, format MDYY is a date format that has three date components; it can be used in the USAGE attribute of a synonym. A real date value, such as March 9, 2004, described by this format is displayed as 03/09/2004, by default. Date formats can be full component and non-full component. Full component formats include all three letters, i.e. D, M, and Y. JUL for Julian can also be included. All other date formats are non-full component. Some date functions require full component arguments for date fields, while others will accept full or non-full components. A date format was formerly called a smart date.

- **Date-Time** functions are for use with timestamps in date-time formats, also known as H formats. A timestamp value refers to internally stored data capable of holding both date and time components with an accuracy of up to a microsecond.

- **Legacy date** functions are for use with legacy dates only. A legacy date refers to formats with date edit options, such as I6YMD, A6MDY, I8YYMD, or A8MDYY. For example, A6MDY is a 6-byte alphanumeric string; the suffix MDY indicates the order in which the date components are stored in the field, and the prefix I or A indicates a numeric or alphanumeric form of representation. For example, a value '030599' can be assigned to a field with format A6MDY, which will be displayed as 03/05/99.

Date formats have an internal representation matching either numeric or alphanumeric format. For example, A6MDY matches alphanumeric format, YYMD and I6DMY match numeric format. When function output is a date in specified by *output_format*, it can be used either for assignment to another date field of this format, or it can be used for further data manipulation in the expression with data of matching formats. Assignment to another field of a different date format, will yield a random result.

Date and Time Components For Date and Date-Time Functions

The following component names and values are supported as arguments for the date and date-time functions.

Component Name	Valid Values
year	0001-9999
quarter	1-4
month	1-12
day-of-year	1-366
day or day-of-month	1-31 (The two names for the component are equivalent.)
week	1-53
weekday	1-7 (Sunday-Saturday)
hour	0-23
minute	0-59
second	0-59
millisecond	0-999
microsecond	0-999999

- For an argument that specifies a length of 8 or 10 characters, use 8 to include milliseconds and 10 to include microseconds in the returned value.
- The last argument is always a USAGE format that indicates the data type returned by the function. The type may be A (alphanumeric), I (integer), D (floating-point double precision), H (date-time), or a standard date format (for example, YYMD).

Standard Date Functions

The following functions operate on fields with standard date formats:

- *DATEADD: Adding or Subtracting a Date Unit to or From a Date*
- *DATECVT: Converting the Format of a Date*

- *DATEDIF: Finding the Difference Between Two Dates*
- *DATEMOV: Moving a Date to a Significant Point*

The other way to supply a date to a standard date function is to use a natural date in a character string, enclosed in single quotation marks. For example, '20040311' is the natural date that corresponds to '2004 March 11'. In another natural date example, you can assign the field CURDATE to the value of the current date using the system variable &YYMD in natural date representation, as follows:

```
CURDATE/YYMD = '&YYMD'
```

DATEADD: Adding or Subtracting a Date Unit to or From a Date

How to:

Add or Subtract a Date Unit to or From a Date

The DATEADD function adds a unit to or subtracts a unit from a field described by a full component date format.

A unit is one of the following:

- **Year.**
- **Month.** If the calculation using the month unit creates an invalid date, DATEADD corrects it to the last day of the month. For example, adding one month to October 31 yields November 30, not November 31 since November has 30 days.
- **Day.**
- **Weekday.** When using the weekday unit, DATEADD does not count Saturday or Sunday. For example, if you add one day to Friday, the result is Monday.
- **Business day.** When using the business day unit, DATEADD uses the BUSDAYS parameter and a holiday file to determine work days; it disregards the rest. This means that if Monday is not a work day, then one business day past Sunday is Tuesday.

Syntax: How to Add or Subtract a Date Unit to or From a Date

```
DATEADD(date, component_code, increment)
```

where:

date

Date

Is any full component standard date, for example, YYMD, MDY, or JUL.

component_code

Alphanumeric

Is one of the following:

`Y` indicates a year component.
`M` indicates a month component.
`D` indicates a day component.
`WD` indicates a weekday component.
`BD` indicates a business day component.

increment

Integer

Is the number of date components to be added to or subtracted from the date. If this number is not a whole unit, it is rounded down to the next largest integer.

Note: DATEADD does not use an output_format; it uses the format of the argument date for the result. As long as the result is a full component date, it can be assigned only to a full component date field or to integer field.

Example: Adding or Subtracting a Date Unit to or From a Date

This example finds a delivery date that is 12 business days after today:

```
DELIV_DATE/YYMD = DATEADD('&DATEMDYY', 'BD', 12);
```

It returns 20040408, which will be Thursday if today is March 23 2004, Tuesday.

To make sure it is Thursday, assign it as

```
DELIV_DAY/W = DATEADD('&DATEMDYY', 'BD', 12);
```

which returns 4, representing Thursday. Note the use of the system variable &YYMD and the natural_date representation of the today's date.

Tip: There is an alternative way to add to or subtract from the date. As long as any standard date is internally presented as a whole number of the least significant component units (that is, a number of days for full component dates, a number of months for YYM or MY format dates, etc.), you can add/subtract the desired number of these units directly, without DATEADD. Note that you must assign the date result to the same format date field, or the same field. For example, assuming YYM_DATE is a date field of format YYM, you can add 13 months to it and assign the result to the field NEW_YYM_DT, in the following statement:

```
NEW_YYM_DT/YYM = YYM_DATE + 13;
```

Otherwise, a non-full component date must be converted to a full component date before using DATEADD.

DATECVT: Converting the Format of a Date

How to:

Convert the Format of a Date

The DATECVT function converts the field value of any standard date format or legacy date format into a new date, in the desired standard date format or legacy date format. If you supply an invalid format, DATECVT returns a zero or a blank.

Syntax: How to Convert the Format of a Date

```
DATECVT(date, in_format, output_format)
```

where:

date

Date

Is the date to be converted. If you supply an invalid date, DATECVT returns zero. When the conversion is performed, a legacy date obeys any DEFCENT and YRTHRESH parameter settings supplied for that field.

in_format

Alphanumeric

Is the format of the date, either standard, legacy, or non-date.

For example:

- A standard, non-legacy, date format (for example, YYMD, YQ, M, DMY, JUL).
- A legacy date format (for example, I6YMD or A8MDYY).
- A non-date format (such as I8 or A6). This format type in the in_format argument causes DATECVT to interpret the whole number in the date field as a full component date.

output_format

Alphanumeric

Is the output date format, either standard, legacy, or non-date.

For example:

- A standard, non-legacy, date format (for example, YYMD, YQ, M, DMY, JUL).
- A legacy date format (for example, I6YMD or A8MDYY).

- A non-date format (such as I8 or A6). This format type in output_format argument causes DATECVT to convert the date into a full component date and return it as a whole number in the format provided.

Example: Converting the Format of a Date

This example first converts a numeric date, NUMDATE, to a character date, and then assigns the result to a non-date alphanumeric field, CHARDATE.

```
CHARDATE/A13 = DATECVT (NUMDATE,'I8YYMD','A8YYMD');
```

Note: DATECVT does not use an output_format; it uses the format of the argument output_format for the result.

DATEDIF: Finding the Difference Between Two Dates

How to:

Find the Difference Between Two Dates

The DATEDIF function returns the difference between two full component standard dates in desired components.

A component is one of the following:

- **Year.** Using the year unit with DATEDIF yields the inverse of DATEADD. If subtracting one year from date X creates date Y, then the count of years between X and Y is one. Subtracting one year from February 29 produces the date February 28.
- **Month.** Using the month unit with DATEDIF yields the inverse of DATEADD. If subtracting one month from date X creates date Y, then the count of months between X and Y is one. If the to-date is the end-of-month, then the month difference may be rounded up (in absolute terms) to guarantee the inverse rule. If one or both of the input dates is the end of the month, DATEDIF takes this into account. This means that the difference between January 31 and April 30 is three months, not two months.
- **Day.**
- **Weekday.** With the weekday unit, DATEDIF does not count Saturday or Sunday when calculating days. This means that the difference between Friday and Monday is one day.
- **Business day.** With the business day unit, DATEDIF uses the BUSDAYS parameter and holiday file to determine which days are working days and disregards the rest. This means that if Monday is not a working day, the difference between Friday and Tuesday is one day.

DATEDIF returns a whole number. If the difference between two dates is not a whole number, DATEDIF truncates the value to the next largest integer. For example, the number of years between March 2, 2001, and March 1, 2002, is zero. If the end date is before the start date, DATEDIF returns a negative number.

Syntax: How to Find the Difference Between Two Dates

```
DATEDIF('from_date', 'to_date', 'component_code')
```

where:

from_date

Date. Is the start date from which to calculate the difference.

to_date

Date.Is the end date from which to calculate the difference.

component_code

Alphanumeric. Is one of the following enclosed in single quotation marks:

`Y` indicates a year component.
`'M'` indicates a month component.
`'D'` indicates a day component.
`'WD'` indicates a weekday component.
`'BD'` indicates a business day component.

Note: DATEDIF does not use output_format because for the result it uses the format 'I8'.

Example: Finding the Difference Between Two Dates

The example finds the number of complete months between today, March 23, 2004, and one specific day in the past

```
DATEDIF('September 11 2001', '20040323', 'M')
```

and returns 30, which can be assigned to a numeric field.

Tip: There is an alternative way to find the difference between dates. As long as any standard date is presented internally as a whole number of the least significant component units (that is, a number of days for full component dates, a number of months for YYM or MY format dates, etc.), you can find the difference in these component units (not any units) directly, without DATEDIF. For example, assume OLD_YYM_DT is a date field in format MYY and NEW_YYM_DT is another date in format YYM. Note that the least significant component for both formats is month, M. The difference in months, then, can be found by subtracting the field OLD_YYM_DT from NEW_YYM_DT in the following statement:

```
MYDIFF/I8 = NEW_YYM_DT/YYM - OLD_YYM_DT;
```

Otherwise, non-full component standard dates or legacy dates should be converted to full component standard dates before using DATEDIF.

DATEMOV: Moving a Date to a Significant Point

How to:

Move a Date to a Significant Point

The DATEMOV function moves a date to a significant point on the calendar.

Syntax: How to Move a Date to a Significant Point

```
DATEMOV(date, 'move-point')
```

where:

date

Date

Is the date to be moved. It must be a full component format date—for example, MDYY or YYJUL.

move-point

Alphanumeric

Is the significant point the date is moved to. An invalid point results in a return code of zero. Valid values are:

'EOM' is the end of month.
'BOM' is the beginning of month.
'EOQ' is the end of quarter.
'BOQ' is the beginning of quarter.
'EOY' is the end of year.
'BOY' is the beginning of year.
'EOW' is the end of week.
'BOW' is the beginning of week.
'NWD' is the next weekday.
'NBD' is the next business day.
'PWD' is the prior weekday.
'PBD' is the prior business day.
'WD-' is a weekday or earlier.
'BD-' is a business day or earlier.
'WD+' is a weekday or later.
'BD+' is a business day or later.

A business day calculation is affected by the BUSDAYS and HDAY parameter settings.

Note: DATEMOV does not use an output_format; it uses the format of the argument date for the result. As long as the result is a full component date, it can be assigned only to a full component date field or to an integer field.

Example: Moving a Date to a Significant Point

This example finds the end day of the current date week

```
DATEDIF('&YYMD', 'EOW')
```

and returns 20040326 if today is 2004, March 23rd. Note the use of the system variable &YYMD and natural date representation in the first argument.

Date-Time Functions

The functions described in this section operate on fields in date-time format (sometimes called H format). However, you can also provide a date as a character string using the macro DT, followed by a character string in parentheses, presenting date and time. Date components are separated by slashes '/'; time components by colons ':'.

Alternatively, the day can be given as a natural day, like 2004 March 31, in parentheses. Either the date or time component can be omitted. For example, the date-time format argument can be expressed as DT(2004/03/11 13:24:25.99) or DT(March 11 2004).

The following is another example that creates a timestamp representing the current date and time. The system variables &YYMD and &TOD are used to obtain the current date and time, respectively:

```
-SET &MYSTAMP = &YYMD | ' ' | EDIT(&TOD,'99:$99:$99') ;
```

Today's date (&YYMD) is concatenated with the time of day (&TOD). The EDIT function is used to change the dots (.) in the time of day variable to colons (:).

The following request uses the DT macro on the alphanumeric date and time variable &MYSTAMP:

```
TABLE FILE CAR
  PRINT CAR NOPRINT
  COMPUTE   DTCUR/HYYMDS = DT(&MYSTAMP);
  IF RECORDLIMIT IS 1;
END
```

The following are date-time functions:

- *DATETRAN: Formatting Dates in International Formats*
- *HADD: Incrementing a Date-Time Value*
- *HCNVRT: Converting a Date-Time Value to Alphanumeric Format*

DATETRAN: Formatting Dates in International Formats

How to:

Format Dates in International Formats

Reference:

Usage Notes for the DATETRAN Function

The DATETRAN function formats dates in international formats.

Syntax: How to Format Dates in International Formats

```
DATETRAN(indate, '(intype)', '([formatops])', 'lang', outlen, output_format)
```

where:

indate

Is the input date (in date format) to be formatted. Note that the date format cannot be an alphanumeric or numeric format with date display options (legacy date format).

intype

Is one of the following character strings indicating the input date components and the order in which you want them to display, enclosed in parentheses and single quotation marks.

These are the single-component input types:

Single-Component Input Type	Description
'(W)'	Day of week component only (original format must have only W component).
'(M)'	Month component only (original format must have only M component).

These are the two-component input types:

Two-Component Input Type	Description
'(YYM)'	Four-digit year followed by month.
'(YM)'	Two-digit year followed by month.
'(MYY)'	Month component followed by four-digit year.
'(MY)'	Month component followed by two-digit year.

These are the three-component input types:

Three-Component Input Type	Description
'(YYMD)'	Four-digit year followed by month followed by day.
'(YMD)'	Two-digit year followed by month followed by day.
'(DMYY)'	Day component followed by month followed by four-digit year.
'(DMY)'	Day component followed by month followed by two-digit year.

Three-Component Input Type	Description
'(MDYY)'	Month component followed by day followed by four-digit year.
'(MDY)'	Month component followed by day followed by two-digit year.
'(MD)'	Month component followed by day (derived from three-component date by ignoring year component).
'(DM)'	Day component followed by month (derived from three-component date by ignoring year component).

formatops

Is a string of zero or more formatting options enclosed in parentheses and single quotation marks. The parentheses and quotation marks are required even if you do not specify formatting options. Formatting options fall into the following categories:

- Options for suppressing initial zeros in month or day numbers.
- Options for translating month or day components to full or abbreviated uppercase or default case (mixed case or lowercase depending on the language) names.
- Date delimiter options and options for punctuating a date with commas.

Valid options for suppressing initial zeros in month or day numbers are:

Format Option	Description
m	Zero-suppresses months (displays numeric months before October as 1 through 9 rather than 01 through 09).
d	Displays days before the tenth of the month as 1 through 9 rather than 01 through 09.
dp	Displays days before the tenth of the month as 1 through 9 rather than 01 through 09 with a period after the number.
do	Displays days before the tenth of the month as 1 through 9. For English (langcode EN) only, displays an ordinal suffix (st, nd, rd, or th) after the number.

Valid month and day name translation options are:

Format Option	Description
T	Displays month as an abbreviated name with no punctuation, all uppercase.
TR	Displays month as a full name, all uppercase.
Tp	Displays month as an abbreviated name followed by a period, all uppercase.
t	Displays month as an abbreviated name with no punctuation. The name is all lowercase or initial uppercase, depending on language code.
tr	Displays month as a full name. The name is all lowercase or initial uppercase, depending on language code.
tp	Displays month as an abbreviated name followed by a period. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).
W	Includes an abbreviated day of the week name at the start of the displayed date, all uppercase with no punctuation.
WR	Includes a full day of the week name at the start of the displayed date, all uppercase.
Wp	Includes an abbreviated day of the week name at the start of the displayed date, all uppercase, followed by a period.
w	Includes an abbreviated day of the week name at the start of the displayed date with no punctuation. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).
wr	Includes a full day of the week name at the start of the displayed date. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).

Format Option	Description
wp	Includes an abbreviated day of the week name at the start of the displayed date followed by a period. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).
X	Includes an abbreviated day of the week name at the end of the displayed date, all uppercase with no punctuation.
XR	Includes a full day of the week name at the end of the displayed date, all uppercase.
Xp	Includes an abbreviated day of the week name at the end of the displayed date, all uppercase, followed by a period.
x	Includes an abbreviated day of the week name at the end of the displayed date with no punctuation. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).
xr	Includes a full day of the week name at the end of the displayed date. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).
xp	Includes an abbreviated day of the week name at the end of the displayed date followed by a period. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).

Valid date delimiter options are:

Format Option	Description
B	Uses a blank as the component delimiter. This is the default if the month or day of week is translated or if comma is used.
.	Uses a period as the component delimiter.
-	Uses a minus sign as the component delimiter. This is the default when the conditions for a blank default delimiter are not satisfied.

Format Option	Description
/	Uses a slash as the component delimiter.
|	Omits component delimiters.
K	Uses appropriate Asian characters as component delimiters.
c	Places a comma after the month name (following T, Tp, TR, t, tp, or tr). Places a comma and blank after the day name (following W, Wp, WR, w, wp, or wr). Places a comma and blank before the day name (following X, XR, x, or xr).
e	Displays the Spanish or Portuguese word de or DE between the day and month and between the month and year. The case of the word de is determined by the case of the month name. If the month is displayed in uppercase, DE is displayed; otherwise de is displayed. Useful for formats DMY, DMYY, MY, and MYY.
D	Inserts a comma after the day number and before the general delimiter character specified.
Y	Inserts a comma after the year and before the general delimiter character specified.

lang

Is the two-character standard ISO code for the language into which the date should be translated, enclosed in single quotation marks. Valid language codes are:

'AR' Arabic

'CS' Czech

'DA' Danish

'DE' German

'EN' English

'ES' Spanish

'FI' Finnish

'FR' French

'EL' Greek

'IW' Hebrew

'IT' Italian

'JA' Japanese

'KO' Korean

'LT' Lithuanian

'NL' Dutch

'NO' Norwegian

'PO' Polish

'PT' Portuguese

'RU' Russian

'SV' Swedish

'TH' Thai

'TR' Turkish

'TW' Chinese (Traditional)

'ZH' Chinese (Simplified)

outlen

Numeric

Is the length of the output field in bytes. If the length is insufficient, an all blank result is returned. If the length is greater than required, the field is padded with blanks on the right.

output_format

Alphanumeric

Example: **Using the DATETRAN Function**

The following request prints the day of the week in the default case of the specific language:

```
DEFINE FILE VIDEOTRK
TRANS1/YYMD=20050104;
TRANS2/YYMD=20051003;

DATEW/W=TRANS1     ;
DATEW2/W=TRANS2    ;
DATEYYMD/YYMDW=TRANS1    ;
DATEYYMD2/YYMDW=TRANS2   ;
```

```
OUT1A/A8=DATETRAN(DATEW, '(W)', '(wr)', 'EN', 8 , 'A8') ;
OUT1B/A8=DATETRAN(DATEW2, '(W)', '(wr)', 'EN', 8 , 'A8') ;
OUT1C/A8=DATETRAN(DATEW, '(W)', '(wr)', 'ES', 8 , 'A8') ;
OUT1D/A8=DATETRAN(DATEW2, '(W)', '(wr)', 'ES', 8 , 'A8') ;
OUT1E/A8=DATETRAN(DATEW, '(W)', '(wr)', 'FR', 8 , 'A8') ;
OUT1F/A8=DATETRAN(DATEW2, '(W)', '(wr)', 'FR', 8 , 'A8') ;
OUT1G/A8=DATETRAN(DATEW, '(W)', '(wr)', 'DE', 8 , 'A8') ;
OUT1H/A8=DATETRAN(DATEW2, '(W)', '(wr)', 'DE', 8 , 'A8') ;
END

TABLE FILE VIDEOTRK
HEADING
"FORMAT wr"
""
"Full day of week name at beginning of date, default case (wr)"
"English / Spanish / French / German"
""
SUM OUT1A AS '' OUT1B AS '' TRANSDATE NOPRINT
OVER OUT1C AS '' OUT1D AS ''
OVER OUT1E AS '' OUT1F AS ''
OVER OUT1G AS '' OUT1H AS ''
ON TABLE HOLD FORMAT HTML
ON TABLE SET PAGE-NUM OFF
ON TABLE SET STYLE *
GRID=OFF, $
END
```

The output is:

FORMAT wr

Full day of week name at beginning of date, default case (wr)
English / Spanish / French / German

Tuesday	Monday
martes	lunes
mardi	lundi
Dienstag	Montag

The following request prints a blank delimited date with an abbreviated month name in English. Initial zeros in the day number are suppressed, and a suffix is added to the end of the number:

```
DEFINE FILE VIDEOTRK
TRANS1/YYMD=20050104;
TRANS2/YYMD=20050302;

DATEW/W=TRANS1     ;
DATEW2/W=TRANS2    ;
DATEYYMD/YYMDW=TRANS1    ;
DATEYYMD2/YYMDW=TRANS2   ;

OUT2A/A15=DATETRAN(DATEYYMD,  '(MDYY)', '(Btdo)', 'EN', 15, 'A15') ;
OUT2B/A15=DATETRAN(DATEYYMD2, '(MDYY)', '(Btdo)', 'EN', 15, 'A15') ;
END

TABLE FILE VIDEOTRK
HEADING
"FORMAT Btdo"
""
"Blank-delimited (B)"
"Abbreviated month name, default case (t)"
"Zero-suppress day number, end with suffix (do)"
"English"
""
SUM OUT2A AS '' OUT2B AS '' TRANSDATE NOPRINT
ON TABLE HOLD FORMAT HTML
ON TABLE SET PAGE-NUM OFF
END
```

The output is:

FORMAT Btdo Blank-delimited (B) Abbreviated month name, default case (t) Zero-suppress day number, end with suffix (do) English	
Jan 4th 2005	Mar 2nd 2005

The following request prints a blank delimited date with an abbreviated month name in German. Initial zeros in the day number are suppressed, and a period is added to the end of the number:

```
DEFINE FILE VIDEOTRK
TRANS1/YYMD=20050104;
TRANS2/YYMD=20050302;

DATEW/W=TRANS1     ;
DATEW2/W=TRANS2    ;
DATEYYMD/YYMDW=TRANS1    ;
DATEYYMD2/YYMDW=TRANS2   ;

OUT3A/A12=DATETRAN(DATEYYMD,  '(DMYY)', '(Btdp)', 'DE', 12, 'A12');
OUT3B/A12=DATETRAN(DATEYYMD2, '(DMYY)', '(Btdp)', 'DE', 12, 'A12');
END

TABLE FILE VIDEOTRK
HEADING
"FORMAT Btdp"
""
"Blank-delimited (B)"
"Abbreviated month name, default case (t)"
"Zero-suppress day number, end with period (dp)"
"German"
""
SUM OUT3A AS '' OUT3B AS '' TRANSDATE NOPRINT
ON TABLE HOLD FORMAT HTML
ON TABLE SET PAGE-NUM OFF
END
```

The output is:

FORMAT Btdp Blank-delimited (B) Abbreviated month name, default case (t) Zero-suppress day number, end with period (dp) German	
4. Jan 2005	2. Mär 2005

The following request prints a blank delimited date in French with a full day name at the beginning and a full month name, in lower case (the default for French):

```
DEFINE FILE VIDEOTRK
TRANS1/YYMD=20050104;
TRANS2/YYMD=20050302;

DATEW/W=TRANS1     ;
DATEW2/W=TRANS2    ;
DATEYYMD/YYMDW=TRANS1    ;
DATEYYMD2/YYMDW=TRANS2   ;

OUT4A/A30 = DATETRAN(DATEYYMD,  '(DMYY)', '(Bwrtr)', 'FR', 30, 'A30');
OUT4B/A30 = DATETRAN(DATEYYMD2, '(DMYY)', '(Bwrtr)', 'FR', 30, 'A30');
END

TABLE FILE VIDEOTRK
HEADING
"FORMAT Bwrtr"
""
"Blank-delimited (B)"
"Full day of week name at beginning of date, default case (wr)"
"Full month name, default case (tr)"
"English"
""
SUM OUT4A AS '' OUT4B AS '' TRANSDATE NOPRINT
ON TABLE HOLD FORMAT HTML
ON TABLE SET PAGE-NUM OFF
END
```

The output is:

FORMAT Bwrtr

Blank-delimited (B)
Full day of week name at beginning of date, default case (wr)
Full month name, default case (tr)
English

mardi 04 janvier 2005	mercredi 02 mars 2005

The following request prints a blank delimited date in Spanish with a full day name at the beginning in lowercase (the default for Spanish) followed by a comma, and with the word de between the day number and month and between the month and year:

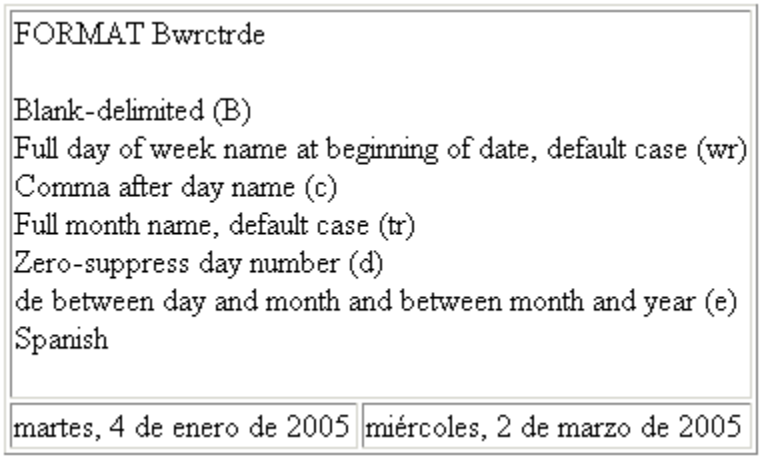
```
DEFINE FILE VIDEOTRK
TRANS1/YYMD=20050104;
TRANS2/YYMD=20050302;

DATEW/W=TRANS1     ;
DATEW2/W=TRANS2    ;
DATEYYMD/YYMDW=TRANS1    ;
DATEYYMD2/YYMDW=TRANS2   ;

OUT5A/A30=DATETRAN(DATEYYMD,  '(DMYY)', '(Bwrctrde)', 'ES', 30, 'A30');
OUT5B/A30=DATETRAN(DATEYYMD2, '(DMYY)', '(Bwrctrde)', 'ES', 30, 'A30');
END

TABLE FILE VIDEOTRK
HEADING
"FORMAT Bwrctrde"
""
"Blank-delimited (B)"
"Full day of week name at beginning of date, default case (wr)"
"Comma after day name (c)"
"Full month name, default case (tr)"
"Zero-suppress day number (d)"
"de between day and month and between month and year (e)"
"Spanish"
""
SUM OUT5A AS '' OUT5B AS '' TRANSDATE NOPRINT
ON TABLE HOLD FORMAT HTML
ON TABLE SET PAGE-NUM OFF
END
```

The output is:



The following request prints a date in Japanese characters with a full month name at the beginning, in the default case and with zero suppression:

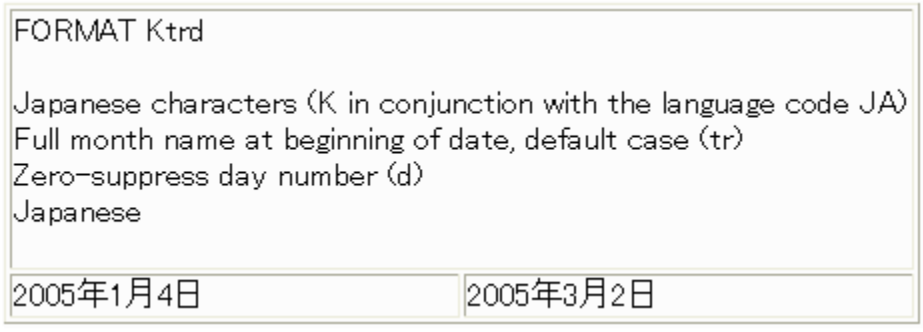
```
DEFINE FILE VIDEOTRK
TRANS1/YYMD=20050104;
TRANS2/YYMD=20050302;

DATEW/W=TRANS1     ;
DATEW2/W=TRANS2    ;
DATEYYMD/YYMDW=TRANS1    ;
DATEYYMD2/YYMDW=TRANS2   ;

OUT6A/A30=DATETRAN(DATEYYMD , '(YYMD)', '(Ktrd)', 'JA', 30, 'A30');
OUT6B/A30=DATETRAN(DATEYYMD2, '(YYMD)', '(Ktrd)', 'JA', 30, 'A30');
END

TABLE FILE VIDEOTRK
HEADING
"FORMAT Ktrd"
""
"Japanese characters (K in conjunction with the language code JA)"
"Full month name at beginning of date, default case (tr)"
"Zero-suppress day number (d)"
"Japanese"
""
SUM OUT6A AS '' OUT6B AS '' TRANSDATE NOPRINT
ON TABLE HOLD FORMAT HTML
ON TABLE SET PAGE-NUM OFF
END
```

The output is:



FORMAT Ktrd Japanese characters (K in conjunction with the language code JA) Full month name at beginning of date, default case (tr) Zero-suppress day number (d) Japanese	
2005年1月4日	2005年3月2日

The following request prints a blank delimited date in Greek with a full day name at the beginning in the default case followed by a comma, and with a full month name in the default case:

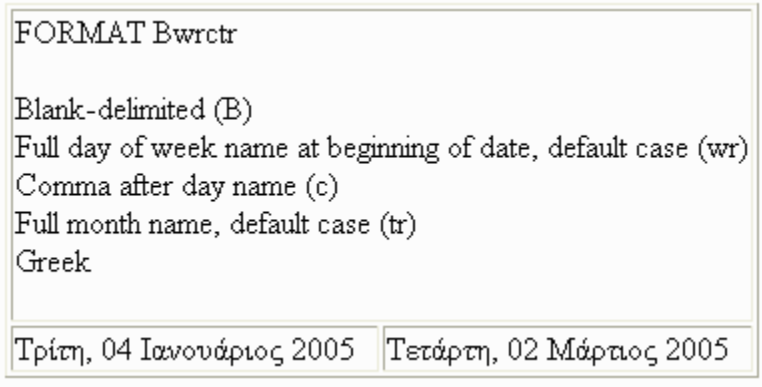
```
DEFINE FILE VIDEOTRK
TRANS1/YYMD=20050104;
TRANS2/YYMD=20050302;

DATEW/W=TRANS1     ;
DATEW2/W=TRANS2    ;
DATEYYMD/YYMDW=TRANS1    ;
DATEYYMD2/YYMDW=TRANS2   ;

OUT7A/A30=DATETRAN(DATEYYMD , '(DMYY)', '(Bwrctr)', 'GR', 30, 'A30');
OUT7B/A30=DATETRAN(DATEYYMD2, '(DMYY)', '(Bwrctr)', 'GR', 30, 'A30');
END

TABLE FILE VIDEOTRK
HEADING
"FORMAT Bwrctrde"
""
"Blank-delimited (B)"
"Full day of week name at beginning of date, default case (wr)"
"Comma after day name (c)"
"Full month name, default case (tr)"
"Greek"
""
SUM OUT7A AS '' OUT7B AS '' TRANSDATE NOPRINT
ON TABLE HOLD FORMAT HTML
ON TABLE SET PAGE-NUM OFF
END
```

The output is:



Reference: Usage Notes for the DATETRAN Function

- The output field, though it must be type A and not AnV, may in fact contain variable length information, since the lengths of month names and day names can vary, and also month and day numbers may be either one or two bytes long if a zero-suppression option is chosen. Unused bytes are filled with blanks.
- All invalid and inconsistent inputs result in all blank output strings. Missing data also results in blank output.
- The base dates (1900-12-31 and 1900-12 or 1901-01) are treated as though the DATEDISPLAY setting were ON (that is, not automatically shown as blanks). To suppress the printing of base dates, which have an internal integer value of 0, test for 0 before calling DATETRAN. For example:

```
RESULT/A40 = IF DATE EQ 0 THEN ' ' ELSE
                DATETRAN (DATE, '(YYMD)', '(.t)', 'FR', 40, 'A40');
```

- Valid translated date components are contained in files named DTLNG*lng* where *lng* is a three-character code that specifies the language. These files must be accessible for each language into which you want to translate dates.
- For these NLS characters to appear correctly, the WebFOCUS Server and Client must be configured with the correct code pages.
- The DATETRAN function is not supported in Dialogue Manager.

HADD: Incrementing a Date-Time Value

How to:

Increment a Date-Time Value

The HADD function increments a date-time value by a given number of units.

Syntax: How to Increment a Date-Time Value

```
HADD(timestamp, component, increment, length, output_format)
```

where

timestamp

Date-Time

Is the timestamp to be incremented.

component

Alphanumeric

Is the name of the component to be incremented. For a list of valid components, see *Date and Time Components For Date and Date-Time Functions* on page 89.

increment

Integer

Is the number of units, positive or negative, by which to increment the component.

length

Integer

Is the number of characters returned as the timestamp. Valid values are:

`8` for a timestamp in date-time format that includes milliseconds.
`10` for a timestamp in date-time format that includes microseconds.

output_format

Date-Time

Example: Incrementing a Date-Time Value

The following example increments thirty months to some specific date-time in the past

```
HADD(DT(2001/09/11 08:54:34), 'MONTH', 30, 8, 'HYYMDS')
```

and returns the timestamp 2004/03/11 08:54:34.00.

HCNVRT: Converting a Date-Time Value to Alphanumeric Format

How to:

Convert a Date-Time Value to Alphanumeric Format

The HCNVRT function converts a date-time value to alphanumeric format for use with operators such as EDIT, CONTAINS, and LIKE.

Syntax: How to Convert a Date-Time Value to Alphanumeric Format

```
HCNVRT(timestamp,(format), length, output_format)
```

where:

timestamp

Date-Time

Is the value to be converted.

format

Alphanumeric

Is the format of the timestamp enclosed in parentheses. It must be a date-time format (H format).

length

Integer

Is the number of characters in the alphanumeric field that is returned. If the length is smaller than the number of characters needed to display the alphanumeric field, the function returns a blank.

output_format

Alphanumeric

Example: Converting a Date-Time Value to Alphanumeric Format

Assume that you have a date-time field DTCUR in H format. To convert this timestamp to an alphanumeric string, use the following syntax:

```
HCNVRT(DTCUR, '(HMDYYS)', 20, 'A20')
```

The function returns the string '03/26/2004 14:25:58' that is assignable to an alphanumeric variable.

HDATE: Converting the Date Portion of a Timestamp Value to a Date Format

How to:

Convert the Date Portion of a Timestamp Value to a Date Format

The HDATE function converts the date portion of a date-time value to the date format YYMD. You can then convert the result to other date formats.

Syntax: How to Convert the Date Portion of a Timestamp Value to a Date Format

```
HDATE(timestamp, output_format)
```

where:

timestamp

Date-Time

Is the value to be converted.

output_format

Date

Is the date of any full component date format or just a full component date format in single quotation marks. This output_format can be assigned to any other date field (whether or not it is a full component), or to a numeric field.

Example: Converting the Date Portion of a Timestamp Value to a Date Format

This example converts the DTCUR field, which is the current date/time timestamp, into a date field using the format DMY:

```
MYDATE/DMY = HDATE(DTCUR, 'YYMD');
```

The function returns the date in format YYMD, then assigns it to MYDATE after conversion to its format MY as 03/04. Note that the output_format of HDATE is presented as a full component date format MDYY, as required.

HDIFF: Finding the Number of Units Between Two Date-Time Values

How to:

Find the Number of Units Between Two Date-Time Values

The HDIFF function calculates the number of date or time component units between two timestamps.

Syntax: How to Find the Number of Units Between Two Date-Time Values

```
HDIFF(end_timestamp, start_timestamp, component, output_format)
```

where:

end_timestamp

Date-Time

Is the timestamp value to subtract from.

start_timestamp

Date-Time

Is the timestamp value to subtract.

component

Alphanumeric

Is the name of the component to be incremented. For a list of valid components, see *Date and Time Components For Date and Date-Time Functions* on page 89. If the component is a week, the WEEKFIRST parameter setting is used in the calculation.

output_format

Floating-point double precision

Example: Finding the Number of Units Between Two Date-Time Values

Assume that we have a date-time field DTCUR in H format, which is has a current date and time timestamp. To find the number of days from President's Day 2004 to today use the expression:

```
DIFDAY/I6 = HDIF(DTCUR, DT(2004/02/16), 'DAY', 'D6.0')
```

The function returns the number of days in double precision floating point format, then assigns it to DIFDAY as integer value. If today is March 31, 2004, the DIFDAY is assigned to 46.

If you wish to obtain results in seconds, use the expression

```
DIFSEC/I9 = HDIF(DTCUR, DT(2004 February 16), 'SECOND', 'D9.0')
```

which assigns 3801600 to DIFSEC. Note that the format 'D9.0' is used with HDIF. Using 'I9' for an output_format in HDIF is invalid.

HDTTM: Converting a Date to a Timestamp

How to:

Convert a Date to a Timestamp

The HDTTM function converts a date value to a timestamp value. The time portion or timestamp is set to midnight.

Syntax: How to Convert a Date to a Timestamp

```
HDTTM(date, length, output_format)
```

where:

date

Date

Is the date to be converted. It must be a full component format date. For example, it can be MDYY or YYJUL.

length

Integer

Is the number of characters returned as the timestamp. Valid values are:
`8` for a timestamp in date-time format that includes milliseconds.
`10` for a timestamp in date-time format that includes microseconds.

output_format

Date-Time

Is the newly created timestamp.

Example: Converting a Date to a Timestamp

This example converts the President's Day date into a timestamp:

```
TS/HYYMDS = HDTTM('February 16 2004', 8, TS)
```

the function returns 2004/02/16 00:00:00 and assigns this timestamp to field TS. Note the zero values of time components in the timestamp. Also note the use of natural date constants in single quotation marks for the date in the first function parameter.

HGETC: Storing the Current Date and Time as a Timestamp

How to:

Store the Current Date and Time as a Timestamp

The HGETC function returns the current date and time as a timestamp in the desired date-time format. If millisecond or microsecond values are not available in your operating environment, the function retrieves the value zero for these components.

Syntax: How to Store the Current Date and Time as a Timestamp

```
HGETC(length, output_format)
```

where:

length

Integer

Is the number of characters returned as a timestamp. Valid values are:

`8` for a timestamp in date-time format that includes milliseconds.
`10` for a timestamp in date-time format that includes microseconds.

output_format

Date-Time

Is the newly created timestamp.

Example: Storing the Current Date and Time as a Timestamp

This example,

```
HGETC(8, 'HYYMDS')
```

creates a timestamp representing the current date and time.

HHMMSS: Retrieving the Current Time

How to:

Retrieve the Current Time

The HHMMSS function retrieves the current time from the operating system as an eight character string, separating the hours, minutes, and seconds with periods.

Syntax: How to Retrieve the Current Time

```
HHMMSS(output_format)
```

where:

output_format

Alphanumeric, at least A8

Note: While not actually a date-time function, HHMMSS included here for your convenience.

Example: Retrieving the Current Time

This example,

```
HMMSS('A10')
```

creates a character string representing current time, like 12.09.47. Note that shorter output_format format will cause truncation of output.

HINPUT: Converting an Alphanumeric String to a Timestamp

How to:

Convert an Alphanumeric String to a Timestamp

The HINPUT function converts an alphanumeric string to a timestamp in date-time format.

Syntax: How to Convert an Alphanumeric String to a Timestamp

```
HINPUT(source_length, source_string, timestamp_length, output_format)
```

where:

source_length

Integer

Is the number of characters in the source_string.

source_string

Alphanumeric

Is the string to be converted. The string can consist of any valid date-time input value, as described in *Date and Time Components For Date and Date-Time Functions* on page 89.

timestamp_length

Integer

Is the number of characters returned as the timestamp. Valid values are:

`8` for a timestamp in date-time format that includes milliseconds.
`10` for a timestamp in date-time format that includes microseconds.

`output_format`

Date-Time

Is the newly created timestamp.

Example: Converting an Alphanumeric String to a Timestamp

This example,

```
DTM/HYYMDS = HINPUT(14, '20040229 13:34:00', 8, DTM);
```

converts the character string (20040229 13:34:00) into a timestamp, which is then assigned to the date-time field DTM. DTM is displayed as 2004/02/29 13:34:00.

HMIDNT: Setting the Time Portion of a Timestamp to Midnight

How to:

Set the Time Portion of a Timestamp to Midnight

The HMIDNT function changes the time portion of a timestamp in date-time format to midnight (all zeroes by default). This allows you to compare a date field with a timestamp in date-time format.

Syntax: How to Set the Time Portion of a Timestamp to Midnight

`HMIDNT(`*`timestamp, length, output_format`*`)`

where:

`timestamp`

Date-Time

Is the timestamp whose time is to be set to midnight.

`length`

Integer

Is the number of characters returned as the timestamp. Valid values are:

`8` for a timestamp in date-time format that includes milliseconds.
`10` for a timestamp in date-time format that includes microseconds.

output_format

Date-Time

Is the timestamp whose time is set to midnight and whose date is copied from timestamp.

Example: Setting the Time Portion of a Timestamp to Midnight

This example converts the character string (20040229 13:34:00) to a timestamp, which is assigned to DTM:

```
DTM/HYYMDS = HINPUT(14, '20040229 13:34:00', 8, DTM);
```

This example resets the time portion of DTM to midnight and assigned the timestamp (02/29/2004 00:00:00) to DTMIDNT:

```
DTMIDNT/HMDYYS = HMIDNT(DTM, 8, DTMIDNT);
```

HMASK: Extracting Components of a Date-Time Field and Preserving Remaining Components

How to:

Move Multiple Date-Time Components to a Target Date-Time Field

Reference:

Usage Notes for the HMASK Function

The HMASK function extracts one or more components from a date-time value and moves them to a target date-time field with all other components of the target field preserved.

Syntax: How to Move Multiple Date-Time Components to a Target Date-Time Field

```
HMASK(source, 'componentstring', input, length, output_format)
```

where:

source

Is the date-time value from which the specified components are extracted.

componentstring

Is a string of codes, in any order, that indicates which components are to be extracted and moved to the output date-time field. The following table shows the valid values. The string is considered to be terminated by any character not in this list:

Code	Description
C	century (the two high-order digits only of the four-digit year)
Y	year (the two low-order digits only of the four-digit year)
YY	Four digit year.
M	month
D	day
H	hour
I	minutes
S	seconds
s	milliseconds (the three high-order digits of the six-digit microseconds value)
u	microseconds (the three low-order digits of the six-digit microseconds value)
m	All six digits of the microseconds value.

input

Is the date-time value that provides all the components for the output that are not specified in the component string.

length

Is the length of the returned date-time value. Valid values are:

`8` indicates a time value that includes milliseconds.
`10` indicates a time value the includes microseconds.

output_format

Is the format of the output value enclosed in single quotation marks. This field must be in date-time format (data type H).

Example: Changing a Date-Time Field Using HMASK

The VIDEOTRK data source has a date-time field named TRANSDATE of format HYYMDI. The following request changes any TRANSDATE value with a time component greater than 11:00 to 8:30 of the following day. First the HEXTR function extracts the minute and seconds portion of the value and compares it to 11:00. If it is greater than 11:00, the HADD function calls HMASK to change the time to 08:30 and adds one day to the date:

```
DEFINE FILE VIDEOTR2
ORIG_TRANSDATE/HYYMDI = TRANSDATE;
TRANSDATE =
IF HEXTR(TRANSDATE, 'HI', 8, 'HHI') GT DT(11:00)
   THEN HADD (HMASK(DT(08:30), 'HISs', TRANSDATE, 8, 'HYYMDI'), 'DAY',
    1,8, 'HYYMDI')
   ELSE TRANSDATE;
END

TABLE FILE VIDEOTR2
PRINT ORIG_TRANSDATE TRANSDATE
BY LASTNAME
BY FIRSTNAME
WHERE ORIG_TRANSDATE NE TRANSDATE
END
```

The output is

```
LASTNAME        FIRSTNAME   ORIG_TRANSDATE    TRANSDATE
--------        ---------   --------------    ---------
BERTAL          MARCIA      1999/07/29 12:19  1999/07/30 08:30
DIZON           JANET       1999/11/05 11:12  1999/11/06 08:30
GARCIA          JOANN       1998/05/08 12:48  1998/05/09 08:30
                            1999/11/30 12:12  1999/12/01 08:30
GRANT           WESTON      1998/10/30 11:12  1998/10/31 08:30
GREEVEN         GEORGIA     1999/04/29 11:28  1999/04/30 08:30
HARRIS          JESSICA     1997/03/28 11:12  1997/03/29 08:30
JOSEPH          JAMES       1999/05/06 11:55  1999/05/07 08:30
KURTZ           DAVID       1998/09/22 11:58  1998/09/23 08:30
MONROE          PATRICK     1999/09/23 11:48  1999/09/24 08:30
PARKER          GLENDA      1999/01/06 12:22  1999/01/07 08:30
RATHER          MICHAEL     1998/02/28 12:33  1998/03/01 08:30
SPIVEY          TOM         1991/11/17 11:28  1991/11/18 08:30
WILSON          KELLY       1999/06/26 12:34  1999/06/27 08:30
WU              MARTHA      1997/12/11 11:48  1997/12/12 08:30
```

Reference: Usage Notes for the HMASK Function

HMASK processing is subject to the DTSTRICT setting. Moving the day (D) component without the month (M) component could lead to an invalid result, which is not permitted if the DTSTRICT setting is ON. Invalid date-time values cause any date-time function to return zeroes.

HNAME: Retrieving a Timestamp Date or Time Component as Alphanumeric Value

How to:

Retrieve a Timestamp Date or Time Component as Alphanumeric Value

The HNAME function extracts a specified component value from a timestamp, in date-time format, and returns it as digits, in alphanumeric format.

Syntax: How to Retrieve a Timestamp Date or Time Component as Alphanumeric Value

```
HNAME(timestamp, component, output_format)
```

where:

timestamp

Date-Time

Is the timestamp from which a component value is to be extracted.

component

Alphanumeric

Is the name of the component to be extracted.

For a list of valid components, see *Date and Time Components For Date and Date-Time Functions* on page 89.

output_format

Alphanumeric, at least A2

The function converts a component value to a string of digits. The year is always four digits, and the hour assumes the 24-hour system.

Example: Retrieving a Timestamp Date or Time Component as an Alphanumeric Value

Assuming that the current time obtained by the function HGETC in the first parameter is 13:22:11, this example returns the string '13' and assigns it to AHOUR:

```
AHOUR/A2 = HNAME(HGETC(8,'HYYMDS'),'HOUR', AHOUR);
```

HPART: Retrieving a Timestamp Date or Time Component as Numeric Value

How to:

Retrieve a Timestamp Date or Time Component as a Numeric Value

The HPART function extracts a specified component value from a timestamp, in date-time format, and returns it in numeric format.

Syntax: How to Retrieve a Timestamp Date or Time Component as a Numeric Value

```
HPART(value, component, output_format)
```

where:

timestamp

Date-Time

Is the timestamp from which a component value is to be extracted.

component

Alphanumeric

Is the name of the component to be incremented. For a list of valid components, see *Date and Time Components For Date and Date-Time Functions* on page 89.

output_format

Integer

Example: Retrieving a Timestamp Date or Time Component as Numeric Value

Assuming that the current time obtained by HGETC in the first parameter is 14:01:39, this example returns a whole number, 14, and assigns it to IHOUR:

```
IHOUR/I2 = HPART(HGETC(8,'HYYMDS'),'HOUR', IHOUR);
```

HSETPT: Inserting a Component Into a Date-Time Value

How to:

Insert a Component Into a Date-Time Value

The HSETPT function inserts the numeric value of a specified component into a date-time value.

Syntax: How to Insert a Component Into a Date-Time Value

```
HSETPT(timestamp, component, value, length, output_format)
```

where:

timestamp

Date-Time

Is the timestamp into which a component value is to be inserted.

component

Alphanumeric

Is the name of the component to be incremented.

For a list of valid components, see *Date and Time Components For Date and Date-Time Functions* on page 89.

value

Integer

Is the numeric value to be inserted for the requested component.

length

Integer

Is the number of characters returned as a timestamp. Valid values are:

`8` for a timestamp in date-time format that includes milliseconds.
`10` for a timestamp in date-time format that includes microseconds.

output_format

Date-Time

Is the timestamp whose chosen component is updated. All other components are copied from the timestamp.

Example: Inserting a Component Into a Date-Time Value

Assuming that the current date and time obtained by HGETC in the first parameter are 03/31/2004 and 13:34:36, this example,

```
UHOUR/HMDYYS = HSETPT(HGETC(8,'HYYMDS'),'HOUR', 7, 8, UHOUR);
```

returns 03/31/2004 07:34:36.

HTIME: Converting the Time Portion of a Date-Time Value to a Number

How to:

Convert the Time Portion of a Date-Time Value to a Number

The HTIME function converts the time portion of a timestamp in date-time format to the number of milliseconds if the first argument is 8, or microseconds if the first argument is 10. To include microseconds, the input timestamp format must be a 10-byte date-time format.

Syntax: How to Convert the Time Portion of a Date-Time Value to a Number

```
HTIME(length, timestamp, output_format)
```

where:

length

Integer

Is the number of characters returned as a timestamp. Valid values are:

8 for a timestamp in date-time format that includes milliseconds.
10 for a timestamp in date-time format that includes microseconds.

timestamp

Date-Time

Is the timestamp from which to convert the time.

output_format

Is floating-point double precision format.

Example: Converting the Time Portion of a Date-Time Value to a Number

Assuming that the current date and time obtained by HGETC in the second parameter are 03/31/2004 and 13:48:14, this example returns and assigns to NMILLI, 49,694,395. (Note that this example uses milliseconds rather than microseconds.)

```
NMILLI/D12.0 = HTIME(8, HGETC(10,'HYYMDS'), NMICRO);
```

Assuming that the first parameter is equal to 10 and the timestamp format is HYYMDSS, this example returns and assigns to NMICRO, 50,686,123,024.

```
NMICRO/D12.0 = HTIME(10, HGETC(10,'HYYMDSS'), NMICRO);
```

HTMTOTS: Converting a Time to a Timestamp

How to:

Convert a Time to a Timestamp

The HTMTOTS function returns a timestamp using the current date to supply the date components of its value, and copies the time components from its input timestamp.

Syntax: How to Convert a Time to a Timestamp

```
HTMTOTS(time, length, output_format)
```

where:

time

Date-Time

Is the timestamp whose time will be used. The date portion will be ignored.

length

Integer

Is the length of the result. This can be one of the following:

`8` for input time values including milliseconds.
`10` for input time values including microseconds.

output_format

Date-Time

Is the timestamp whose date is set to current date, and whose time is copied from time.

Example: Converting a Time to a Timestamp

This example produces a timestamp, whose date and time are current, and stores the result in a column with the format in the field HMDYYS:

```
HMDYYS = HTMTOTS(DT(&MYTOD), 8, 'HMDYYS');
```

The result is 03/26/2004 13:48:14.

HYYWD: Returning the Year and Week Number From a Date-time Value

How to:

Return the Year and Week Number From a Date-time Value

The week number returned by HNAME and HPART can actually be in the year preceding or following the input date.

The HYYWD function returns both the year and the week number from a given date-time value.

The output is edited to conform to the ISO standard format for dates with week numbers, yyyy-Www-d.

Syntax: How to Return the Year and Week Number From a Date-time Value

```
HYYWD(dtvalue, outfield)
```

where:

dtvalue

Date-time

Is the date-time value to be edited, the name of a date-time field that contains the value, or an expression that returns the value.

outfield

Alphanumeric

Is the field that contains the result, or the format of the output value enclosed in single quotation marks.

The field must be at least 10 characters long. The output is in the following format:

```
yyyy-Www-d
```

where:

yyyy

Is the four-digit year.

ww

Is the two-digit week number (01 to 53).

d

Is the single-digit day of the week (1 to 7). The d value is relative to the current WEEKFIRST setting. If WEEKFIRST is 2 or ISO2 (Monday), then Monday is represented in the output as 1, Tuesday as 2.

Using the EDIT function, you can extract the individual subfields from this output.

Example: Returning the Year and Week Number From a Date-time Value

The following request against the VIDEOTR2 data source calls HYYWD to convert the TRANSDATE date-time field to the ISO standard format for dates with week numbers. WEEKFIRST is set to ISO2, which produces ISO standard week numbering:

```
SET WEEKFIRST = ISO2
TABLE FILE VIDEOTR2
SUM TRANSTOT QUANTITY
COMPUTE ISODATE/A10 = HYYWD(TRANSDATE, 'A10');
BY TRANSDATE
WHERE QUANTITY GT 1
END
```

The output is:

```
TRANSDATE          TRANSTOT  QUANTITY  ISODATE
---------          --------  --------  -------
1991/06/24 04:43     16.00         2  1991-W26-1
1991/06/25 01:17      2.50         2  1991-W26-2
1991/06/27 02:45     16.00         2  1991-W26-4
1996/08/17 05:11      5.18         2  1996-W33-6
1998/02/04 04:11     12.00         2  1998-W06-3
1999/01/30 04:16     13.00         2  1999-W04-6
1999/04/22 06:19      3.75         3  1999-W16-4
1999/05/06 05:14      1.00         2  1999-W18-4
1999/08/09 03:17     15.00         2  1999-W32-1
1999/09/09 09:18     14.00         2  1999-W36-4
1999/10/16 09:11      5.18         2  1999-W41-6
1999/11/05 11:12      2.50         2  1999-W44-5
1999/12/09 09:47      5.18         2  1999-W49-4
1999/12/15 04:04      2.50         2  1999-W50-3
```

Example: Extracting a Component From a Date Returned by HYYWD

The following request against the VIDEOTR2 data source calls HYYWD to convert the TRANSDATE date-time field to the ISO standard format for dates with week numbers. It then uses the EDIT function to extract the week component from this date. WEEKFIRST is set to ISO2, which produces ISO standard week numbering:

```
SET WEEKFIRST = ISO2
TABLE FILE VIDEOTR2
SUM TRANSTOT QUANTITY
COMPUTE ISODATE/A10 = HYYWD(TRANSDATE, 'A10');
COMPUTE WEEK/A2 = EDIT(ISODATE, '$$$$$$99$$');
BY TRANSDATE
WHERE QUANTITY GT 1 AND DATE EQ 1991
END
```

The output is:

```
TRANSDATE         TRANSTOT  QUANTITY  ISODATE     WEEK
---------         --------  --------  -------     ----
1991/06/24 04:43     16.00         2  1991-W26-1  26
1991/06/25 01:17      2.50         2  1991-W26-2  26
1991/06/27 02:45     16.00         2  1991-W26-4  26
```

Legacy Date Functions

The functions described in this section are legacy date functions. They were created for use with dates in integer or alphanumeric format. They are no longer recommended for date manipulation. Standard date and date-time functions are preferred.

The legacy date functions are:

- *TODAY: Returning the Current Date*
- *AYM: Adding or Subtracting Months To or From a Date*
- *AYMD: Adding or Subtracting Days To or From a Date*
- *CHGDAT: Changing the Format of a Date*
- *DA Functions: Converting a Legacy Date to an Integer*
- *DMY, MDY, YMD: Calculating the Difference Between Two Dates*
- *DOWK and DOWKL: Finding the Day of the Week*
- *DT Functions: Converting an Integer to a Date*
- *GREGDT: Converting From Julian to Gregorian Format*
- *JULDAT: Converting From Gregorian to Julian Format*

- *YM: Calculating Elapsed Months*

TODAY: Returning the Current Date

How to:

Return the Current Date

The TODAY function retrieves the current date from the operating system in the format MM/DD/YY or MM/DD/YYYY. You can remove the default embedded slashes in the date using the EDIT function. The DATE function always returns a date that is current. Therefore, if you are running an application late at night, you may want to use TODAY.

You can also retrieve the date in the format MM/DD/YY or MM/DD/YYYY using the Dialogue Manager system variable &DATE. You can retrieve the date without the slashes using the system variables &YMD, &MDY, and &DMY. The system variable &DATEfmt retrieves the date in a specified format.

Syntax: How to Return the Current Date

```
TODAY(output_format)
```

where:

output_format

Alphanumeric, at least A8

Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.

The following rules apply:

- If DATEFNS=ON and the format is A8 or A9, TODAY returns the 2-digit year.
- If DATEFNS=ON and the format is A10 or greater, TODAY returns the 4-digit year.
- If DATEFNS=OFF, TODAY returns the 2-digit year, regardless of the format of output_format.

Tip: TODAY is not actually a legacy date function. It is included here for your convenience. The preferred way to obtain the current date is by using the system variables &YYMD and related variables.

Example: Retrieving the Current Date

TODAY retrieves the current date and stores it in a column with the format A10.

```
TODAY('A10')
```

AYM: Adding or Subtracting Months To or From a Date

How to:

Add or Subtract Months To or From Dates

The AYM function adds months to or subtracts months from a date in year-month format. If necessary, you can convert a date to year-moth format using the CHGDAT or EDIT function.

Syntax: How to Add or Subtract Months To or From Dates

```
AYM(indate, months, output_format)
```

where:

indate

I4, I4YM, I6 or I6YYM

Is the legacy date in integer year-month format. If the date is not valid, the function returns a 0.

months

Integer

Is the number of months you are adding to or subtracting from the date. To subtract months, use a negative number.

output_format

I4YM or I6YYM

Is the resulting legacy date.

Tip: If the input date is in integer year-month-day format (I6YMD or I8YYMD), divide the date by 100 to convert to year-month format and set the result to an integer. This drops the day portion of the date.

Example: Adding Months to a Date

AYM adds six months to HIRE_MONTH and stores the result in a column with the format I4YM.

```
AYM(HIRE_MONTH, 6, 'I4YM')
```

For 99/04, the result is 99/10.

For 98/11, the result is 99/05.

AYMD: Adding or Subtracting Days To or From a Date

How to:

Add or Subtract Days to or From a Date

The AYMD function adds days to or subtracts days from a date in year-month-day format. You can convert a date to this format using the CHGDAT or EDIT function.

Syntax: How to Add or Subtract Days to or From a Date

```
AYMD(indate, days, output_format)
```

where:

indate

I6, I6YMD, I8, I8YYMD

Is the legacy date in year-month-day format. If the date is not valid, the function returns 0.

days

Integer

Is the number of days you are adding to or subtracting from the indate. To subtract days, use a negative number.

output_format

I6, I6YMD, I8, or I8YYMD

Is the same format as the indate. If the addition or subtraction of days crosses forward or backward into another century, the century digits of the output year are adjusted.

Example: Adding Days to a Date

AYMD adds 35 days to each value in the HIRE_DATE field, and stores the result in a column with the format I6YMD.

```
AYMD(HIRE_DATE, 35, 'I6YMD')
```

For 99/08/01, the result is 99/09/05.

For 99/01/04, the result is 99/02/08.

CHGDAT: Changing the Format of a Date

How to:

Change the Format of a Date

Reference:

Short to Long Conversion

The CHGDAT function rearranges the year, month, and day portions of an input character string representing a date. It may also convert the input string from long to short or short to long date representation. Long representation contains all three date components: year, month, and day; short representation omits one or two components, such as year, month, or day. The input and output date strings are described by display options that specify both the order of components (year, month, day) in the date string and whether two or four digits are used for the year (for example, 04 or 2004). CHGDAT reads an input date character string and creates an output date character string that represents the same date in a different way.

Note: CHGDAT requires a date character string as input, not a date itself. Whether the input is a standard or legacy date, convert it to a date character string (using the EDIT or DATECVT functions, for example) before applying CHGDAT.

The order of date components in the date character string is described by display options comprised of the following characters in your chosen order:

Character	Description
D	Day of the month (01 through 31).
M	Month of the year (01 through 12).
Y[Y]	Year. Y indicates a two-digit year (such as 04); YY indicates a four-digit year (such as 2004).

To spell out the month rather than use a number in the resulting string, append one of the following characters to the display options for the resulting string:

Character	Description
T	Displays the month as a three-letter abbreviation.
X	Displays the full name of the month.

Display options can consist of up to five display characters. Characters other than those display options are ignored.

For example: The display options 'DMYY' specify that the date string starts with a two-digit day, followed by a two-digit month and a four-digit year.

Note: Display options are *not* date formats.

Reference: Short to Long Conversion

If you are converting a date from short to long representation (for example, from year-month to year-month-day), the function supplies the portion of the date missing in the short representation, as shown in the following table.

Portion of Date Missing	Portion Supplied by Function
Day (for example, from YM to YMD)	Last day of the month.
Month (for example, from Y to YM)	Last month of the year (December).
Year (for example, from MD to YMD)	The year 99.
Converting year from two-digit to four-digit (for example, from YMD to YYMD)	If DATEFNS=ON, the century will be determined by the 100-year window defined by DEFCENT and YRTHRESH. If DATEFNS=OFF, the year 19xx is supplied, where xx is the last two digits in the year.

Syntax: How to Change the Format of a Date

```
CHGDAT('in_display_options','out_display_options', date_string,
output_format)
```

where:

in_display_options

A1 to A5

Is a series of up to five display options that describe the layout of *date_string*. These options can be stored in an alphanumeric field or supplied as a literal enclosed in single quotation marks.

out_display_options

A1 to A5

Is a series of up to five display options that describe the layout of the converted date string. These options can be stored in an alphanumeric field or supplied as a literal enclosed in single quotation marks.

date_string

A2 to A8

Is the input date character string with date components in the order specified by *in_display_options*.

Note that if the original date is in numeric format, you must convert it to a date character string. If *date_string* does not correctly represent the date (the date is invalid), the function returns blank spaces.

output_format

A*xx*, where *xx* is a number of characters large enough to fit the date string specified by *out_display_options*. A17 is long enough to fit the longest date string.

Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.

Note: Since CHGDAT uses a date string (as opposed to a date) and returns a date string with up to 17 characters, use the EDIT or DATECVT functions or any other means to convert the date to or from a date character string.

Example: Converting the Date Display From YMD to MDYYX

ALPHA_HIRE is HIRE_DATE converted from numeric to alphanumeric format. CHGDAT converts each value in ALPHA_HIRE from displaying the components as YMD to MDYYX and stores the result in a column with the format A17. The option X in the output value displays the full name of the month.

```
CHGDAT('YMD', 'MDYYX', ALPHA_HIRE, 'A17')
```

DA Functions: Converting a Legacy Date to an Integer

How to:

Convert a Legacy Date to an Integer

The DA functions convert a legacy date to the number of days between a specific day in the past, called 'base date', and a specified date. By converting a date to the number of days, you can subtract dates and calculate the intervals between them, or you can add to or subtract numbers from the dates to get new dates.

You can convert the result of a DA function back to a legacy date using the DT functions.

Syntax: How to Convert a Legacy Date to an Integer

```
function(indate, output_format)
```

where:

function

Is one of the following:

DADMY converts a date in day-month-year format.
DADYM converts a date in day-year-month format.
DAMDY converts a date in month-day-year format.
DAMYD converts a date in month-year-day format.
DAYDM converts a date in year-day-month format.
DAYMD converts a date in year-month-day format.

indate

I6xxx, where xxx corresponds to the function DAxxx in the list above.

Is the legacy date to be converted. If indate is a numeric literal, enter only the last two digits of the year; the function assumes the century component. If the date is invalid, the function returns a 0.

output_format

Integer

Example: Converting Dates and Calculating the Difference Between Them

DAYMD converts DAT_INC and HIRE_DATE to the number of days since December 31, 1899 and the smaller number is then subtracted from the larger number:

```
DAYMD(DAT_INC, 'I8') - DAYMD(HIRE_DATE, 'I8')
```

DMY, MDY, YMD: Calculating the Difference Between Two Dates

How to:

Calculate the Difference Between Two Dates

The DMY, MDY, and YMD functions calculate the difference between two legacy dates in numeric format.

Syntax: How to Calculate the Difference Between Two Dates

```
function(from_date, to_date)
```

where:

function

Is one of the following:

DMY calculates the difference between two dates in day-month-year format.
MDY calculates the difference between two dates in month-day-year format.
YMD calculates the difference between two dates in year-month-day format.

from_date

I6xxx or I8xxx, where xxx corresponds to the specified function (DMY, YMD, or MDY).

Is the beginning legacy date.

to_date

I6xxx or I8xxx where xxx corresponds to the specified function (DMY, YMD, or MDY).

Is the end legacy date.

Example: Calculating the Number of Days Between Two Dates

YMD calculates the number of days between the dates in HIRE_DATE and DAT_INC.

```
YMD(HIRE_DATE, DAT_INC)
```

DOWK and DOWKL: Finding the Day of the Week

How to:

Find the Day of the Week

The DOWK and DOWKL functions find the day of the week that corresponds to a date. DOWK returns the day as a three-letter abbreviation; DOWKL displays the full name of the day.

Syntax: How to Find the Day of the Week

```
{DOWK|DOWKL}(indate, output_format)
```

where:

indate

I6YMD or I8YMD

Is the legacy date. If the date is not valid, the function returns spaces. If indate is specified in I6YMD format and DEFCENT and YRTHRESH values have not been set, the function assumes the 20th century.

output_format

DOWK: A4, DOWKL: A12

Example: Finding the Day of the Week

DOWK determines the day of the week that corresponds to the value in the HIRE_DATE field and stores the result in a column with the format A4.

```
DOWK(HIRE_DATE, 'A4')
```

For 80/06/02, the result is MON.

For 82/08/01, the result is SUN.

DT Functions: Converting an Integer to a Date

How to:

Convert an Integer to a Date

There are six DT functions; each one converts a number into a date of a different format.

Syntax: How to Convert an Integer to a Date

```
function(number, output_format)
```

where:

function

Is one of the following:

DTDMY converts a number to a day-month-year date.
DTDYM converts a number to a day-year-month date.
DTMDY converts a number to a month-day-year date.
DTMYD converts a number to a month-year-day date.
DTYDM converts a number to a year-day-month date.
DTYMD converts a number to a year-month-day date.

number

Integer

Is the number of days since 'base' date, possibly received from the functions DAxxx.

output_format

I8xxx, where xxx corresponds to the function DTxxx, for example I8DTDMY.

The output format is determined by the specified function (that is, DTDMY, DTDYM, DTMDY, DTMYD, DTYDM, or DTYMD).

Example: Converting an Integer to a Date

DTMDY converts NEWF (which was converted to the number of days by DAYMD) to the corresponding date and stores the result in a column with the format I8MDYY.

```
DTMDY(NEWF, 'I8MDYY')
```

For 81/11/02, the result is 11/02/1981.

For 82/05/01, the result is 05/01/1982.

GREGDT: Converting From Julian to Gregorian Format

How to:

Convert From Julian to Gregorian Format

The GREGDT function converts a date in Julian format (year-number_of_the_day) to Gregorian format (year-month-day). A date in Julian format is a five- or seven-digit number. The first two or four digits are the year; the last three digits are the number of the day, counting from January 1. For example, January 1, 1999 in Julian format is either 99001 or 1999001; June21, 2004 in Julian format is 2004173.

Syntax: How to Convert From Julian to Gregorian Format

```
GREGDT(indate, output_format)
```

where:

indate

I5 or I7

Is the Julian date. If the date is invalid, the function returns a 0.

output_format

I6, I8, I6YMD or I8YYMD

Example: Converting From Julian to Gregorian Format

GREGDT converts JULIAN to YYMD (Gregorian) format. It determines the century using the default DEFCENT and YRTHRESH parameter settings. The result is stored in a column with the format I8.

```
GREGDT(JULIAN, 'I8')
```

For 82213, the result is 19820801.

For 82004, the result is 19820104.

JULDAT: Converting From Gregorian to Julian Format

How to:

Convert From Gregorian to Julian Format

The JULDAT function converts a date from Gregorian format (year-month-day) to Julian format (year-number_of_the_day). A date in Julian format is a five- or seven-digit number. The first two or four digits are the year; the last three digits are the number of the day, counting from January 1. For example, January 1, 1999 in Julian format is either 99001 or 1999001.

Syntax: How to Convert From Gregorian to Julian Format

```
JULDAT(indate, output_format)
```

where:

indate

I6, I8, I6YMD or I8YYMD

Is the legacy date to convert.

output_format

I5 or I7

Example: Converting From Gregorian to Julian Format

JULDAT converts the HIRE_DATE field to Julian format. It determines the century using the default DEFCENT and YRTHRESH parameter settings. The result is stored in a column with the format I7.

```
JULDAT(HIRE_DATE, 'I7')
```

For 82/08/01, the result is 1982213.

For 82/01/04, the result is 1982004.

YM: Calculating Elapsed Months

How to:

Calculate Elapsed Months

The YM function calculates the number of months that elapse between two dates. The dates must be in year-month format. You can convert a date to this format by using the CHGDAT or EDIT function.

Syntax: How to Calculate Elapsed Months

```
YM(from_date, to_date, output_format)
```

where:

from_date

I4YM or I6YYM

Is the start date. If the date is not valid, the function returns a 0.

todate

I4YM or I6YYM

Is the end date. If the date is not valid, the function returns a 0.

output_format

Integer

Tip: If from_date or todate is in integer year-month-day format (I6YMD or I8YYMD), simply divide by 100 to convert to year-month format and set the result to an integer. This drops the day portion of the date.

Example: Calculating Elapsed Months

YM calculates the difference between HIRE_MONTH and MONTH_INC and stores the results in a column with the format I3.

```
YM(HIRE_MONTH, MONTH_INC, 'I3')
```

6 Format Conversion Functions

Format conversion functions convert fields from one format to another.

Topics:

ATODBL: Converting an Alphanumeric String to Double-Precision Format

How to:

Convert an Alphanumeric String to Double-Precision Format

The ATODBL function converts a number in alphanumeric format to decimal (double-precision) format.

Syntax: How to Convert an Alphanumeric String to Double-Precision Format

```
ATODBL(source_string, length, output_format)
```

where:

source_string

Alphanumeric

Is the string consisting of digits and, optionally, one decimal point to be converted.

length

Alphanumeric

Is the length of the source_string in bytes. This can be a numeric constant, or a field or variable that contains the value. If you specify a numeric constant, enclose it in single quotation marks, for example '12'. The length can be from 1 to 15 characters.

output_format

Double precision floating-point

Example: Converting an Alphanumeric Field to Double-Precision Format

ATODBL converts EMP_ID into double-precision format.

```
ATODBL(EMP_ID, '09', 'D12.2')
```

For 112847612, the result is 112,847,612.00.

For 117593129, the result is 117,593,129.00.

EDIT: Converting the Format of a Field

How to:

Convert the Format of a Field

The EDIT function converts an alphanumeric field that contains numeric characters to a numeric value or a numeric field to a character string.

This function for manipulating a field in an expression that uses an operation which requires operands in a particular format.

When EDIT output is assigned to a new field, the format of the latter must correspond to the format of the returned value. For example, if EDIT converts a numeric field to a character string, you must give the new field an alphanumeric format.

EDIT deals with symbols of the source value in the following ways:

- When an alphanumeric field is converted to a numeric value, a sign or decimal point in the field is stored as part of the numeric value.
- When converting a floating-point or packed-decimal field to a character string, EDIT removes the sign, the decimal point, and all digits to the right of the decimal point. It then right-justifies the remaining digits and adds leading zeros to achieve the specified field length. Converting a number with more than nine significant digits in floating-point or packed-decimal format may produce an incorrect result.

Syntax: How to Convert the Format of a Field

```
EDIT(fieldname)
```

where

fieldname

Alphanumeric or Numeric

Is the field name.

Note: EDIT can also be used for extracting and adding characters from the source_string after converting a numeric fieldname. For details, see *EDIT: Extracting or Adding Characters* on page 40.

Example: Converting From Numeric to Alphanumeric Format

EDIT converts HIRE_DATE (a legacy date format) to alphanumeric format.

```
EDIT(HIRE_DATE)
```

For 82/04/01, the result it APRIL 01 1982.

For 81/11/02, the result it NOVEMBER 02 1981.

FTOA: Converting a Number to Alphanumeric Format

How to:

Convert a Number to Alphanumeric Format

The FTOA function converts a number up to 16 digits from numeric format to alphanumeric format. It retains the decimal positions of the number and right-justifies it with leading spaces. You can add edit options to a number converted by FTOA.

When using FTOA to convert a number containing decimals to a character string, you must specify an alphanumeric format large enough to accommodate both the integer and decimal portions of the number. For example, a D12.2 format will be converted to A14. If the output format is not large enough, decimals are truncated.

Syntax: How to Convert a Number to Alphanumeric Format

```
FTOA(number, '(format)', output_format)
```

where:

number

Numeric F or D (single and double precision floating-point)

Is the number to be converted.

format

Alphanumeric

Is the format of the number to be converted.

Parentheses are required around the format.

output_format

Alphanumeric

Example: Converting From Numeric to Alphanumeric Format

FTOA converts GROSS from floating point double-precision to alphanumeric format.

```
FTOA(GROSS, '(D12.2)', 'A15')
```

For $1,815.00, the result is 1,815.00.

For $2,255.00, the result is 2,255.00.

HEXBYT: Converting a Decimal Value to a Character

How to:

Convert a Decimal Value to a Character

The HEXBYT function obtains the ASCII or EBCDIC character equivalent of a decimal value. It returns a single alphanumeric character in the ASCII or EBCDIC character set. You can use this function to produce characters that, although printable, are not on your keyboard. If the characters you use are not printable, use the CTRAN function to convert them to printable characters.

The display of special characters is platform dependent; not all special characters will display. Printable EBCDIC or ASCII characters and their decimal equivalents are listed in *Character Chart for ASCII and EBCDIC* on page 17.

Syntax: How to Convert a Decimal Value to a Character

```
HEXBYT(decimal_value, output)
```

where:

decimal_value

Integer

Is the value that is assumed to be an ASCII or EBCDIC code for the character to be presented. A value greater than 255 is treated as the remainder of the decimal_value divided by *256*.

output_format

Alphanumeric

Example: Converting a Decimal Integer to a Character

HEXBYT converts LAST_INIT_CODE to its character equivalent and stores the result in in a column with the format A1.

```
HEXBYT(LAST_INIT_CODE, 'A1')
```

On an ASCII platform, for 83, the result is S.

On ASCII platform, for 74, the result is J.

ITONUM: Converting a Large Number from Integer to Double-Precision Format

How to:

Convert a Large Number From Integer to Double-Precision Format

The ITONUM function converts a large number in a non-FOCUS data source from special long integer to double-precision format. This is useful for some programming languages and some non-FOCUS data storage systems that use special long integers, which do not fit the regular integer format (four bytes in length) supported in the synonym, and, therefore, require conversion to double-precision format.

You must specify how many of the right-most bytes in the input field are significant. The result is an 8-byte double-precision field.

Syntax: How to Convert a Large Number From Integer to Double-Precision Format

```
ITONUM(maxbytes, infield, output_format)
```

where:

maxbytes

Numeric

Is the maximum number of bytes in the 8-byte infield that have significant digits, including the binary sign. Valid values are:

5 ignores the 3 left-most bytes.
6 ignores the 2 left-most bytes.
7 ignores the left-most byte.

infield

A8

Is the field that contains the number. Both the USAGE and ACTUAL formats of the field must be A8.

output_format

Double precision floating-point (D*n*.*d*)

Example: Converting a Large Binary Integer to Double-Precision Format

ITONUM converts BINARYFLD to double-precision format.

```
ITONUM(6, BINARYFLD, 'D14')
```

ITOPACK: Converting a Large Binary Integer to Packed-Decimal Format

How to:

Convert a Large Binary Integer to Packed-Decimal Format

The ITOPACK function converts a large number in a non-FOCUS data source from special long integer to double-precision format. This is useful for some programming languages and some non-FOCUS data storage systems that use special long integers, which do not fit the regular integer format (four bytes in length) supported in the synonym, and, therefore, require conversion to packed-decimal format.

You must specify how many of the right-most bytes in the input field are significant. The result is an 8-byte packed-decimal field of up to 15 significant numeric positions (for example, P15 or P16.2).

Limit: For a field defined as 'PIC 9(15) COMP' or the equivalent (15 significant digits), the maximum number that can be converted is 167,744,242,712,576.

Syntax: How to Convert a Large Binary Integer to Packed-Decimal Format

```
ITOPACK(maxbytes, infield, output_format)
```

where:

maxbytes

Numeric

Is the maximum number of bytes in the 8-byte infield that have significant numeric data, including the binary sign. Valid values are:

5 ignores the 3 left-most bytes (up to 11 significant positions).
6 ignores the 2 left-most bytes (up to 14 significant positions).
7 ignores the left-most byte (up to 15 significant positions).

infield

A8

Is the field that contains the binary number. Both the USAGE and ACTUAL formats of the field must be A8.

output_format

Numeric

Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. The format must be P*n* or P*n.d*.

Example: Converting a Large Binary Integer to Packed-Decimal Format

ITOPACK converts BINARYFLD to packed-decimal format.

```
ITOPACK(6, BINARYFLD, 'P14.4')
```

ITOZ: Converting a Number to Zoned Format

How to:

Convert a Number to Zoned Format

The ITOZ function converts a number in numeric format to zoned-decimal format. Although a request cannot process zoned numbers, it can write zoned fields to an extract file for use by an external program.

Syntax: How to Convert a Number to Zoned Format

```
ITOZ(length, in_value, output_format)
```

where:

length

Integer

Is the number of bytes in the in_value. The maximum number of bytes is 15. The last byte includes the sign.

in_value

Numeric

Is the number to be converted. The number is truncated to an integer before it is converted.

output_format

Alphanumeric

Example: Converting a Number to Zoned Format

ITOZ converts CURR_SAL to zoned format.

```
ITOZ(8, CURR_SAL, 'A8')
```

PCKOUT: Writing a Packed Number of Variable Length

How to:

Write a Packed Number of Variable Length

The PCKOUT function writes a packed-decimal number of variable length to an extract file. When a request saves a packed-decimal number to an extract file, it typically writes it as an 8- or 16-byte field, regardless of its format specification. With PCKOUT, you can vary the field's length from 1 to 16 bytes.

Syntax: How to Write a Packed Number of Variable Length

```
PCKOUT(in_value, length, output_format)
```

where:

in_value

Numeric

Is the input value. It can be in packed, integer, single- or double-precision floating-point format. If it is not in integer format, the value is rounded to the nearest whole number.

length

Numeric

Is the number of bytes in the output_format, from 1 to 16.

output_format

Alphanumeric

This function returns the field as alphanumeric, although it contains packed data.

Example: Writing a Packed Number of Variable Length

PCKOUT converts CURR_SAL to a five-byte packed format.

```
PCKOUT(CURR_SAL, 5, 'A5')
```

PTOA: Converting a Number to Alphanumeric Format

How to:

Convert a Number to Alphanumeric Format

The PTOA function converts a number from numeric format to alphanumeric format. It retains the decimal positions of a number and right-justifies it with leading spaces. You can add edit options to a number converted by PTOA.

PTOA is similar to FTOA, however PTOA converts numbers with 17format to 31 digits.

When using PTOA to convert a number containing decimals to a character string, you must specify an alphanumeric format large enough to accommodate both the integer and decimal portions of the number. For example, a P18.2 format is converted to A20. If the output format is not large enough, decimals are truncated.

Syntax: How to Convert a Number to Alphanumeric Format

```
PTOA(number, '(format)', output_format)
```

where:

number

Numeric P (Packed Decimal) or F or D (single and double precision floating-point)

Is the number to be converted.

format

Alphanumeric

Is the format of the number to be converted enclosed in parentheses.

Parentheses are required around the format.

output_format

Alphanumeric

Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. The length of this argument must be greater than the length of the number and must account for edit options and a possible negative sign.

Example: Converting From Packed to Alphanumeric Format

PTOA converts PGROSS from packed-decimal to alphanumeric format.

```
PTOA(PGROSS, FMT, 'A17')
```

UFMT: Converting an Alphanumeric String to Hexadecimal

How to:

Convert an Alphanumeric String to Hexadecimal

The UFMT function converts characters in the source_string to their hexadecimal representation. This function is useful for examining data of unknown format. As long as you know the length of the data, you can examine its content.

Syntax: How to Convert an Alphanumeric String to Hexadecimal

```
UFMT(source_string, length, output_format)
```

where:

source_string

Alphanumeric

Is the string to convert.

length

Integer

Is the number of characters in the source_string.

output_format

Alphanumeric

The format of the output_format must be alphanumeric and its length must be twice the length of the source string.

Example: Converting an Alphanumeric String to Hexadecimal

UFMT converts each value in JOBCODE to its hexadecimal representation and stores it in a column with the format A6.

```
UFMT(JOBCODE, 3, 'A6')
```

For A01, the result is C1F0F1.

For A02, the result is C1F0F2.

XTPACK: Writing a Packed Number With Up to 31 Significant Digits to an Output File

How to:

Store Packed Values in an Alphanumeric Field

The XTPACK function stores packed numbers with up to 31 significant digits in an alphanumeric field, retaining decimal data. This permits writing a short or long packed field of any length, 1 to 16 bytes, to an output file.

Syntax: How to Store Packed Values in an Alphanumeric Field

```
XTPACK (infield, outlength, outdec, outfield)
```

where:

infield

Numeric

Is the field that contains the packed value.

outlength

Numeric

Is the length of the alphanumeric field that will hold the converted packed field. Can be from 1 to 16.

outdec

Numeric

Is the number of decimal positions for *outfield*.

outfield

Alphanumeric

Is the name of the field to contain the result or the format of the field enclosed in single quotation marks.

Example: Writing a Long Packed Number to an Output File

The following request creates a long packed decimal field named LONGPCK . ALPHAPCK (format A13) is the result of applying XTPACK to the long packed field. PCT_INC, LONGPCK, and ALPHAPCK are then written to a SAVE file named XTOUT.

```
DEFINE FILE EMPLOYEE
LONGPCK/P25.2 = PCT_INC + 11111111111111111111;
ALPHAPCK/A13 = XTPACK(LONGPCK,13,2,'A13');
END
TABLE FILE EMPLOYEE
PRINT PCT_INC LONGPCK ALPHAPCK
WHERE PCT_INC GT 0
  ON TABLE SAVE AS XTOUT
END
```

The SAVE file has the following fields and formats:

```
ALPHANUMERIC RECORD NAMED  XTOUT
FIELDNAME                          ALIAS         FORMAT        LENGTH

PCT_INC                            PI            F6.2            6
LONGPCK                                          P25.2          25
ALPHAPCK                                         A13            13

TOTAL                                                           44
SAVED...
```

7 Numeric Functions

Numeric functions perform calculations on numeric constants and fields.

Topics:

ABS: Calculating Absolute Value

How to:

Calculate Absolute Value

The ABS function returns the absolute value of a number.

Syntax: How to Calculate Absolute Value

```
ABS(in_value)
```

where:

in_value

Numeric

Is the value for which the absolute value is returned. If you use an expression, use parentheses as needed to ensure the correct order of evaluation.

Example: Calculating Absolute Value

ABS calculates the absolute value of DIFF.

```
ABS(DIFF);
```

For 15, the result is 15.

For -2, the result is 2.

CHKPCK: Validating a Packed Field

How to:

Validate a Packed Field

The CHKPCK function validates the data in a field described as packed-decimal format (if available on your platform). The function prevents a data exception from occurring when a request reads a field that is expected to contain a valid packed number but does not.

To use CHKPCK:

1. Ensure that the synonym (USAGE and ACTUAL attributes) defines the field as alphanumeric, not packed. This does *not* change the field data, which remains packed, but it enables the request to read the data without a data exception.

2. Call CHKPCK to examine the field. The function returns the output to a field defined as packed. If the value it examines is a valid packed number, the function returns the value; if the value is not packed, the function returns an error code.

Syntax: How to Validate a Packed Field

```
CHKPCK(length, in_value, error, output_format)
```

where:

length

Numeric

Is the number of bytes in the packed-decimal field. The length can be between 1 and 16 bytes.

in_value

Alphanumeric

Is the value to be verified as packed decimal. The in_value is intentionally described as alphanumeric, not packed.

error

Numeric

Is the error code that the function returns if a value is not in packed-decimal format. Choose an error code outside the range of data. The error code is first truncated to an integer, then converted to packed format. However, it may appear on a report with a decimal point because of the format of the output field.

output_format

Packed-decimal

Example: Validating Packed Data

CHKPCK validates the values in PACK_SAL, and store the result in a column with the format P8CM. Values not in packed format return the error code -999. Values in packed format appear accurately.

```
CHKPCK(8, PACK_SAL, -999, 'P8CM')
```

DMOD, FMOD, and IMOD: Calculating the Remainder From a Division

How to:

Calculate the Remainder From a Division

The MOD functions calculate the remainder from a division. Each function returns the remainder in a different format.

The functions use the following formula.

```
remainder = dividend - INT(dividend/divisor) * divisor
```

- DMOD returns the remainder as a decimal number.
- FMOD returns the remainder as a floating-point number.
- IMOD returns the remainder as an integer.

Syntax: How to Calculate the Remainder From a Division

```
function(dividend, divisor, output_format)
```

where:

function

Is one of the following:

DMOD returns the remainder as a decimal number.
FMOD returns the remainder as a floating-point number.
IMOD returns the remainder as an integer.

dividend

Numeric

Is the number being divided.

divisor

Numeric

Is the number dividing the dividend.

output_format

Numeric

Is the result, whose format is determined by the function used.

Example: Calculating the Remainder From a Division

IMOD divides ACCTNUMBER by 1000 and stores the remainder in a column with the format I3L.

```
IMOD(ACCTNUMBER, 1000, 'I3L')
```

For 122850108, the result is 108.

For 163800144, the result is 144.

EXP: Raising 'e' to the Nth Power

How to:

Raise 'e' to the Nth Power

The EXP function raises the value "e" (approximately 2.72) to a specified power. This function is the inverse of the LOG function, which returns an argument's logarithm.

EXP calculates the result by adding terms of an infinite series. If a term adds less than .000001 percent to the sum, the function ends the calculation and returns the result as a double-precision number.

Syntax: How to Raise 'e' to the Nth Power

```
EXP(power, output_format)
```

where:

power

Numeric

Is the power that "e" is raised to.

output_format

Double-precision floating-point

Example: Raising 'e' to the Nth Power

EXP raises "e" to the power designated by the &POW variable, specified here as 3. The result is then rounded to the nearest integer with the .5 rounding constant. The result has the format D15.3.

```
EXP(&POW, 'D15.3') + 0.5;
```

For 3, the result is APPROXIMATELY 20.

EXPN: Evaluating a Number in Scientific Notation

How to:

Evaluate a Number in Scientific Notation

The EXPN function evaluates a number expressed in scientific notation.

Syntax: How to Evaluate a Number in Scientific Notation

```
EXPN(n.nn {E|D} {+|-} p)
```

where:

n.nn

Numeric

Is a numeric constant that consists of a whole number component, followed by a decimal point, followed by a fractional component.

E, D

Denotes scientific notation. E and D are interchangeable.

+, -

Indicates if p is positive or negative.

p

Integer

Is the power of 10 to which to raise n.nn.

Note: EXPN does not use an output_format. The format of the result is floating-point double precision.

Example: Evaluating a Number in Scientific Notation

EXPN evaluates SCI_DATA.

```
EXPN(SCI_DATA)
```

For 1.03E+2, the result is 103.

INT: Finding the Greatest Integer

How to:

Find the Greatest Integer

The INT function returns the integer component of a number.

Syntax: How to Find the Greatest Integer

`INT(in_value)`

where:

in_value

Numeric

Is the value for which the integer component is returned. If you supply an expression, use parentheses as needed to ensure the correct order of evaluation.

Note: INT does not use an output_format. The format of the result is floating-point double precision.

Example: Finding the Greatest Integer

INT finds the greatest integer in DED_AMT.

```
INT(DED_AMT)
```

For $1,261.40, the result is 1261.

For $1,668.69, the result is 1668.

LOG: Calculating the Natural Logarithm

How to:

Calculate the Natural Logarithm

The LOG function returns the natural logarithm of a number.

Syntax: How to Calculate the Natural Logarithm

`LOG(in_value)`

where:

in_value

Numeric

Is the value for which the natural logarithm is calculated. If you supply an expression, use parentheses as needed to ensure the correct order of evaluation. If the argument is less than or equal to 0, LOG returns 0.

Note: LOG does not use an output_format. The format of the result is floating-point double precision.

Example: Calculating the Natural Logarithm

LOG calculates the logarithm of CURR_SAL.

```
LOG(CURR_SAL)
```

For $29,700.00, the result is 10.30.

For $26,862.00, the result is 10.20.

MAX and MIN: Finding the Maximum or Minimum Value

How to:

Find the Maximum or Minimum Value

The MAX and MIN functions return the maximum or minimum value, respectively, from a list of values.

Syntax: How to Find the Maximum or Minimum Value

```
{MAX|MIN}(value1, value2, ...)
```

where:

MAX

Returns the maximum value.

MIN

Returns the minimum value.

value1, value2

Numeric

Are the values for which the maximum or minimum value is returned, the name of a field that contains the values, or an expression that returns the values. If you supply an expression, use parentheses as needed to ensure the correct order of evaluation.

Note: MAX and MIN do not use an output_format. The format of the result is floating-point double precision.

Example: Determining the Minimum Value

MIN returns either the value of ED_HRS or the constant 30, whichever is lower.

```
MIN(ED_HRS, 30)
```

For 45.00, the result is 30.00.

For 25.00, the result is 25.00.

NORMSDST: Calculating Standard Cumulative Normal Distribution

How to:

Calculate the Cumulative Standard Normal Distribution Function

Reference:

Characteristics of the Normal Distribution

The NORMSDST function performs calculations on a standard normal distribution curve, calculating the percentage of data values that are less than or equal to a normalized value. A normalized value is a point on the X-axis of a standard normal distribution curve in standard deviations from the mean. This is useful for determining percentiles in normally distributed data.

The NORMSINV function is the inverse of NORMSDST.

The results of NORMSDST are returned as double-precision and are accurate to 6 significant digits.

A standard normal distribution curve is a normal distribution that has a mean of 0 and a standard deviation of 1. The total area under this curve is 1. A point on the X-axis of the standard normal distribution is called a normalized value. Assuming that your data is normally distributed, you can convert a data point to a normalized value to find the percentage of scores that are less than or equal to the raw score.

You can convert a value (raw score) from your normally distributed data to the equivalent normalized value (z-score) as follows:

```
z = (raw_score - mean)/standard_deviation
```

To convert from a z-score back to a raw score, use the following formula:

```
raw_score = z * standard_deviation + mean
```

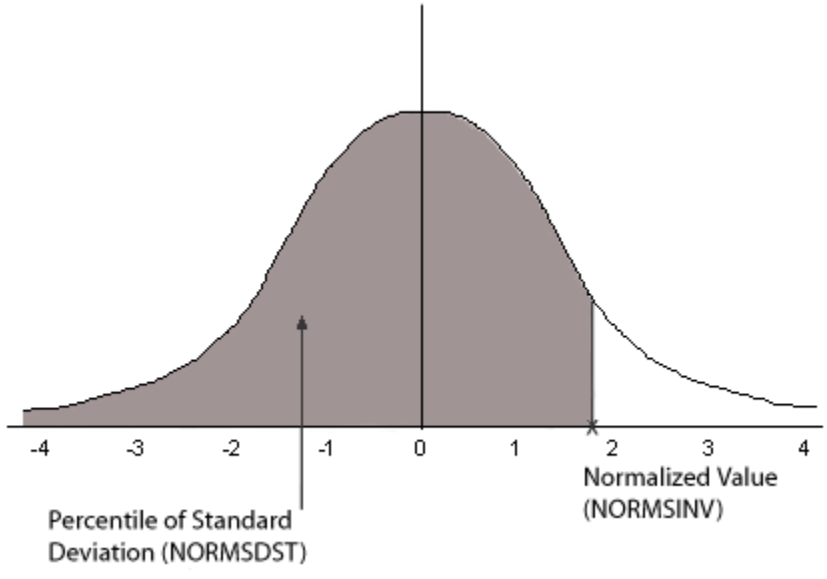
The mean of data points xi, where i is from 1 to n is:

$(\Sigma x_i)/n$

The standard deviation of data points xi, where i is from 1 to n is:

$SQRT(((\Sigma x_i^2 - (\Sigma x_i)^2/n)/(n - 1)))$

The following diagram illustrates the results of the NORMSDST and NORMSINV functions.



Reference: Characteristics of the Normal Distribution

Many common measurements are normally distributed. A plot of normally distributed data values approximates a bell-shaped curve. The two measures required to describe any normal distribution are the mean and the standard deviation:

- The mean is the point at the center of the curve.
- The standard deviation describes the spread of the curve. It is the distance from the mean to the point of inflection (where the curve changes direction).

Syntax: How to Calculate the Cumulative Standard Normal Distribution Function

```
NORMSDST(value, 'D8');
```

where:

value

Is a normalized value.

D8

Is the required format for the result. The value returned by the function is double-precision. You can assign it to a field with any valid numeric format.

Example: Using the NORMSDST Function

NORMSDST finds the percentile for Z and stores the result in a column with the format D8.

```
NORMSDST(Z, 'D8')
```

For -.07298, the result is .47091.

For -.80273 the result is .21106.

NORMSINV: Calculating Inverse Cumulative Normal Distribution

How to:

Calculate the Inverse Cumulative Standard Normal Distribution Function

The NORMSINV function performs calculations on a standard normal distribution curve, finding the normalized value that forms the upper boundary of a percentile in a standard normal distribution curve. This is the inverse of NORMSDST.

The results of NORMSINV are returned as double-precision and are accurate to 6 significant digits.

Syntax: How to Calculate the Inverse Cumulative Standard Normal Distribution Function

```
NORMSINV(value, 'D8');NORMSINV function;numeric functions:NORMSINV;
```

where:

value

Is a number between 0 and 1 (which represents a percentile in a standard normal distribution).

D8

Is the required format for the result. The value returned by the function is double-precision. You can assign it to a field with any valid numeric format.

Example: Using the NORMSINV Function

NORMSINV returns a normalized value from a percentile found using NORMSDST.

```
NORMSINV(NORMSD, 'D8')
```

For .21106, the result is -.80273.

For .47091, the result is -.07298

PRDNOR and PRDUNI: Generating Reproducible Random Numbers

How to:

Generate Reproducible Random Numbers

The PRDNOR and PRDUNI functions generate reproducible random numbers.

- PRDNOR generates reproducible double-precision random numbers normally distributed with an arithmetic mean of 0 and a standard deviation of 1.
- PRDUNI generates reproducible double-precision random numbers uniformly distributed between 0 and 1 (that is, any random number it generates has an equal probability of being anywhere between 0 and 1).

Syntax: How to Generate Reproducible Random Numbers

```
{PRDNOR|PRDUNI}(seed, output_format)
```

where:

`PRDNOR`

Generates reproducible double-precision random numbers normally distributed with an arithmetic mean of 0 and a standard deviation of 1.

`PRDUNI`

Generates reproducible double-precision random numbers uniformly distributed between 0 and 1.

`seed`

Numeric

Is the seed or the field that contains the seed, up to nine bytes. The seed is truncated to an integer. Using the same seed always produces the same set of numbers.

`output_format`

Double-precision

Example: Generating Reproducible Random Numbers

PRDNOR assigns random numbers and stores them in a column with the format D12.2.

```
PRDNOR(40, 'D12.2')
```

RDNORM and RDUNIF: Generating Random Numbers

How to:

Generate Random Numbers

The RDNORM and RDUNIF functions generate random numbers.

- RDNORM generates double-precision random numbers normally distributed with an arithmetic mean of 0 and a standard deviation of 1.
- RDUNIF generates double-precision random numbers uniformly distributed between 0 and 1 (that is, any random number it generates has an equal probability of being anywhere between 0 and 1).

Syntax: How to Generate Random Numbers

```
{RDNORM|RDUNIF}(output_format)
```

where:

`RDNORM`

Generates double-precision random numbers normally distributed with an arithmetic mean of 0 and a standard deviation of 1.

`RDUNIF`

Generates double-precision random numbers uniformly distributed between 0 and 1.

`output_format`

Double-precision

Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.

Example: Generating Random Numbers

RDNORM assigns random numbers and stores them in a column with the format D12.2.

```
RDNORM('D12.2')
```

SQRT: Calculating the Square Root

How to:

Calculate the Square Root

The SQRT function calculates the square root of a number.

Syntax: How to Calculate the Square Root

```
SQRT(in_value)
```

where:

in_value

Numeric

Is the value for which the square root is calculated. If you supply an expression, use parentheses as needed to ensure the correct order of evaluation. If you supply a negative number, the result will be zero.

Note: MAX does not use an output_format. The result's format is floating-point double precision.

Example: Calculating the Square Root

SQRT calculates the square root of LISTPR.

```
SQRT(LISTPR)
```

For 19.98, the result is 4.47.

For 14.98, the result is 3.87.

8 System Functions

System functions call the operating system to obtain information about the operating environment or to use a system service.

Topics:

CLSDDREC: Close All Files Opened by the PUTDDREC Function

How to:

Close All Files Opened by the PUTDDREC Function

The CLSDDREC function closes all files opened by the PUTDDREC function. If PUTDDREC is called in a Dialogue Manager -SET command, the files opened by PUTDDREC are not closed automatically until the end of a request or connection. In this case, you can close the files and free the memory used to store information about open file by calling the CLSDDREC function.

Syntax: How to Close All Files Opened by the PUTDDREC Function

```
CLSDDREC(output_format)
```

where:

output_format

Integer

Is the return code, which can be one of the following values:

0 - Files are closed.
1 - Error while closing the files.

Example: Closing Files Opened by the PUTDDREC Function

This example closes files opened by the PUTDDREC function:

```
CLSDDREC('I1')
```

FEXERR: Retrieving an Error Message

How to:

Retrieve an Error Message

The FEXERR function retrieves an Information Builders error message. It is especially useful in a procedure using a command that suppresses the display of output messages.

An error message consists of up to four lines of text; the first line contains the message and the remaining three contain a detailed explanation, if one exists. FEXERR retrieves the first line of the error message.

Syntax: How to Retrieve an Error Message

```
FEXERR(error, 'A72')
```

where:

error

Numeric

Is the error number, up to five digits long.

output_format

Alphanumeric, A72

Is the format of the output value whose length is 72. The maximum length of an Information Builders error message is 72 characters.

Example: Retrieving an Error Message

FEXERR retrieves the error message whose number is contained in the &ERR variable, in this case 650. The result has the format A72.

```
FEXERR(&ERR, 'A72')
```

The result is (FOC650) THE DISK IS NOT ACCESSED.

FGETENV: Retrieving the Value of an Environment Variable

How to:

Retrieve the Value of an Environment Variable

The FGETENV function retrieves the value of an operating system environment variable and returns it as an alphanumeric string.

Syntax: How to Retrieve the Value of an Environment Variable

```
FGETENV(length, varname, outlen, output_format)
```

where:

length

Integer

Is the number of characters in the environment variable, varname.

varname

Alphanumeric

Is the name of the environment variable whose value is being retrieved.

outlen

Integer

Is the length of the environment variable value that is returned.

output_format

Alphanumeric

Example: Retrieving the Value of an Environment Variable

This example,

```
FGETENV(6, 'WINDIR', 16, 'A16')
```

returns C:\WINDOWS by default.

This example,

```
FGETENV(9, 'EDAEXTSEC', 3, 'A3')
```

returns ON, if the iWay server's security is on.

FPUTENV: Assigning a Value to an Environment Variable

How to:

Assign Value to an Environment Variable

The FPUTENV function assigns a character string to an operating system environment variable.

Limit: You cannot use FPUTENV to set or change FOCPRINT, FOCPATH, or USERPATH. Once started, these variables are held in memory and will not be re-read from the environment even if you change it.

Syntax: How to Assign Value to an Environment Variable

```
FPUTENV(varname_length, varname, value_length, value, output_format)
```

where:

varname_length

Integer

Is the number of characters in the environment variable, varname.

varname

Alphanumeric

Is the name of the environment variable. The name must be right-justified and padded with blanks to the maximum length specified by varname_length.

value_length

Alphanumeric

Is the maximum length of the environment variable value.

value

Alphanumeric

Is the value you want to assign to the environment variable, varname.

output_format

Integer

Is the return code. If varname is set successfully, the return code is 0; any other value indicates that a failure occurred.

Note: The sum of varname_length and value_length cannot exceed 64 characters.

Example: Assigning a Value to an Environment Variable

FPUTENV assigns the value FOCUS/Shell to the PS1 variable and stores it in a column with the format A12.

```
FPUTENV(3,'PS1', 12 'FOCUS/Shell:', 'A12')
```

The request displays the following prompt when the user issues the UNIX shell command SH:

```
FOCUS/Shell:
```

GETUSER: Retrieving a User ID

How to:

Retrieve a User ID

The GETUSER function retrieves the ID of the connected user.

Syntax: How to Retrieve a User ID

```
GETUSER(output_format)
```

where:

output_format

Alphanumeric, at least A8

Is the result field, whose length depends on the platform on which the function is issued. Provide a length as long as required for your platform; otherwise the output will be truncated.

Example: Retrieving a User ID

GETUSER retrieves the user ID of the person running the flow.

```
GETUSER(USERID)
```

PUTDDREC: Write a Character String as a Record in a Sequential File

How to:

Write a Character String as a Record in a Sequential File

The PUTDDREC function writes a character string as a record in a sequential file. The file must be identified with a FILEDEF (DYNAM on MVS) command. If the file is defined as an existing file (with the APPEND option), the new record is appended. If the file is defined as NEW and it already exists, the new record overwrites the existing file.

PUTDDREC opens the file if it is not already open. Each call to PUTDDREC can use the same file or a new one. All of the files opened by PUTDDREC remain open until the end of a request or connection,. At the end of the request or connection, all files opened by PUTDDREC are automatically closed.

If PUTDDREC is called in a Dialogue Manager -SET command, the files opened by PUTDDREC are not closed automatically until the end of a request or connection. In this case, you can close the files and free the memory used to store information about open file by calling the CLSDDREC function.

Syntax: How to Write a Character String as a Record in a Sequential File

```
PUTDDREC(ddname, dd_len, record_string, record_len, output_format)
```

where:

ddname

Alphanumeric

Is the logical name assigned to the sequential file in a FILEDEF command.

dd_len

Numeric

Is the number of characters in the logical name.

record_string

Alphanumeric

Is the character string to be added as the new record in the sequential file.

record_len

Numeric

Is the number of characters to add as the new record.

output_format

Integer

Is the return code, which can have one of the following values:

`0` - Record is added.
`-1` - FILEDEF statement is not found.
`-2` - Error while opening the file.
`-3` - Error while adding the record to the file.

Example: **Writing a Character String as a Record in a Sequential File**

Using the CAR synonym as input,

```
FILEDEF LOGGING DISK baseapp/logging.dat
PUTDDREC('LOGGING', 7, 'Country:' | COUNTRY, 20, 'I5')
```

would return the value 0, and would write the following lines to logging.dat:

Country: ENGLAND

Country: JAPAN

Country: ITALY

Country: W GERMANY

Country: FRANCE

9 SQL Character Functions

SQL character functions manipulate alphanumeric fields and character strings.

Topics:

CHAR_LENGTH: Finding the Length of a Character String

How to:

Find the Length of a Character String

The CHAR_LENGTH function returns the length of a character string. CHARACTER_LENGTH is identical to CHAR_LENGTH.

This function is most useful for columns described as VARCHAR—that is, variable length character. For example, if a column described as GLOSS VARCHAR(10) contains

```
'bryllig'
'slythy '
'toves  '
```

then CHAR_LENGTH(GLOSS) would return

```
7
6
5
```

If the column is described as CHAR—that is, non-variable length character—the same number is returned for all rows. In this case, CHAR_LENGTH(GLOSS) would return

```
10
10
10
```

To avoid counting trailing blanks use CHAR_LENGTH(TRIM (TRAILING FROM GLOSS)). See *TRIM: Removing Leading or Trailing Characters (SQL)* on page 188 for details.

Syntax: How to Find the Length of a Character String

```
CHAR_LENGTH(arg)
```

where:

arg

Character string

Is the value whose length is to be determined.

This function returns an integer value.

Example: Finding the Length of a Character String

CHAR_LENGTH finds the length of the string. This example,

```
CHAR_LENGTH('abcdef')
```

returns 6.

This example,

```
CHAR_LENGTH('abcdef   ')
```

returns 9, since trailing blanks are counted.

CONCAT: Concatenating Two Character Strings

> **How to:**
>
> Concatenate Two Character Strings

The CONCAT function concatenates the values of two arguments. The result is a character string consisting of the characters of the first argument followed by the characters of the second argument.

Syntax: How to Concatenate Two Character Strings

```
CONCAT(arg1, arg2)
```

where:

arg1, arg2

Character strings

Are the strings to be concatenated.

The length of the result is the sum of the lengths of the two arguments. If either argument is variable-length, so is the result; otherwise, the result is fixed-length.

Example: Concatenating Two Character Strings

CONCAT concatenates two string. This example,

```
CONCAT('abc', 'def')
```

returns abcdef.

DIGITS: Converting a Numeric Value to a Character String

How to:

Convert a Numeric Value to a Character String

The DIGITS function extracts the digits of a decimal or integer value into a character string. The sign and decimal point of the number (if present) are ignored.

Note: This function is available only for DB2.

Syntax: How to Convert a Numeric Value to a Character String

```
DIGITS(arg)
```

where:

arg

Numeric (decimal or integer, not floating-point)

Is the numeric value.

The length of the resulting string is determined by the precision of the argument.

Example: Converting a Numeric Value to a Character String

DIGITS converts a numeric value to a character string. This example,

```
DIGITS(-444.321)
```

returns 0000444321.

EDIT: Editing a Value According To a Format (SQL)

How to:

Edit a Value According To a Format

The EDIT function edits a numeric or character value according to a format specified by a mask. (It works exactly like the EDIT function in FOCUS.)

A 9 in the mask indicates the corresponding character in the source value is copied into the result. A $ in the mask indicates that the corresponding character is to be ignored. Any other character is inserted into the result.

Syntax: How to Edit a Value According To a Format

```
EDIT(arg, mask)
```

where:

arg

Numeric or character string

Is the value to be edited.

mask

character string

Indicates how the editing is to proceed.

This function returns a character string whose length is determined by the mask.

Example: Editing a Value According To a Format

EDIT extracts a character from a string. This example,

```
EDIT('FRED'        , '9$$$')
```

returns F.

This example,

```
EDIT('123456789', '999-99-9999')
```

returns 123-45-6789.

LCASE: Converting a Character String to Lower Case

How to:

Convert a Character String to Lower Case

The LCASE function converts a character string value to lower-case. That is, capital letters are replaced by their corresponding lower-case values.

LOWER and LOWERCASE are identical to LCASE.

Syntax: How to Convert a Character String to Lower Case

```
LCASE(arg)
```

where:

arg

character string

Is the value to be converted to lower case.

This function returns a varying character string. The length is the same as the input argument.

Example: Converting a Character String to Lower Case

LCASE converts a character string to lower-case. This example,

```
LCASE('XYZ')
```

returns xyz.

LTRIM: Removing Leading Spaces

How to:

Remove Leading Spaces

The LTRIM function removes leading spaces from a character string.

Syntax: How to Remove Leading Spaces

```
LTRIM(arg)
```

where:

arg

character string

Is the value to be trimmed.

This function returns a varying character string. The data type of the result has a length equal to that of the input argument (although the value may be shorter).

Example: Removing Leading Spaces

LTRIM removes leading spaces. This example,

```
LTRIM('   ABC   ')
```

returns 'ABC   '.

POSITION: Finding the Position of a Substring

How to:

Find the Position of a Substring

The POSITION function returns the position within a character string of a specified substring. If the substring does not appear in the character string, the result is 0. Otherwise, the value returned is one greater than the number of characters in the string preceding the start of the first occurrence of the substring.

Syntax: How to Find the Position of a Substring

```
POSITION(substring IN arg)
```

where:

substring

character string

Is the substring to search for.

arg

character string

Is the string to be searched for the substring.

This function returns an integer value.

Example: Finding the Position of a Substring

POSITION returns the position of a substring. This example,

```
POSITION ('A'    IN 'AEIOU')
```

returns 1.

This example,

```
POSITION ('IOU' IN 'AEIOU')
```

returns 3.

This example,

```
POSITION ('Y'     IN 'AEIOU')
```

returns 0.

RTRIM: Removing Trailing Spaces

How to:

Remove Trailing Spaces

The RTRIM function removes trailing spaces from a character string.

Syntax: How to Remove Trailing Spaces

```
RTRIM(arg)
```

where:

arg

character string

Is the value to be trimmed.

This function returns a varying character string. The data type of the result has a length equal to that of the input argument (although the value may be shorter).

Example: Removing Trailing Spaces

RTRIM removes trailing spaces. This example,

```
RTRIM('   ABC   ')
```

returns ' ABC'.

SUBSTR: Extracting a Substring From a String Value (SQL)

How to:

Extract a Substring From a String Value

The SUBSTR function returns a substring of a character value. You specify the start position of the substring within the value. You can also specify the length of the substring (if omitted, the substring extends from the start position to the end of the string value). If the specified length value is longer than the input string, the result is the full input string.

SUBSTRING is identical to SUBSTR.

Syntax: How to Extract a Substring From a String Value

```
SUBSTR(arg FROM start-pos [FOR length])
```

or

```
SUBSTR(arg, start-pos [, length])
```

where:

arg

character string

Is the field containing the parent character string.

start-pos

Integer

Is the position within *arg* at which the substring begins.

length

Integer

If present, is the length of the substring. This function returns a varying character string. The data type of the result has a length equal to that of the input argument (although the value may be shorter).

Example: **Extracting a Substring From a String Value**

SUBSTR function returns a substring. This example,

```
SUBSTR('ABC' FROM 2)
```

Returns BC.

This example,

```
SUBSTRING('ABC' FROM 1 FOR 2)
```

returns AB.

This example,

```
SUBSTR('ABC', 10)
```

returns ABC.

TRIM: Removing Leading or Trailing Characters (SQL)

How to:

Remove Leading or Trailing Characters

The TRIM function removes leading and/or trailing characters from a character string. The character to be removed may be specified. If no character is specified, the space character is assumed. Whether to remove leading and/or trailing characters may be specified. Without this specification, both leading and trailing appearances of the specified character are removed.

Syntax: How to Remove Leading or Trailing Characters

```
TRIM(arg)
TRIM(trim-where [trim-char] FROM arg)
TRIM(trim-char FROM arg)
```

where:

arg

character string

Is the source string value to be trimmed.

trim-where

Value may be LEADING, TRAILING or BOTH. Indicates where characters will be removed. If not specified, BOTH is assumed.

trim-char

character string

Is the character to be removed. If not specified, the space character is assumed.

This function returns a varying character string. The data type of the result has a length equal to that of the input argument (although the value may be shorter).

Example: Removing Leading or Trailing Characters

TRIM removes leading and/or trailing characters. This example,

```
TRIM('  ABC  ')
```

returns ABC.

This example,

```
TRIM(LEADING FROM '  ABC  ')
```

returns 'ABC '.

This example,

```
TRIM(TRAILING FROM '  ABC  ')
TRIM(BOTH 'X' FROM 'XXYYYXXX') = ('YYY')
```

returns ' ABC'

This example,

```
TRIM(BOTH 'X' FROM 'XXYYYXXX')
```

returns YYY.

UCASE: Converting a Character String to Uppercase

How to:

Convert a Character String to Uppercase

The UCASE function converts a character string value to uppercase. That is, lowercase letters are replaced by their corresponding uppercase values. UPPER and UPPERCASE are identical to UCASE.

Syntax: How to Convert a Character String to Uppercase

```
UCASE(arg)
```

where:

arg

character string

Is the value to be converted to uppercase.

This function returns a character string whose length is the same as that of the input argument.

Example: Converting a Character String to Uppercase

UCASE converts a character string value to uppercase. This example,

```
UCASE('abc')
```

returns ABC.

VARGRAPHIC: Converting to Double-byte Character Data

How to:

Convert to the double-byte character format

The VARGRAPHIC function converts the input value to double-byte character data

Syntax: How to Convert to the double-byte character format

```
VARGRAPHICarg
```

where:

arg

character, graphic, or date

Is the input value.

Note: This function can only be used for DB2 and can only be used with Direct or Automatic Passthru. This function returns the value in double-byte character format.

10 SQL Date and Time Functions

SQL date and time functions perform manipulations on date and time values.

Topics:

CURRENT_DATE: Obtaining the Date

How to:

Obtain the Current Date

The CURRENT_DATE function returns the operating system's current date in the form YYYYMMDD.

Syntax: How to Obtain the Current Date

```
CURRENT_DATE
```

This function returns the date in YYMD format.

Example: Obtaining the Current Date

On August 18, 2005, CURRENT_DATE will return 20050818.

CURRENT_TIME: Obtaining the Time

How to:

Obtain the Current Time

The CURRENT_TIME function returns the operating system's current time in the form HHMMSS. You may specify the number of decimal places for fractions of a second–0, 3, or 6 places. Zero (0) places is the default.

Syntax: How to Obtain the Current Time

```
CURRENT_TIME[(precision)]
```

where:

precision

Integer constant

Is the number of decimal places for fractions of a second. Possible values are 0, 3, and 6.

This function returns the time (format: HHIS if no decimal places; HHISs if 3 decimal places; HHISsm if 6 decimal places).

Example: Obtaining the Current Time

At exactly half past 11 AM:

CURRENT_TIME returns 113000.

CURRENT_TIME(3) returns 113000000.

CURRENT_TIME(6) returns 113000000000.

CURRENT_TIMESTAMP: Obtaining the Timestamp (Date/Time)

How to:

Obtain the Current Timestamp

The CURRENT_TIMESTAMP function returns the operating system's current timestamp (date and time) in the form YYYYMMDDHHMMSS. You may specify the number of decimal places for fractions of a second–0, 3, or 6 places. Six (6) places is the default.

Syntax: How to Obtain the Current Timestamp

```
CURRENT_TIMESTAMP[(precision)]
```

where:

precision

Integer constant

Is the number of decimal places for fractions of a second. Possible values are 0, 3, and 6.

This function returns a timestamp (format: HYYMDS if no decimal places; HYYMDs if 3 decimal places; HYYMDm if 6 decimal places).

Example: Obtaining the Current Timestamp

At 2:11:23 PM on October 9, 2005:

CURRENT_TIMESTAMP returns 20051009141123000000.

CURRENT_TIMESTAMP(0) returns 20051009141123.

CURRENT_TIMESTAMP(3) returns 20051009141123000.

CURRENT_TIMESTAMP(6) returns 20051009141123000000.

DAY: Obtaining the Day of the Month From a Date/Timestamp

How to:

Obtain the Day of the Month From a Date or Timestamp

The DAY function returns the day of the month from a date or timestamp value.

Syntax: **How to Obtain the Day of the Month From a Date or Timestamp**

```
DAY(arg)
```

where:

arg

Date or timestamp

Is the input value.

This function returns an integer value.

Example: **Obtaining the Day of the Month From a Date or Timestamp**

DAY returns the day of the month from a date or timestamp. This example,

```
DAY('1976-07-04')
```

returns 4.

This example,

```
DAY('2001-01-22 10:00:00')
```

returns 22.

DAYS: Obtaining the Number of Days Since January 1, 1900

How to:

Obtain the Number of Days Since January 1, 1900

The DAYS function returns the number of days since January 1, 1900.

Syntax: **How to Obtain the Number of Days Since January 1, 1900**

```
DAYS(arg)
```

where:

arg

Date or timestamp

Is the input argument.

This function returns an integer value.

Example: Obtaining the Number of Days Since January 1, 1900

DAYS returns the number of days since January 1, 1900. This example,

```
DAYS('2000-01-01')
```

returns 36525.

EXTRACT: Obtaining a Datetime Field From Date/Time/Timestamp

How to:

Obtain a Datetime Field From a Date, Time, or Timestamp

The EXTRACT function can be used to obtain the year, month, day of month, hour, minute, second, millisecond, or microsecond component of a date, time, or timestamp value.

Syntax: How to Obtain a Datetime Field From a Date, Time, or Timestamp

```
EXTRACT(field FROM arg)
```

where:

arg

Date, time, or timestamp

Is the input argument.

field

Is the datetime field of interest. Possible values are YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, MILLISECOND and MICROSECOND.

This function returns an integer value.

Note:

- YEAR, MONTH and DAY can be used only if the argument is date or timestamp.
- HOUR, MINUTE, SECOND, MILLISECOND and MICROSECOND can be used only if the argument is time or timestamp.

Example: Obtaining a Datetime Field From a Date, Time, or Timestamp

EXTRACT returns the components of a date, time, or timestamp. This example,

```
EXTRACT(YEAR FROM '2000-01-01')
```

returns 2000.

This example,

```
EXTRACT(HOUR FROM '11:22:33')
```

returns 11.

This example,

```
EXTRACT(MICROSECOND FROM '2000-01-01 11:22:33.456789')
```

returns 456,789.

HOUR: Obtaining the Hour From Time/Timestamp

How to:

Obtain the Hour From a Time or Timestamp

The HOUR function returns the hour field from a time or timestamp value.

Syntax: How to Obtain the Hour From a Time or Timestamp

```
HOUR(arg)
```

where:

arg

Time or timestamp

Is the input value.

This function returns an integer value.

Example: Obtaining the Hour From a Time or Timestamp

HOUR returns the hour from a time or timestamp. This example,

```
HOUR('11:22:33')
```

returns 11.

This example,

```
HOUR('2001-01-22 10:00:00')
```

returns 10.

MICROSECOND: Obtaining Microseconds From Time/Timestamp

How to:

Obtain the Number of Microseconds From a Time or Timestamp

The MICROSECOND function returns the number of microseconds from a time or timestamp value.

Syntax: How to Obtain the Number of Microseconds From a Time or Timestamp

```
MICROSECOND(arg)
```

where:

arg

Time or timestamp

Is the input value.

This function returns an integer value.

Example: Obtaining the Number of Microseconds From a Time or Timestamp

MICROSECOND returns the microseconds from a time or timestamp. This example,

```
MICROSECOND('11:22:33.456789')
```

returns 456,789.

This example,

```
MICROSECOND('2001-01-22 10:00:00')
```

returns 0.

MILLISECOND: Obtaining Milliseconds From Time/Timestamp

How to:

Obtain the Number of Milliseconds From a Time or Timestamp

The MILLISECOND function returns the number of milliseconds from a time or timestamp value.

Syntax: How to Obtain the Number of Milliseconds From a Time or Timestamp

```
MILLISECOND(arg)
```

where:

arg

Time or timestamp

Is the input value.

This function returns an integer value.

Example: Obtaining the Number of Milliseconds From a Time or Timestamp

MILLISECOND returns the number of milliseconds from a time or timestamp. This example,

```
MILLISECOND('11:22:33.456')
```

returns 456.

This example,

```
MILLISECOND('2001-01-22 10:11:12')
```

returns 0.

MINUTE: Obtaining the Minute From Time/Timestamp

How to:

Obtain the Minute From a Time or Timestamp

The MINUTE function returns the number of minutes from a time or timestamp value.

Syntax: How to Obtain the Minute From a Time or Timestamp

```
MINUTE(arg)
```

where:

arg

Time or timestamp

Is the input value.

This function returns an integer value.

Example: Obtaining the Minute From a Time or Timestamp

MINUTE returns the minutes from a time or timestamp. This example,

```
MINUTE('11:22:33')
```

returns 22.

This example,

```
MINUTE('2001-01-22 10:11:12')
```

returns 11.

MONTH: Obtaining the Month From Date/Timestamp

How to:

Obtain the Month From a Date or Timestamp

The MONTH function returns the month field from a date or timestamp value.

Syntax: How to Obtain the Month From a Date or Timestamp

```
MONTH(arg)
```

where:

arg

Date or timestamp

Is the input value.

This function returns an integer value.

Example: Obtaining the Month From a Date or Timestamp

MONTH returns the month from a date or timestamp. This example,

```
MONTH('1976-07-04')
```

returns 7.

This example,

```
MONTH('2001-01-22 10:00:00')
```

returns 1.

SECOND: Obtaining the Second Field From Time/Timestamp

How to:

Obtain the Second Field From a Time or Timestamp

The SECOND function returns the second field from a time or timestamp value.

Syntax: How to Obtain the Second Field From a Time or Timestamp

```
SECOND(arg)
```

where:

arg

Time or timestamp

Is the input value.

This function returns an integer value.

Example: Obtaining the Second Field From a Time or Timestamp

SECOND returns seconds from a time or timestamp. This example,

```
SECOND('11:22:33')
```

returns 33.

This example,

```
SECOND('2001-01-22 12:24:36')
```

returns 36.

YEAR: Obtaining the Year From Date/Timestamp

How to:

Obtain the Year From a Date or Timestamp

The YEAR function returns the year field from a date or timestamp value.

Syntax: How to Obtain the Year From a Date or Timestamp

```
YEAR(arg)
```

where:

arg

Date or timestamp

Is the input value.

This function returns an integer value.

Example: Obtaining the Year From a Date or Timestamp

YEAR returns the year from a date or timestamp value. This example,

```
YEAR('1976-07-04')
```

returns 1976.

This example,

```
YEAR('2001-01-22 10:00:00')
```

returns 2001.

11 SQL Data Type Conversion Functions

SQL data type conversion functions convert fields from one data type to another.

Topics:

CAST: Converting to a Specific Data Type

How to:

Convert to a Specific Data Type

The CAST function converts the value of its argument to a specified data type.

Syntax: How to Convert to a Specific Data Type

```
CAST(arg AS data-type)
```

where:

arg

Any data type that can be converted to the result data type

Is the value to be converted.

data-type

Is the result data type: CHARACTER, CHARACTER VARYING, NUMERIC, DECIMAL, INTEGER, SMALLINT, FLOAT, REAL, DOUBLE PRECISION, DATE, TIME or TIMESTAMP.

This function returns the input value converted to the specified data type.

Example: Converting to a Specific Data Type

CAST converts a value to a specified data type. This example,

```
CAST(2.5 AS INTEGER)
```

returns 2.

This example,

```
CAST('3.333' AS FLOAT)
```

returns 3.333.

CHAR: Converting to a Character String

How to:

Convert to a Character String

The CHAR function converts its argument to a character string.

Syntax: How to Convert to a Character String

```
CHAR(arg)
```

where:

arg

Any type

Is the value to be converted.

This function returns a character string whose length is of sufficient size to hold the value.

Example: Converting to a Character String

CHAR converts a value to a character string. This example,

```
CHAR(566.23)
```

returns 566.23.

DATE: Converting to a Date

How to:

Convert to a Date

The DATE function converts its argument to a date. The type of the argument value may be character, date, or timestamp.

If the argument is:

- A character, its value must correctly represent a date; that date is the result.
- A date, its value is returned.
- A timestamp, the date portion of the timestamp's value is returned.

Syntax: How to Convert to a Date

```
DATE(arg)
```

where:

arg

character string, date, or timestamp

Is the value to be converted.

The DATE function returns a date in YYMD format.

Example: Converting to a Date

DATE converts a value to a date. This example,

```
DATE('1999-03-29 14:39:30')
```

returns 19990329.

DECIMAL: Converting to Decimal Format

How to:

Convert to the Decimal Format

The DECIMAL function converts a number to fixed-length decimal format.

Syntax: How to Convert to the Decimal Format

```
DECIMAL(arg, [length [,dec-places]])
```

where:

arg

Numeric

Is the input value.

length

Integer

The maximum number of digits in the integer portion of the result. The default is 15.

dec-places

Integer

Is the number of decimal places in the result. The default is the same number of decimal places as in the type of the argument.

This function returns a numeric value in fixed-length decimal format.

Example: Converting to Decimal Format

DECIMAL converts a number to fixed-length decimal format. This example,

```
DECIMAL(5.12345, 4, 2)
```

returns 5.12.

FLOAT: Converting to Floating Point Format

How to:

Convert to the Floating Point Format

The FLOAT function converts a number to floating-point format.

Syntax: How to Convert to the Floating Point Format

`FLOAT(arg)`

where:

arg

Numeric

Is the input value.

This function returns the value in floating-point format.

Example: Converting to Floating Point Format

FLOAT converts a number to floating-point format. This example,

`FLOAT(3)`

returns 3.0.

INT: Converting to an Integer

How to:

Convert to an Integer

The INT function converts a number to an integer. If the input value is not an integer, the result is truncated.

INTEGER is identical to INT.

Syntax: How to Convert to an Integer

`INT(arg)`

where:

arg

Numeric

Is the input value.

This function returns the number in integer format.

Example: Converting to an Integer

INT converts a number to an integer. This example,

```
INT(4.8)
```

returns 4.

SMALLINT: Converting to a Small Integer

How to:

Convert to a Small Integer

The SMALLINT function converts a number to a small integer. Generally, a small integer occupies only 2 bytes in memory.

Syntax: How to Convert to a Small Integer

```
SMALLINT(arg)
```

where:

arg

Numeric

Is the input value.

This function returns the number in small integer format.

Example: Converting to a Small Integer

SMALLINT converts a number to a small integer. This example,

```
SMALLINT(3.5)
```

returns 3.

TIME: Converting to a Time

How to:

Convert to a Time

The TIME function converts its argument to a time. The type of the argument value may be character, time, or timestamp.

- If the argument is a character, its value must correctly represent a time; that time is the result.
- If the argument is a time, its value is returned.
- If the argument is a timestamp, the time portion of the timestamp's value is returned.

Syntax: How to Convert to a Time

```
TIME(arg)
```

where:

arg

character string, time, or timestamp

Is the input value.

This function returns a time.

Example: Converting to a Time

TIME converts a value argument to a time. This example,

```
TIME('2004-03-15 01:02:03.444')
```

returns 010203444.

TIMESTAMP: Converting to a Timestamp

How to:

Convert to a Timestamp

The TIMESTAMP function converts its argument to a timestamp. The argument type can be character, date, time, or timestamp.

- If the argument is a character, its value must correctly represent a timestamp; that timestamp is the result.

- If the argument is a date, the value of the result is the timestamp, with the date component equal to the argument and the time component equal to midnight.
- If the argument is a time, the value of the result is the timestamp, with the date component equal to the current date, and the time component equal to the argument.
- If the argument is a timestamp, its value is returned.

Syntax: How to Convert to a Timestamp

```
TIMESTAMP(arg)
```

where:

arg

character string, date, time, or timestamp

Is the input value.

This function returns a timestamp.

Example: Converting to a Timestamp

TIMESTAMP converts a value to a timestamp. This example,

```
TIMESTAMP('2004-06-24')
```

returns 20040624000000.

This example,

```
TIMESTAMP('11:22:33')
```

returns 20010101112233, if the current date is January 1, 2001.

12 SQL Numeric Functions

SQL numeric functions perform calculations on numeric constants and fields.

Topics:

ABS: Returning an Absolute Value (SQL)

How to:

Return an Absolute Value

The ABS function returns the absolute value of a number.

Syntax: How to Return an Absolute Value

```
ABS(arg)
```

where:

arg

Numeric

Is the input value.

This function returns the value as the same datatype as the argument. For example, if the argument is an integer, the result will be also be an integer.

Example: Returning an Absolute Value

ABS returns the absolute value of a number. This example,

```
ABS(-5.5)
```

returns 5.5.

LOG: Returning a Logarithm (SQL)

How to:

Return a Logarithm

The LOG function returns the natural logarithm of the input value.

Syntax: How to Return a Logarithm

```
LOG(arg)
```

where:

arg

Numeric

Is the input value.

This function returns double precision numbers with 3 decimal places.

Example: Returning a Logarithm

LOG returns the natural logarithm of a value. This example,

```
LOG(4)
```

returns 1.386.

SQRT Returning a Square Root (SQL)

How to:

Return a Square Root

The SQRT function returns the square root of the input value.

Syntax: How to Return a Square Root

```
sqrt(arg)
```

where:

arg

Numeric

Is the input value.

This function returns double precision numbers with 3 decimal places.

Example: Returning a Square Root

SQRT returns the square root of a value. This example,

```
SQRT(4)
```

returns 2.000.

13 SQL Miscellaneous Functions

The SQL functions described in this chapter perform a variety of conversions, tests, and manipulations.

Topics:

- COUNTBY: Incrementing Column Values Row by Row
- HEX: Converting to Hexadecimal
- IF: Testing a Condition
- LENGTH: Obtaining the Physical Length of a Data Item
- VALUE: Coalescing Data Values

COUNTBY: Incrementing Column Values Row by Row

How to:

Increment Column Values Row by Row

The COUNTBY function produces a column whose values are incremented row-by-row by a specified amount.

Syntax: How to Increment Column Values Row by Row

```
COUNTBY(arg)
```

where:

arg

Integer

Is the value that is incremented for each record.

This function returns an integer value.

Example: Incrementing Column Values Row by Row

In the query,

```
SELECT COUNTBY(1), COUNTBY(2) FROM T
```

the first column takes on the values 1, 2, 3, ..., and the second column takes on the values 2, 4, 6, ...

HEX: Converting to Hexadecimal

How to:

Convert to Hexadecimal

The HEX function converts its input value to hexadecimal.

Note: This function is available only for DB2, Ingres, and Informix.

Syntax: How to Convert to Hexadecimal

```
HEX(arg)
```

where:

arg

Numeric

Is the input value.

This function returns an alphanumeric value.

Example: Converting a Value to Hex

This example,

```
HEX('110')
```

returns 6E.

IF: Testing a Condition

How to:

Test a Condition

The IF function tests a condition and returns a value based on whether the condition is true or false.

Syntax: How to Test a Condition

```
IF(test, val1, val2)
```

where:

test

Condition

Is an SQL search condition, which evaluates to true or false.

val1, val2

Are expressions of compatible types.

This function returns a value of the type of val1 and val2. If test is true, val1 is returned, otherwise val2 is returned.

Example: Testing a Condition

This example tests COUNTRY. If the value is ENGLAND, it returns LONDON. Otherwise, it returns PARIS.

```
IF(COUNTRY = 'ENGLAND', 'LONDON', 'PARIS') =
   'LONDON'   if COUNTRY is 'ENGLAND'
   'PARIS'    otherwise.
```

This example tests COUNTRY. If the value is ENGLAND, it returns LONDON. If the value is FRANCE, it returns PARIS. Otherwise, it returns ROME.

```
IF(COUNTRY = 'ENGLAND', 'LONDON',
   IF(COUNTRY = 'FRANCE', 'PARIS', 'ROME')) =
   'LONDON'   if COUNTRY is 'ENGLAND'
   'PARIS'    if COUNTRY = 'FRANCE'
   'ROME'     otherwise.
```

LENGTH: Obtaining the Physical Length of a Data Item

How to:

Obtain the Physical Length of a Data Item

The LENGTH function returns the actual length in memory of a data item.

Syntax: How to Obtain the Physical Length of a Data Item

```
LENGTH(arg)
```

where:

arg

Any type

Is the length of the argument. It can be between 1 and 16 bytes.

This function returns an integer value.

Example: Obtaining the Physical Length of a Data Item

LENGTH returns the length in memory of a data item. This example,

```
LENGTH('abcdef')
```

returns 6.

This example,

```
LENGTH(3)
```

returns 4.

VALUE: Coalescing Data Values

How to:

Coalesce Data Values

The VALUE function can take 2 or more arguments. The first argument that is not NULL is returned. If all arguments are NULL, NULL is returned.

Syntax: How to Coalesce Data Values

```
VALUE(arg1, arg2, [... argn])
```

where:

arg1, arg2, ..., argn

Any type

The types must be compatible.

This function returns the compatible type of the arguments.

Example: Coalescing Data Values

This example,

```
VALUE('A', 'B')
```

return A.

This example,

```
VALUE(NULL, 'B')
```

return B.

This example,

```
VALUE(NULL, NULL)
```

return NULL.

14 SQL Operators

SQL operators are used to evaluate expressions.

Topics:

- CASE: SQL Case Operator
- COALESCE: Coalescing Data Values
- NULLIF: NULLIF Operator

CASE: SQL Case Operator

How to:

Use the SQL Case Operator

The CASE operator allows a value to be computed depending on the values of expressions or the truth or falsity of conditions.

Syntax: How to Use the SQL Case Operator

In the first format below the value of *test-expr* is compared to *value-expr-1, ..., value-expr-n* in turn:

- If any of these match, the value of the result is the corresponding *result-expr*.
- If there are no matches and the ELSE clause is present, the result is *else-expr*.
- If there are no matches and the ELSE clause is not present, the result is NULL.

In the second format below the values of *cond-1, ..., cond-n* are evaluated in turn.

- If any of these are true, the value of the result is the corresponding *result-expr*.
- If no conditions are true and the ELSE clause is present, the result is *else-expr*.
- If no conditions are true and the ELSE clause is not present, the result is NULL.

Format 1

```
CASE test-expr
   WHEN value-expr-1 THEN result-expr-1
   . . .
   WHEN value-expr-n THEN result-expr-n
   [ ELSE else-expr ]
END
```

Format 2

```
CASE
   WHEN cond-1 THEN result-expr-1
   . . .
   WHEN cond-n THEN result-expr-n
   [ ELSE else-expr ]
END
```

where:

test-expr

Any type

Is the value to be tested in Format 1.

value-expr1, ... , value-expr-n

Any type of compatible with *test-expr*.

Are the values *test-expr* is tested against in Format 1.

result-expr1, ... , result-expr-n

Any type

Are the values that become the result value if:

- The corresponding *value-expr* matches *test-expr* (Format 1).

or

- The corresponding *cond* is true (Format 2).

The result expressions must all have a compatible type.

cond-1, ..., cond-n

Condition

Are conditions that are tested in Format 2.

else-expr

Any type

Is the value of the result if no matches are found. Its type must be compatible with the result expressions.

This operator returns the compatible type of the result expressions.

Example: Using the SQL Case Operator

CASE returns values based on expressions. This example,

```
CASE COUNTRY
   WHEN 'ENGLAND' THEN 'LONDON'
   WHEN 'FRANCE' THEN 'PARIS'
   WHEN 'ITALY' THEN 'ROME'
   ELSE 'UNKNOWN'
END
```

returns LONDON when the value is ENGLAND, PARIS when the value is FRANCE, ROME when the value is ITALY, and UNKNOWN when there is no match.

COALESCE: Coalescing Data Values

How to:

Coalesce Data Values

The COALESCE operator can take 2 or more arguments. The first argument that is not NULL is returned. If all arguments are NULL, NULL is returned.

Syntax: How to Coalesce Data Values

```
COALESCE(arg1, arg2, [ ... argn ])
```

where:

arg1, arg2, ..., argn

Any type

Are data values. The types of the arguments must be compatible.

This operator returns the compatible type of the arguments.

Example: Coalescing Data Values

This example,

```
COALESCE('A', 'B')
```

return A.

This example,

```
COALESCE(NULL, 'B')
```

return B.

This example,

```
COALESCE(NULL, NULL)
```

return NULL.

NULLIF: NULLIF Operator

How to:

Use the NULLIF Operator

The NULLIF operator returns NULL if its two arguments are equal. Otherwise, the first argument is returned.

Syntax: How to Use the NULLIF Operator

```
NULLIF(arg1, arg2)
```

where:

arg1, arg2

Any type

Are data values. The types of the two arguments must be compatible.

This operator returns the compatible type of the arguments.

Example: Using the NULLIF Operator

NULLIF operator returns NULL if two values are equal. This example,

```
NULLIF(IDNUM, -1)
```

returns NULL if the identification number is -1, otherwise it returns the number.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

X

Y

Reader Comments

In an ongoing effort to produce effective documentation, the Documentation Services staff at Information Builders welcomes any opinion you can offer regarding this manual.

Please use this form to relay suggestions for improving this publication or to alert us to corrections. Identify specific pages where applicable. You can contact us through the following methods:

Mail:	Documentation Services - Customer Support Information Builders, Inc. Two Penn Plaza New York, NY 10121-2898
Fax:	(212) 967-0460
E-mail:	*books_info@ibi.com*
Web form:	*http://www.informationbuilders.com/bookstore/derf.html*

Name:_____

Company:_____

Address:_____

Telephone:_____ Date:_____

Email:_____

Comments:

Reader Comments