Contents

Preface ........................................................................ 15
Conventions ...................................................................... 17
Related Publications ................................................................. 18
Customer Support ................................................................... 18
Information You Should Have .......................................................... 19
User Feedback ...................................................................... 20
iWay Software Training and Professional Services ......................... 20

1. Functions Overview ........................................................... 21
Function Arguments .................................................................. 21
Function Categories .................................................................. 22
Character Chart for ASCII and EBCDIC ............................................. 22

2. Simplified Analytic Functions ................................................. 31
FORECAST_MOVAVE: Using a Simple Moving Average ...................... 31
FORECAST_EXPAVE: Using Single Exponential Smoothing ................. 37
FORECAST_DOUBLEXP: Using Double Exponential Smoothing ........... 40
FORECAST_SEASONAL: Using Triple Exponential Smoothing ............... 42
FORECAST_LINEAR: Using a Linear Regression Equation ...................... 47
PARTITION_AGGR: Creating Rolling Calculations ............................... 50
PARTITION_REF: Using Prior or Subsequent Field Values in Calculations 59
INCREASE: Calculating the Difference Between the Current and a Prior Value of a Field .......................................................... 63
PCT_INCREASE: Calculating the Percentage Difference Between the Current and a Prior Value of a Field .................................................. 67
PREVIOUS: Retrieving a Prior Value of a Field ..................................... 70
RUNNING_AVE: Calculating an Average Over a Group of Rows ........... 72
RUNNING_MAX: Calculating a Maximum Over a Group of Rows .......... 75
RUNNING_MIN: Calculating a Minimum Over a Group of Rows ........... 78
RUNNING_SUM: Calculating a Sum Over a Group of Rows ................... 81

3. Simplified Character Functions ............................................. 85
CHAR_LENGTH: Returning the Length in Characters of a String .......... 86
CONCAT: Concatenating Strings After Removing Trailing Blanks From the First 87
DIGITS: Converting a Number to a Character String ......................... 90

Functions Reference
GET_TOKEN: Extracting a Token Based on a String of Delimiters ........................................ 92
INITCAP: Capitalizing the First Letter of Each Word in a String ........................................ 93
LAST_NONBLANK: Retrieving the Last Field Value That is Neither Blank nor Missing .............. 95
LOWER: Returning a String With All Letters Lowercase ..................................................... 96
LPAD: Left-Padding a Character String ................................................................................. 98
LTRIM: Removing Blanks From the Left End of a String ...................................................... 99
PATTERNS: Returning a Pattern That Represents the Structure of the Input String ............... 101
POSITION: Returning the First Position of a Substring in a Source String ......................... 102
REGEX: Matching a String to a Regular Expression ............................................................. 104
REPLACE: Replacing a String ................................................................................................. 106
RPAD: Right-Padding a Character String ............................................................................. 108
RTRIM: Removing Blanks From the Right End of a String .................................................. 110
SPLIT: Extracting an Element From a String ........................................................................ 111
SUBSTRING: Extracting a Substring From a Source String ................................................. 112
TOKEN: Extracting a Token From a String ............................................................................ 114
TRIM_: Removing a Leading Character, Trailing Character, or Both From a String .......... 116
UPPER: Returning a String With All Letters Uppercase ..................................................... 118

4. Character Functions ........................................................................................................ 121
ARGLEN: Measuring the Length of a String ......................................................................... 122
ASIS: Distinguishing Between Space and Zero ..................................................................... 123
BITSON: Determining If a Bit Is On or Off .......................................................................... 125
BITVAL: Evaluating a Bit String as an Integer ...................................................................... 127
BYTVAL: Translating a Character to Decimal ....................................................................... 129
CHKFMT: Checking the Format of a String .......................................................................... 131
CHKNUM: Checking a String for Numeric Format ................................................................. 135
CTRAN: Translating One Character to Another ..................................................................... 137
CTRFLD: Centering a Character String .................................................................................. 143
EDIT: Extracting or Adding Characters ................................................................................ 144
GETTOK: Extracting a Substring (Token) ............................................................................. 146
LCWORD: Converting a String to Mixed-Case ...................................................................... 148
LCWORD2: Converting a String to Mixed-Case .................................................................... 149
LCWORD3: Converting a String to Mixed-Case .................................................................... 151
LJUST: Left-Justifying a String ........................................................ 152
LOCASE: Converting Text to Lowercase ....................................................... 153
OVRLAY: Overlaying a Character String ..................................................... 155
PARAG: Dividing Text Into Smaller Lines ................................................... 158
PATTERN: Generating a Pattern From a String .......................................... 160
POSIT: Finding the Beginning of a Substring .............................................. 163
REVERSE: Reversing the Characters in a String ......................................... 164
RJUST: Right-Justifying a Character String ............................................... 166
SOUNDEX: Comparing Character Strings Phonetically ................................... 167
SPELLNM: Spelling Out a Dollar Amount ................................................... 169
SQUEEZ: Reducing Multiple Spaces to a Single Space .................................. 171
STRIP: Removing a Character From a String ............................................ 172
STRREP: Replacing Character Strings ....................................................... 174
SUBSTR: Extracting a Substring ............................................................. 176
TRIM: Removing Leading and Trailing Occurrences .................................... 178
UPCASE: Converting Text to Uppercase .................................................... 181
XMLDECOD: Decoding XML-Encoded Characters ....................................... 184
XMLENCOD: XML-Encoding Characters .................................................... 186

5. Variable Length Character Functions ......................................... 189
Overview ............................................................................... 189
LENV: Returning the Length of an Alphanumeric Field ................................ 190
LOCASV: Creating a Variable Length Lowercase String ............................ 191
POSITV: Finding the Beginning of a Variable Length Substring ................. 193
SUBSTV: Extracting a Variable Length Substring ................................... 195
TRIMV: Removing Characters From a String ........................................... 197
UPCASV: Creating a Variable Length Uppercase String .......................... 199

6. Character Functions for DBCS Code Pages .................................... 203
DCTRAN: Translating A Single-Byte or Double-Byte Character to Another .......... 203
DEDIT: Extracting or Adding Characters .................................................. 204
DSTRIPl: Removing a Single-Byte or Double-Byte Character From a String .......... 206
DSUBSTR: Extracting a Substring .......................................................... 207
JPPTRANS: Converting Japanese Specific Characters .................................. 208
7. **Data Source and Decoding Functions** .......................... 219

- KKFCUT: Truncating a String ......................................................... 213
- SFTDEL: Deleting the Shift Code From DBCS Data ...................... 214
- SFTINS: Inserting the Shift Code Into DBCS Data ....................... 216

8. **Simplified Date and Date-Time Functions** ............................... 261

- DT_CURRENT_DATE: Returning the Current Date ......................... 262
- DT_CURRENT_DATETIME: Returning the Current Date and Time .......... 262
- DT_CURRENT_TIME: Returning the Current Time ......................... 263
- DTADD: Incrementing a Date or Date-Time Component .................. 264
- DTDIFF: Returning the Number of Component Boundaries Between Date or Date-Time Values 267
- DTIME: Extracting Time Components From a Date-Time Value .......... 269
- DTPART: Returning a Date or Date-Time Component in Integer Format 271
- DTRUNC: Returning the Start of a Date Period for a Given Date ....... 273

9. **Date Functions** .............................................................. 277

- Overview of Date Functions ....................................................... 278
- Using Standard Date Functions .................................................. 279
  - Specifying Work Days ......................................................... 280
  - Specifying Business Days ................................................... 280
  - Specifying Holidays ......................................................... 280
Enabling Leading Zeros For Date and Time Functions in Dialogue Manager ................. 286
DATEADD: Adding or Subtracting a Date Unit to or From a Date .................................. 287
DATECVT: Converting the Format of a Date ................................................................. 291
DATEDIF: Finding the Difference Between Two Dates .................................................. 293
DATEDMOV: Moving a Date to a Significant Point ....................................................... 297
DATETRAN: Formatting Dates in International Formats ............................................... 304
FIYR: Obtaining the Financial Year .................................................................................. 320
FIQTR: Obtaining the Financial Quarter ......................................................................... 322
FIYYQ: Converting a Calendar Date to a Financial Date .............................................. 325
TODAY: Returning the Current Date .............................................................................. 328
Using Legacy Date Functions ......................................................................................... 329
Using Old Versions of Legacy Date Functions ............................................................... 329
Using Dates With Two- and Four-Digit Years ................................................................. 330
AYMD: Adding or Subtracting Days ................................................................................ 331
CHGDAT: Changing How a Date String Displays ............................................................ 333
DA Functions: Converting a Legacy Date to an Integer .................................................. 336
DMY, MDY, YMD: Calculating the Difference Between Two Dates ................................ 338
DOWK and DOWKL: Finding the Day of the Week ......................................................... 339
DT Functions: Converting an Integer to a Date .............................................................. 341
GREGDT: Converting From Julian to Gregorian Format ................................................. 342
JULDAT: Converting From Gregorian to Julian Format .................................................. 344
YM: Calculating Elapsed Months ..................................................................................... 346

10. Date-Time Functions .................................................................................................. 349
Using Date-Time Functions ............................................................................................. 350
Date-Time Parameters ..................................................................................................... 350
  Specifying the Order of Date Components .................................................................. 350
  Specifying the First Day of the Week for Use in Date-Time Functions ....................... 351
  Controlling Processing of Date-Time Values .............................................................. 353
Supplying Arguments for Date-Time Functions ............................................................... 354
Using Date-Time Formats ............................................................................................... 355
  Numeric String Format ............................................................................................... 356
  Formatted-string Format ............................................................................................ 356
Contents

Translated-string Format.................................................................357
Time Format..................................................................................357
Assigning Date-Time Values..........................................................358
HADD: Incrementing a Date-Time Value........................................361
HCNVRT: Converting a Date-Time Value to Alphanumeric Format ....364
HDATE: Converting the Date Portion of a Date-Time Value to a Date Format..........................366
HDIFF: Finding the Number of Units Between Two Date-Time Values..................367
HDTTM: Converting a Date Value to a Date-Time Value..........................369
HGETC: Storing the Current Local Date and Time in a Date-Time Field ............371
HGETZ: Storing the Current Coordinated Universal Time in a Date-Time Field .........373
HHMMSS: Retrieving the Current Time.............................................375
HHMS: Converting a Date-Time Value to a Time Value..........................376
HINPUT: Converting an Alphanumeric String to a Date-Time Value..................377
HMIDNT: Setting the Time Portion of a Date-Time Value to Midnight..............379
HNAME: Retrieving a Date-Time Component in Alphanumeric Format ...............381
HPART: Retrieving a Date-Time Component as a Numeric Value.................383
HSETPT: Inserting a Component Into a Date-Time Value..........................385
HTIME: Converting the Time Portion of a Date-Time Value to a Number..........387
HTMTOTS or TIMETOTS: Converting a Time to a Timestamp..........................388
HYYWD: Returning the Year and Week Number From a Date-Time Value............390

11. Simplified Conversion Functions................................................395
CHAR: Returning a Character Based on a Numeric Code............................395
COMPACTFORMAT: Displaying Numbers in an Abbreviated Format................396
CTRLCHAR: Returning a Non-Printable Control Character..........................398
FPRINT: Displaying a Value in a Specified Format..................................400
HEXTYPE: Returning the Hexadecimal View of an Input Value.......................402
PHONETIC: Returning a Phonetic Key for a String..................................405
TO_INTEGER: Converting a Character String to an Integer Value...................407
TO_NUMBER: Converting a Character String to a Numeric Value....................408

12. Format Conversion Functions....................................................409
ATODBL: Converting an Alphanumeric String to Double-Precision Format..........410
EDIT: Converting the Format of a Field..............................................412
13. Simplified Numeric Functions ......................................................... 439
   CEILING: Returning the Smallest Integer Value Greater Than or Equal to a Value .......... 439
   EXPONENT: Raising e to a Power .................................................. 441
   FLOOR: Returning the Largest Integer Less Than or Equal to a Value ...................... 442
   MOD: Calculating the Remainder From a Division .................................... 444
   POWER: Raising a Value to a Power ............................................... 445

14. Numeric Functions ................................................................. 447
   ABS: Calculating Absolute Value .................................................. 447
   CHKPCK: Validating a Packed Field .............................................. 448
   DMOD, FMOD, and IMOD: Calculating the Remainder From a Division ...................... 451
   EXP: Raising e to the Nth Power ................................................ 453
   EXPN: Evaluating a Number in Scientific Notation .................................. 454
   INT: Finding the Greatest Integer .............................................. 455
   LOG: Calculating the Natural Logarithm ......................................... 456
   MAX and MIN: Finding the Maximum or Minimum Value .................................. 457
   NORMSDST and NORMSINV: Calculating Normal Distributions ......................... 458
      NORMSDST: Calculating Standard Cumulative Normal Distribution .................. 459
      NORMSINV: Calculating Inverse Cumulative Normal Distribution .................. 462
   PRDNOR and PRDUNI: Generating Reproducible Random Numbers ....................... 463
   RDNORM and RDUNIF: Generating Random Numbers ................................... 466
   SQRT: Calculating the Square Root ............................................. 468
### 15. Simplified Statistical Functions

- Specify the Partition Size for Simplified Statistical Functions
- **CORRELATION**: Calculating the Degree of Correlation Between Two Sets of Data
- **KMEANS_CLUSTER**: Partitioning Observations Into Clusters Based on the Nearest Mean Value
- **MULTIREGRESS**: Creating a Multivariate Linear Regression Column
- **OUTLIER**: Identifying Outliers in Numeric Data
- **RSERVE**: Running an R Script
- **STDDEV**: Calculating the Standard Deviation for a Set of Data Values

### 16. Simplified System Functions

- **EDAPRINT**: Inserting a Custom Message in the EDAPRINT Log File
- **ENCRIPT**: Encrypting a Password
- **GETENV**: Retrieving the Value of an Environment Variable
- **PUTENV**: Assigning a Value to an Environment Variable
- **SLACK**: Posting a Message to a Slack Channel

### 17. System Functions

- **CLSDDREC**: Closing All Files Opened by the PUTDDREC Function
- **FEXERR**: Retrieving an Error Message
- **FGETENV**: Retrieving the Value of an Environment Variable
- **FPUTENV**: Assigning a Value to an Environment Variable
- **GETUSER**: Retrieving a User ID
- **JOBNAME**: Retrieving the Current Process Identification String
- **PUTDDREC**: Writing a Character String as a Record in a Sequential File
- **SLEEP**: Suspending Execution for a Given Number of Seconds
- **SYSVAR**: Retrieving the Value of a z/OS System Variable

### 18. Simplified Geography Functions

- **GIS_DISTANCE**: Calculating the Distance Between Geometry Points
- **GIS_DRIVE_ROUTE**: Calculating the Driving Directions Between Geometry Points
- **GIS_GEOCODE_ADDR**: Geocoding a Complete Address
- **GIS_GEOCODE_ADDR_CITY**: Geocoding an Address Line, City, and State
- **GIS_GEOCODE_ADDR_POSTAL**: Geocoding an Address Line and Postal Code
19. SQL Character Functions ................................................................. 545
   CHAR_LENGTH: Finding the Length of a Character String ......................... 545
   CONCAT: Concatenating Two Character Strings .................................... 546
   DIGITS: Converting a Numeric Value to a Character String ....................... 547
   EDIT: Editing a Value According to a Format (SQL) ................................ 548
   LCASE: Converting a Character String to Lowercase ................................ 549
   LTRIM: Removing Leading Spaces ..................................................... 549
   POSITION: Finding the Position of a Substring .................................... 550
   RTRIM: Removing Trailing Spaces ..................................................... 551
   SUBSTR: Extracting a Substring From a String Value (SQL) ....................... 551
   TRIM: Removing Leading or Trailing Characters (SQL) ............................. 553
   UCASE: Converting a Character String to Uppercase ................................ 554
   VARGRAPHIC: Converting to Double-byte Character Data .......................... 554

20. SQL Date and Time Functions ......................................................... 557
   CURRENT_DATE: Obtaining the Date .................................................... 557
   CURRENT_TIME: Obtaining the Time ................................................... 558
   CURRENT_TIMESTAMP: Obtaining the Timestamp (Date/Time) ....................... 558
   DAY: Obtaining the Day of the Month From a Date/Timestamp ..................... 559
   DAYS: Obtaining the Number of Days Since January 1, 0001 ......................... 560
   EXTRACT: Obtaining a Datetime Field From Date/Time/Timestamp .............. 560
   HOUR: Obtaining the Hour From Time/Timestamp ................................... 561
   MICROSECOND: Obtaining Microseconds From Time/Timestamp .................... 562
   MILLISECOND: Obtaining Milliseconds From Time/Timestamp ....................... 563
   MINUTE: Obtaining the Minute From Time/Timestamp ................................ 563
   MONTH: Obtaining the Month From Date/Timestamp ................................... 564
SECONa: Obtaining the Second Field From Time/Timestamp .................................. 564
QUARTER: Returning the Quarter of the Year ...................................................... 565
WEEKDAY: Returning the Day of the Week .......................................................... 566
YEAR: Obtaining the Year From a Date or Timestamp ......................................... 566

21. SQL Data Type Conversion Functions ......................................................... 569
CAST: Converting to a Specific Data Type ......................................................... 569
CHAR: Converting to a Character String ............................................................ 570
CHAR: Converting to a Standard Date-Time Format ............................................ 571
DATE: Converting to a Date ................................................................................. 572
DECIMAL: Converting to Decimal Format ........................................................... 572
FLOAT: Converting to Floating Point Format ..................................................... 573
INT: Converting to an Integer .............................................................................. 574
SMALLINT: Converting to a Small Integer ........................................................... 574
TIME: Converting to a Time .................................................................................. 575
TIMESTAMP: Converting to a Timestamp ............................................................ 576

22. SQL Numeric Functions .................................................................................... 577
ABS: Returning an Absolute Value (SQL) ............................................................. 577
CEIL: Returning the Smallest Integer Greater Than or Equal to a Value ............... 578
FLOOR: Returning the Largest Integer Less Than or Equal to a Value (SQL) ....... 578
GREATEST: Returning the Largest Value ............................................................ 579
LEAST: Returning the Smallest Value .................................................................... 580
LOG: Returning a Logarithm (SQL) ..................................................................... 581
EXP: Returning e Raised to a Power ..................................................................... 581
MOD: Returning the Remainder of a Division ..................................................... 582
POWER: Raising a Value to a Power (SQL) .......................................................... 582
SQRT: Returning a Square Root (SQL) ................................................................. 583

23. SQL Miscellaneous Functions ........................................................................... 585
COUNTBY: Incrementing Column Values Row by Row ....................................... 585
DB_EXPR: Inserting an SQL Expression Into a Request (SQL) .............................. 586
HEX: Converting to Hexadecimal ......................................................................... 588
IF: Testing a Condition ......................................................................................... 588
LENGTH: Obtaining the Physical Length of a Data Item ...................................... 589
24. SQL Operators ............................................................ 591

   CASE: SQL Case Operator ................................................. 591
   COALESCE: Coalescing Data Values .................................... 593
   NULLIF: NULLIF Operator .................................................. 594
# Preface

This content describes how to use Information Builders-supplied functions to perform complex calculations and manipulate data in procedures. It is intended for application developers and end users.

## How This Manual Is Organized

This manual includes the following chapters:

<table>
<thead>
<tr>
<th>Chapter/Appendix</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Functions Overview</td>
<td>Introduces functions and explains the different types of available functions.</td>
</tr>
<tr>
<td>2 Simplified Analytic Functions</td>
<td>Describes analytic functions that have streamlined parameter lists, similar to those used by SQL functions.</td>
</tr>
<tr>
<td>3 Simplified Character Functions</td>
<td>Describes character functions that have streamlined parameter lists, similar to those used by SQL functions.</td>
</tr>
<tr>
<td>4 Character Functions</td>
<td>Describes character functions that manipulate alphanumeric fields and character strings.</td>
</tr>
<tr>
<td>5 Variable Length Character Functions</td>
<td>Describes variable-length character functions which manipulate alphanumeric fields and character strings.</td>
</tr>
<tr>
<td>6 Character Functions for DBCS Code Pages</td>
<td>Describes functions that manipulate strings of DBCS and SBCS characters when the configuration uses a DBCS code page.</td>
</tr>
<tr>
<td>7 Data Source and Decoding Functions</td>
<td>Describes data source and decoding functions that search for data source records, retrieve data source records or values, and assign values based on the value of an input field.</td>
</tr>
<tr>
<td>8 Simplified Date and Date-Time Functions</td>
<td>Describes date and date-time functions that have streamlined parameter lists, similar to those used by SQL functions.</td>
</tr>
<tr>
<td>9 Date Functions</td>
<td>Describes date functions that manipulate date values.</td>
</tr>
<tr>
<td>Chapter/Appendix</td>
<td>Contents</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------</td>
</tr>
<tr>
<td>10 Date-Time Functions</td>
<td>Describes date-time functions that manipulate date-time values.</td>
</tr>
<tr>
<td>11 Simplified Conversion Functions</td>
<td>Describes conversion functions that have streamlined parameter lists, similar to those used by SQL functions.</td>
</tr>
<tr>
<td>12 Format Conversion Functions</td>
<td>Describes format conversion functions that convert fields from one format to another.</td>
</tr>
<tr>
<td>13 Simplified Numeric Functions</td>
<td>Describes numeric functions that have streamlined parameter lists, similar to those used by SQL functions.</td>
</tr>
<tr>
<td>14 Numeric Functions</td>
<td>Describes numeric functions that perform calculations on numeric constants and fields.</td>
</tr>
<tr>
<td>15 Simplified Statistical Functions</td>
<td>Describes functions that perform statistical calculations.</td>
</tr>
<tr>
<td>16 Simplified System Functions</td>
<td>Describes system functions that have streamlined parameter lists, similar to those used by SQL functions.</td>
</tr>
<tr>
<td>17 System Functions</td>
<td>Describes system functions that call the operating system to obtain information about the operating environment or to use a system service.</td>
</tr>
<tr>
<td>18 Simplified Geography Functions</td>
<td>Describes geography functions that have streamlined parameter lists, similar to those used by SQL functions.</td>
</tr>
<tr>
<td>19 SQL Character Functions</td>
<td>Describes SQL character functions which manipulate alphanumeric fields and character strings.</td>
</tr>
<tr>
<td>20 SQL Date and Time Functions</td>
<td>Describes SQL date and time functions which manipulate date and time values.</td>
</tr>
<tr>
<td>21 SQL Data Type Conversion Functions</td>
<td>Describes SQL format conversion functions which convert fields from one format to another.</td>
</tr>
<tr>
<td>Chapter/Appendix</td>
<td>Contents</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------</td>
</tr>
<tr>
<td>22</td>
<td>SQL Numeric Functions</td>
</tr>
<tr>
<td>23</td>
<td>SQL Miscellaneous Functions</td>
</tr>
<tr>
<td>24</td>
<td>SQL Operators</td>
</tr>
</tbody>
</table>

**Conventions**

The following table describes the conventions that are used in this manual.

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>THIS TYPEFACE</strong> or <strong>this typeface</strong></td>
<td>Denotes syntax that you must enter exactly as shown.</td>
</tr>
<tr>
<td><strong>this typeface</strong></td>
<td>Represents a placeholder (or variable) in syntax for a value that you or the system must supply.</td>
</tr>
<tr>
<td><strong>underscore</strong></td>
<td>Indicates a default setting.</td>
</tr>
<tr>
<td><strong>this typeface</strong></td>
<td>Represents a placeholder (or variable), a cross-reference, or an important term. It may also indicate a button, menu item, or dialog box option that you can click or select.</td>
</tr>
<tr>
<td><strong>Key + Key</strong></td>
<td>Indicates keys that you must press simultaneously.</td>
</tr>
<tr>
<td><strong>{ }</strong></td>
<td>Indicates two or three choices. Type one of them, not the braces.</td>
</tr>
<tr>
<td><strong>[ ]</strong></td>
<td>Indicates a group of optional parameters. None are required, but you may select one of them. Type only the parameter in the brackets, not the brackets.</td>
</tr>
<tr>
<td>**</td>
<td>**</td>
</tr>
</tbody>
</table>
Related Publications

Visit our Technical Content 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. Access Focal Point at 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 website, 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 Services (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 questions. Information Builders consultants can also give you general guidance regarding product capabilities. 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 most effectively, be ready 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?

☐ Provide 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 Technical Content Management staff welcomes your opinions regarding this document. You can contact us through our website http://documentation.informationbuilders.com/connections.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 website (http://education.informationbuilders.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 website (http://www.informationbuilders.com/support).
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.

In this chapter:

- Function Arguments
- Function Categories
- Character Chart for ASCII and EBCDIC

Function Arguments

All function arguments except the last one are input arguments. The formats for these arguments 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 argument is the last function argument. With few exceptions, it is a required argument whose only goal is to provide a format for the output of a function. 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.
**Note:** With CDN ON, numeric function arguments must be delimited by a comma followed by a space.

### Function Categories

Functions are grouped into the following areas:

- **Character Functions**
- **Variable Length Character Functions**
- **Character Functions for DBCS Code Pages**
- **Data Source and Decoding Functions**
- **Date Functions**
  - **Using Standard Date Functions**
  - **Using Legacy Date Functions**
- **Date-Time 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.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>ASCII</th>
<th>EBCDIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>!</td>
<td>exclamation point</td>
</tr>
<tr>
<td>34</td>
<td>&quot;</td>
<td>quotation mark</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
<td>number sign</td>
</tr>
<tr>
<td>36</td>
<td>$</td>
<td>dollar sign</td>
</tr>
<tr>
<td>37</td>
<td>%</td>
<td>percent</td>
</tr>
<tr>
<td>Decimal</td>
<td>ASCII</td>
<td>EBCDIC</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>38</td>
<td>&amp;</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>'</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>(</td>
<td>left parenthesis</td>
</tr>
<tr>
<td>41</td>
<td>)</td>
<td>right parenthesis</td>
</tr>
<tr>
<td>42</td>
<td>*</td>
<td>asterisk</td>
</tr>
<tr>
<td>43</td>
<td>+</td>
<td>plus sign</td>
</tr>
<tr>
<td>44</td>
<td>,</td>
<td>comma</td>
</tr>
<tr>
<td>45</td>
<td>-</td>
<td>hyphen</td>
</tr>
<tr>
<td>46</td>
<td>.</td>
<td>period</td>
</tr>
<tr>
<td>47</td>
<td>/</td>
<td>slash</td>
</tr>
<tr>
<td>48</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>49</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>51</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>52</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>53</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>54</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>55</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>56</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>57</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>58</td>
<td>:</td>
<td>colon</td>
</tr>
<tr>
<td>59</td>
<td>;</td>
<td>semicolon</td>
</tr>
</tbody>
</table>
### Character Chart for ASCII and EBCDIC

<table>
<thead>
<tr>
<th>Decimal</th>
<th>ASCII</th>
<th>EBCDIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td><code>&lt;</code></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td><code>=</code></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td><code>&gt;</code></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td><code>?</code></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td><code>@</code></td>
<td></td>
</tr>
<tr>
<td>65</td>
<td><code>A</code></td>
<td></td>
</tr>
<tr>
<td>66</td>
<td><code>B</code></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td><code>C</code></td>
<td></td>
</tr>
<tr>
<td>68</td>
<td><code>D</code></td>
<td></td>
</tr>
<tr>
<td>69</td>
<td><code>E</code></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td><code>F</code></td>
<td></td>
</tr>
<tr>
<td>71</td>
<td><code>G</code></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td><code>H</code></td>
<td></td>
</tr>
<tr>
<td>73</td>
<td><code>I</code></td>
<td></td>
</tr>
<tr>
<td>74</td>
<td><code>J</code></td>
<td><code>§</code></td>
</tr>
<tr>
<td>75</td>
<td><code>K</code></td>
<td><code>. </code></td>
</tr>
<tr>
<td>76</td>
<td><code>L</code></td>
<td><code>&lt;</code></td>
</tr>
<tr>
<td>77</td>
<td><code>M</code></td>
<td><code>( </code></td>
</tr>
<tr>
<td>78</td>
<td><code>N</code></td>
<td><code>+</code></td>
</tr>
<tr>
<td>79</td>
<td><code>O</code></td>
<td>`</td>
</tr>
<tr>
<td>80</td>
<td><code>P</code></td>
<td><code>&amp;</code></td>
</tr>
<tr>
<td>81</td>
<td><code>Q</code></td>
<td></td>
</tr>
</tbody>
</table>

24 Information Builders
<table>
<thead>
<tr>
<th>Decimal</th>
<th>ASCII</th>
<th>EBCDIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>82</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>83</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>84</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>85</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>86</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>87</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>88</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>89</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>90</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>91</td>
<td>[</td>
<td>opening bracket</td>
</tr>
<tr>
<td>92</td>
<td>\</td>
<td>back slant</td>
</tr>
<tr>
<td>93</td>
<td>]</td>
<td>closing bracket</td>
</tr>
<tr>
<td>94</td>
<td>^</td>
<td>caret</td>
</tr>
<tr>
<td>95</td>
<td>_</td>
<td>underscore</td>
</tr>
<tr>
<td>96</td>
<td>`</td>
<td>grave accent</td>
</tr>
<tr>
<td>97</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>98</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>99</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>100</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>101</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>102</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>103</td>
<td>g</td>
<td>g</td>
</tr>
<tr>
<td>Decimal</td>
<td>ASCII</td>
<td>EBCDIC</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>104</td>
<td>h</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>k</td>
<td>,</td>
</tr>
<tr>
<td>108</td>
<td>l</td>
<td>%</td>
</tr>
<tr>
<td>109</td>
<td>m</td>
<td>_</td>
</tr>
<tr>
<td>110</td>
<td>n</td>
<td>&gt;</td>
</tr>
<tr>
<td>111</td>
<td>o</td>
<td>?</td>
</tr>
<tr>
<td>112</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>q</td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>117</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>118</td>
<td>v</td>
<td></td>
</tr>
<tr>
<td>119</td>
<td>w</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>z</td>
<td>:</td>
</tr>
<tr>
<td>123</td>
<td>{</td>
<td>#</td>
</tr>
<tr>
<td>124</td>
<td></td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>}</td>
<td>'</td>
</tr>
<tr>
<td>Decimal</td>
<td>ASCII</td>
<td>EBCDIC</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>----------------</td>
</tr>
<tr>
<td>126</td>
<td>~</td>
<td>= equal sign</td>
</tr>
<tr>
<td>127</td>
<td>&quot;</td>
<td>quotation mark</td>
</tr>
<tr>
<td>129</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>130</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>131</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>132</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>133</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>134</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>135</td>
<td>g</td>
<td>g</td>
</tr>
<tr>
<td>136</td>
<td>h</td>
<td>h</td>
</tr>
<tr>
<td>137</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>145</td>
<td>j</td>
<td>j</td>
</tr>
<tr>
<td>146</td>
<td>k</td>
<td>k</td>
</tr>
<tr>
<td>147</td>
<td>l</td>
<td>l</td>
</tr>
<tr>
<td>148</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td>149</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>150</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>151</td>
<td>p</td>
<td>p</td>
</tr>
<tr>
<td>152</td>
<td>q</td>
<td>q</td>
</tr>
<tr>
<td>153</td>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td>162</td>
<td>s</td>
<td>s</td>
</tr>
<tr>
<td>163</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>Decimal</td>
<td>ASCII</td>
<td>EBCDIC</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>164</td>
<td></td>
<td>u</td>
</tr>
<tr>
<td>165</td>
<td></td>
<td>v</td>
</tr>
<tr>
<td>166</td>
<td></td>
<td>w</td>
</tr>
<tr>
<td>167</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>168</td>
<td></td>
<td>y</td>
</tr>
<tr>
<td>169</td>
<td></td>
<td>z</td>
</tr>
<tr>
<td>185</td>
<td></td>
<td>\grave accent</td>
</tr>
<tr>
<td>193</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>194</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>195</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>196</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>197</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>198</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>199</td>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>200</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>201</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>209</td>
<td>J</td>
<td>J</td>
</tr>
<tr>
<td>210</td>
<td>K</td>
<td>K</td>
</tr>
<tr>
<td>211</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>212</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>213</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>214</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Decimal</td>
<td>ASCII</td>
<td>EBCDIC</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>215</td>
<td></td>
<td>P</td>
</tr>
<tr>
<td>216</td>
<td></td>
<td>Q</td>
</tr>
<tr>
<td>217</td>
<td></td>
<td>R</td>
</tr>
<tr>
<td>226</td>
<td></td>
<td>S</td>
</tr>
<tr>
<td>227</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>228</td>
<td></td>
<td>U</td>
</tr>
<tr>
<td>229</td>
<td></td>
<td>V</td>
</tr>
<tr>
<td>230</td>
<td></td>
<td>W</td>
</tr>
<tr>
<td>231</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>232</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>233</td>
<td></td>
<td>Z</td>
</tr>
<tr>
<td>240</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>241</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>242</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>243</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>244</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>245</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>246</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>247</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>248</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>249</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>Character Chart for ASCII and EBCDIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------------------</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Information Builders                 |

30

Information Builders
Simplified Analytic Functions

The analytic functions enable you to perform calculations and retrievals using multiple rows in the internal matrix.

In this chapter:

- **FORECAST_MOVAVE**: Using a Simple Moving Average
- **FORECAST_EXPAVE**: Using Single Exponential Smoothing
- **FORECAST_DOUBLEXP**: Using Double Exponential Smoothing
- **FORECAST_SEASONAL**: Using Triple Exponential Smoothing
- **FORECAST_LINEAR**: Using a Linear Regression Equation
- **PARTITION_AGGR**: Creating Rolling Calculations
- **PARTITION_REF**: Using Prior or Subsequent Field Values in Calculations
- **INCREASE**: Calculating the Difference Between the Current and a Prior Value of a Field
- **PCT_INCREASE**: Calculating the Percentage Difference Between the Current and a Prior Value of a Field
- **PREVIOUS**: Retrieving a Prior Value of a Field
- **RUNNING_AVE**: Calculating an Average Over a Group of Rows
- **RUNNING_MAX**: Calculating a Maximum Over a Group of Rows
- **RUNNING_MIN**: Calculating a Minimum Over a Group of Rows
- **RUNNING_SUM**: Calculating a Sum Over a Group of Rows

**FORECAST_MOVAVE: Using a Simple Moving Average**

A simple moving average is a series of arithmetic means calculated with a specified number of values from a field. Each new mean in the series is calculated by dropping the first value used in the prior calculation, and adding the next data value to the calculation.

Simple moving averages are sometimes used to analyze trends in stock prices over time. In this scenario, the average is calculated using a specified number of periods of stock prices. A disadvantage to this indicator is that because it drops the oldest values from the calculation as it moves on, it loses its memory over time. Also, mean values are distorted by extreme highs and lows, since this method gives equal weight to each point.
Predicted values beyond the range of the data values are calculated using a moving average that treats the calculated trend values as new data points.

The first complete moving average occurs at the \( n \)th data point because the calculation requires \( n \) values. This is called the lag. The moving average values for the lag rows are calculated as follows: the first value in the moving average column is equal to the first data value, the second value in the moving average column is the average of the first two data values, and so on until the \( n \)th row, at which point there are enough values to calculate the moving average with the number of values specified.

**Syntax:** How to Calculate a Simple Moving Average Column

\[
\text{FORECAST_MOVAVE}(\text{display}, \text{infield, interval, npredict, npointl})
\]

where:

**display**

Keyword

Specifies which values to display for rows of output that represent existing data. Valid values are:

- **INPUT_FIELD.** This displays the original field values for rows that represent existing data.
- **MODEL_DATA.** This displays the calculated values for rows that represent existing data.

**Note:** You can show both types of output for any field by creating two independent COMPUTE commands in the same request, each with a different display option.

**infield**

Is any numeric field. It can be the same field as the result field, or a different field. It cannot be a date-time field or a numeric field with date display options.

**interval**

Is the increment to add to each sort field value (after the last data point) to create the next value. This must be a positive integer. To sort in descending order, use the BY HIGHEST phrase. The result of adding this number to the sort field values is converted to the same format as the sort field.

For date fields, the minimal component in the format determines how the number is interpreted. For example, if the format is YMD, MDY, or DMY, an interval value of 2 is interpreted as meaning two days. If the format is YM, the 2 is interpreted as meaning two months.
**nnpredict**

Is the number of predictions for FORECAST to calculate. It must be an integer greater than or equal to zero. Zero indicates that you do not want predictions, and is only supported with a non-recursive FORECAST. For the SEASONAL method, nnpredict is the number of periods to calculate. The number of points generated is:

\[ \text{nperiod} \times \text{nnpredict} \]

**npointl**

Is the number of values to average for the MOVAVE method.

**Example:**  **Calculating a New Simple Moving Average Column**

This request defines an integer value named PERIOD to use as the independent variable for the moving average. It predicts three periods of values beyond the range of the retrieved data. The MOVAVE column on the report output shows the calculated moving average numbers for existing data points.

```plaintext
DEFINE FILE GGSALES
SDATE/YYM = DATE;
SYEAR/Y = SDATE;
SMONTH/M = SDATE;
PERIOD/I2 = SMONTH;
END
TABLE FILE GGSALES
SUM UNITS DOLLARS
COMPUTE  MOVAVE/D10.1= FORECAST_MOVAVE(MODEL_DATA, DOLLARS,1,3,3);
BY CATEGORY BY PERIOD
WHERE SYEAR EQ 97 AND CATEGORY NE 'Gifts'
ON TABLE SET STYLE *
GRID=OFF, $
ENDSTYLE
END
```
In the report, the number of values to use in the average is 3 and there are no UNITS or DOLLARS values for the generated PERIOD values.

Each average (MOVAVE value) is computed using DOLLARS values where they exist. The calculation of the moving average begins in the following way:

- The first MOVAVE value (801,123.0) is equal to the first DOLLARS value.
The second MOVAVE value (741,731.5) is the mean of DOLLARS values one and two:
\[(801,123 + 682,340) / 2.\]

The third MOVAVE value (749,513.7) is the mean of DOLLARS values one through three:
\[(801,123 + 682,340 + 765,078) / 3.\]

The fourth MOVAVE value (712,897.3) is the mean of DOLLARS values two through four:
\[(682,340 + 765,078 + 691,274) / 3.\]

For predicted values beyond the supplied values, the calculated MOVAVE values are used as new data points to continue the moving average. The predicted MOVAVE values (starting with 694,975.6 for PERIOD 13) are calculated using the previous MOVAVE values as new data points. For example, the first predicted value (694,975.6) is the average of the data points from periods 11 and 12 (620,264 and 762,328) and the moving average for period 12 (702,334.7). The calculation is: 694,975 = (620,264 + 762,328 + 702,334.7)/3.

**Example: Displaying Original Field Values in a Simple Moving Average Column**

This request defines an integer value named PERIOD to use as the independent variable for the moving average. It predicts three periods of values beyond the range of the retrieved data. It uses the keyword INPUT_FIELD as the first argument in the FORECAST parameter list. The trend values do not display in the report. The actual data values for DOLLARS are followed by the predicted values in the report column.

```
DEFINE FILE GGSALES
SDATE/YYM = DATE;
SYEAR/Y = SDATE;
SMONTH/M = SDATE;
PERIOD/I2 = SMONTH;
END

TABLE FILE GGSALES
SUM UNITS DOLLARS
COMPUTE MOVAVE/D10.1 = FORECAST_MOVAVE(INPUT_FIELD,DOLLARS,1,3,3);
BY CATEGORY BY PERIOD
WHERE SYEAR EQ 97 AND CATEGORY NE 'Gifts'
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```
## FORECAST_MOVAVE: Using a Simple Moving Average

The output is shown in the following image:

<table>
<thead>
<tr>
<th>Category</th>
<th>PERIOD</th>
<th>Unit Sales</th>
<th>Dollar Sales</th>
<th>MOVAVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coffee</td>
<td>1</td>
<td>61666</td>
<td>801,123</td>
<td>801,123.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>54870</td>
<td>682,340</td>
<td>682,340.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>61608</td>
<td>765,078</td>
<td>765,078.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>57050</td>
<td>691,274</td>
<td>691,274.0</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>59229</td>
<td>720,444</td>
<td>720,444.0</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>58466</td>
<td>742,457</td>
<td>742,457.0</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>60771</td>
<td>747,253</td>
<td>747,253.0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>54633</td>
<td>655,896</td>
<td>655,896.0</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>57829</td>
<td>730,317</td>
<td>730,317.0</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>57012</td>
<td>724,412</td>
<td>724,412.0</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>51110</td>
<td>620,264</td>
<td>620,264.0</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>58981</td>
<td>762,328</td>
<td>762,328.0</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>694,975.6</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>719,879.4</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>705,729.9</td>
</tr>
<tr>
<td>Food</td>
<td>1</td>
<td>54394</td>
<td>672,727</td>
<td>672,727.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>54894</td>
<td>699,073</td>
<td>699,073.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>52713</td>
<td>642,802</td>
<td>642,802.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>58026</td>
<td>718,514</td>
<td>718,514.0</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>53289</td>
<td>660,740</td>
<td>660,740.0</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>58742</td>
<td>734,705</td>
<td>734,705.0</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>60127</td>
<td>760,586</td>
<td>760,586.0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>55622</td>
<td>695,235</td>
<td>695,235.0</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>55787</td>
<td>683,140</td>
<td>683,140.0</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>57340</td>
<td>713,768</td>
<td>713,768.0</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>57459</td>
<td>710,138</td>
<td>710,138.0</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>57290</td>
<td>705,315</td>
<td>705,315.0</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>708,397.8</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>707,817.7</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>708,651.9</td>
</tr>
</tbody>
</table>
FORECAST_EXPAVE: Using Single Exponential Smoothing

The single exponential smoothing method calculates an average that allows you to choose weights to apply to newer and older values.

The following formula determines the weight given to the newest value.

\[ k = \frac{2}{1+n} \]

where:

- \( k \) is the newest value.
- \( n \) is an integer greater than one. Increasing \( n \) increases the weight assigned to the earlier observations (or data instances), as compared to the later ones.

The next calculation of the exponential moving average (EMA) value is derived by the following formula:

\[ EMA = (EMA \cdot (1-k)) + (datavalue \cdot k) \]

This means that the newest value from the data source is multiplied by the factor \( k \) and the current moving average is multiplied by the factor \((1-k)\). These quantities are then summed to generate the new EMA.

**Note:** When the data values are exhausted, the last data value in the sort group is used as the next data value.

**Syntax:**

**How to Calculate a Single Exponential Smoothing Column**

\[
\text{FORECAST_EXPAVE}(\text{display}, \text{infield}, \text{interval}, \\
\text{nnpredict, npoint1})
\]

where:

- **display**
  - Keyword
  - Specifies which values to display for rows of output that represent existing data. Valid values are:
    - **INPUT_FIELD**. This displays the original field values for rows that represent existing data.
    - **MODEL_DATA**. This displays the calculated values for rows that represent existing data.
Note: You can show both types of output for any field by creating two independent COMPUTE commands in the same request, each with a different display option.

infield
Is any numeric field. It can be the same field as the result field, or a different field. It cannot be a date-time field or a numeric field with date display options.

interval
Is the increment to add to each sort field value (after the last data point) to create the next value. This must be a positive integer. To sort in descending order, use the BY HIGHEST phrase. The result of adding this number to the sort field values is converted to the same format as the sort field.

For date fields, the minimal component in the format determines how the number is interpreted. For example, if the format is YMD, MDY, or DMY, an interval value of 2 is interpreted as meaning two days. If the format is YM, the 2 is interpreted as meaning two months.

npredict
Is the number of predictions for FORECAST to calculate. It must be an integer greater than or equal to zero. Zero indicates that you do not want predictions, and is only supported with a non-recursive FORECAST.

npoint1
For EXPAVE, this number is used to calculate the weights for each component in the average. This value must be a positive whole number. The weight, k, is calculated by the following formula:

\[ k = \frac{2}{1 + \text{npoint1}} \]

Example: Calculating a Single Exponential Smoothing Column
The following defines an integer value named PERIOD to use as the independent variable for the moving average. It predicts three periods of values beyond the range of retrieved data.

```
DEFINE FILE GGSALES
SDATE/YYM = DATE;
SYEAR/Y = SDATE;
SMONTH/M = SDATE;
PERIOD/I2 = SMONTH;
END
TABLE FILE GGSALES
SUM UNITS DOLLARS
COMPUTE EXPAVE/D10.1 = FORECAST_EXPAVE(MODEL_DATA,DOLLARS,1,3,3);
BY CATEGORY BY PERIOD
WHERE SYEAR EQ 97 AND CATEGORY NE 'Gifts'
ON TABLE SET STYLE *
GRID=OFF;$
ENDSTYLE
```
The output is shown in the following image:

<table>
<thead>
<tr>
<th>Category</th>
<th>PERIOD</th>
<th>Unit Sales</th>
<th>Dollar Sales</th>
<th>EXPAVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coffee</td>
<td>1</td>
<td>61666</td>
<td>801123</td>
<td>801,123.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>54870</td>
<td>682340</td>
<td>741,731.5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>61608</td>
<td>765078</td>
<td>753,404.8</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>57050</td>
<td>691274</td>
<td>722,339.4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>59229</td>
<td>720444</td>
<td>721,391.7</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>58466</td>
<td>742457</td>
<td>731,924.3</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>60771</td>
<td>747253</td>
<td>739,588.7</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>54633</td>
<td>655896</td>
<td>697,742.3</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>57829</td>
<td>730317</td>
<td>714,029.7</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>57012</td>
<td>724412</td>
<td>719,220.8</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>51110</td>
<td>620264</td>
<td>669,742.4</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>58981</td>
<td>762328</td>
<td>716,035.2</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>739,181.6</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>750,754.8</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>756,541.4</td>
</tr>
<tr>
<td>Food</td>
<td>1</td>
<td>54394</td>
<td>672727</td>
<td>672,727.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>54894</td>
<td>699073</td>
<td>685,900.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>52713</td>
<td>642802</td>
<td>664,351.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>58026</td>
<td>718514</td>
<td>691,432.5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>53289</td>
<td>660740</td>
<td>676,086.3</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>58742</td>
<td>734705</td>
<td>705,395.6</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>60127</td>
<td>760586</td>
<td>732,990.8</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>55622</td>
<td>695235</td>
<td>714,112.9</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>55787</td>
<td>683140</td>
<td>698,626.5</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>57340</td>
<td>713768</td>
<td>706,197.2</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>57459</td>
<td>710138</td>
<td>708,167.6</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>57290</td>
<td>705315</td>
<td>706,741.3</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>706,028.2</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>705,671.6</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>705,493.3</td>
</tr>
</tbody>
</table>

In the report, three predicted values of EXPAVE are calculated within each value of CATEGORY. For values outside the range of the data, new PERIOD values are generated by adding the interval value (1) to the prior PERIOD value.

Each average (EXPAVE value) is computed using DOLLARS values where they exist. The calculation of the moving average begins in the following way:

- The first EXPAVE value (801,123.0) is the same as the first DOLLARS value.
- The second EXPAVE value (741,731.5) is calculated as follows. Note that because of rounding and the number of decimal places used, the value derived in this sample calculation varies slightly from the one displayed in the report output:
  
  \[ n=3 \text{ (number used to calculate weights)} \]
  
  \[ k = \frac{2}{1+n} = \frac{2}{4} = 0.5 \]
  
  \[ \text{EXPAVE} = (\text{EXPAVE} \times (1-k)) + (\text{new-DOLLARS} \times k) = (801123 \times 0.5) + (682340 \times 0.50) = 400561.5 + 341170 = 741731.5 \]
The third EXPAVE value (753,404.8) is calculated as follows:

\[
\text{EXPAVE} = (\text{EXPAVE} \times (1-k)) + (\text{new-DOLLARS} \times k) = (741731.5 \times 0.5) + (765078 \times 0.50) = 370865.75 + 382539 = 753404.75
\]

**FORECAST_DOUBLEXP: Using Double Exponential Smoothing**

Double exponential smoothing produces an exponential moving average that takes into account the tendency of data to either increase or decrease over time without repeating. This is accomplished by using two equations with two constants.

- The first equation accounts for the current time period and is a weighted average of the current data value and the prior average, with an added component \( b \) that represents the trend for the previous period. The weight constant is \( k \):

  \[
  \text{DOUBLEXP}(t) = k \times \text{datavalue}(t) + (1-k) \times (\text{DOUBLEXP}(t-1) + b(t-1))
  \]

- The second equation is the calculated trend value, and is a weighted average of the difference between the current and previous average and the trend for the previous time period. \( b(t) \) represents the average trend. The weight constant is \( g \):

  \[
  b(t) = g \times (\text{DOUBLEXP}(t) - \text{DOUBLEXP}(t-1)) + (1 - g) \times b(t-1)
  \]

These two equations are solved to derive the smoothed average. The first smoothed average is set to the first data value. The first trend component is set to zero. For choosing the two constants, the best results are usually obtained by minimizing the mean-squared error (MSE) between the data values and the calculated averages. You may need to use nonlinear optimization techniques to find the optimal constants.

The equation used for forecasting beyond the data points with double exponential smoothing is

\[
\text{forecast}(t+m) = \text{DOUBLEXP}(t) + m \times b(t)
\]

where:

\( m \)

Is the number of time periods ahead for the forecast.

**Syntax:**

**FORECAST_DOUBLEXP**

\[
\text{FORECAST\_DOUBLEXP}(\text{display, infield, interval, npredict, npoint1, npoint2})
\]
where:

`display`  
Keyword  
Specifies which values to display for rows of output that represent existing data. Valid values are:

- **INPUT_FIELD.** This displays the original field values for rows that represent existing data.

- **MODEL_DATA.** This displays the calculated values for rows that represent existing data.

**Note:** You can show both types of output for any field by creating two independent COMPUTE commands in the same request, each with a different display option.

`infield`  
Is any numeric field. It can be the same field as the result field, or a different field. It cannot be a date-time field or a numeric field with date display options.

`interval`  
Is the increment to add to each sort field value (after the last data point) to create the next value. This must be a positive integer. To sort in descending order, use the BY HIGHEST phrase. The result of adding this number to the sort field values is converted to the same format as the sort field.

For date fields, the minimal component in the format determines how the number is interpreted. For example, if the format is YMD, MDY, or DMY, an interval value of 2 is interpreted as meaning two days. If the format is YM, the 2 is interpreted as meaning two months.

`npredict`  
Is the number of predictions for FORECAST to calculate. It must be an integer greater than or equal to zero. Zero indicates that you do not want predictions, and is only supported with a non-recursive FORECAST.

`nperiod`  
For the SEASONAL method, it is a positive whole number that specifies the number of data points in a period.

`npoint1`  
For DOUBLEXP, this number is used to calculate the weights for each component in the average. This value must be a positive whole number. The weight, k, is calculated by the following formula:

\[ k = \frac{2}{1 + npoint1} \]
For DOUBLEXP, this positive whole number is used to calculate the weights for each term in the trend. The weight, \( g \), is calculated by the following formula:

\[
g = \frac{2}{1 + n_{\text{point2}}} \]

**Example:** Calculating a Double Exponential Smoothing Column

The following sums the ACTUAL_YTD field of the CENTSTMT data source by period, and calculates a single exponential and double exponential moving average. The report columns show the calculated values for existing data points.

```
TABLE FILE CENTSTMT
SUM ACTUAL_YTD
COMPUTE EXP/D15.1 = FORECAST_EXPAVE(MODEL_DATA,ACTUAL_YTD,1,0,3);
DOUBLEXP/D15.1 = FORECAST_DOUBLEXP(MODEL_DATA,ACTUAL_YTD,1,0,3,3);
BY PERIOD
WHERE GL_ACCOUNT LIKE '3%%'
ON TABLE SET STYLE *
GRID=OFF,$
END
```

The output is shown in the following image:

<table>
<thead>
<tr>
<th>PERIOD</th>
<th>Actual</th>
<th>EXP</th>
<th>DOUBLEXP</th>
</tr>
</thead>
<tbody>
<tr>
<td>2002/01</td>
<td>12,957,681.</td>
<td>12,957,681.0</td>
<td>12,957,681.0</td>
</tr>
<tr>
<td>2002/02</td>
<td>25,441,971.</td>
<td>19,199,826.0</td>
<td>22,439,246.3</td>
</tr>
<tr>
<td>2002/03</td>
<td>39,164,321.</td>
<td>29,182,073.5</td>
<td>34,791,885.1</td>
</tr>
<tr>
<td>2002/04</td>
<td>52,733,326.</td>
<td>40,957,699.8</td>
<td>48,845,816.0</td>
</tr>
<tr>
<td>2002/05</td>
<td>66,765,920.</td>
<td>53,861,809.9</td>
<td>63,860,955.9</td>
</tr>
<tr>
<td>2002/06</td>
<td>80,952,492.</td>
<td>67,407,150.9</td>
<td>79,188,052.9</td>
</tr>
</tbody>
</table>

**FORECAST_SEASONAL: Using Triple Exponential Smoothing**

Triple exponential smoothing produces an exponential moving average that takes into account the tendency of data to repeat itself in intervals over time. For example, sales data that is growing and in which 25% of sales always occur during December contains both trend and seasonality. Triple exponential smoothing takes both the trend and seasonality into account by using three equations with three constants.
For triple exponential smoothing you need to know the number of data points in each time period (designated as L in the following equations). To account for the seasonality, a seasonal index is calculated. The data is divided by the prior season index and then used in calculating the smoothed average.

- The first equation accounts for the current time period, and is a weighted average of the current data value divided by the seasonal factor and the prior average adjusted for the trend for the previous period. The weight constant is k:

  \[ \text{SEASONAL}(t) = k \times \left( \frac{\text{data value}(t)}{I(t-L)} \right) + (1-k) \times \left( \text{SEASONAL}(t-1) + b(t-1) \right) \]

- The second equation is the calculated trend value, and is a weighted average of the difference between the current and previous average and the trend for the previous time period. b(t) represents the average trend. The weight constant is g:

  \[ b(t) = g \times (\text{SEASONAL}(t) - \text{SEASONAL}(t-1)) + (1-g) \times (b(t-1)) \]

- The third equation is the calculated seasonal index, and is a weighted average of the current data value divided by the current average and the seasonal index for the previous season. I(t) represents the average seasonal coefficient. The weight constant is p:

  \[ I(t) = p \times \left( \frac{\text{data value}(t)}{\text{SEASONAL}(t)} \right) + (1-p) \times I(t-L) \]

These equations are solved to derive the triple smoothed average. The first smoothed average is set to the first data value. Initial values for the seasonality factors are calculated based on the maximum number of full periods of data in the data source, while the initial trend is calculated based on two periods of data. These values are calculated with the following steps:

1. The initial trend factor is calculated by the following formula:

   \[ b(0) = \frac{1}{L} \left( \frac{(y(L+1)-y(1))}{L} + \frac{(y(L+2)-y(2))}{L} + \ldots + \frac{(y(2L)-y(L))}{L} \right) \]

2. The calculation of the initial seasonality factor is based on the average of the data values within each period, A(j) (1<=j<=N):

   \[ A(j) = \frac{y((j-1)L+1) + y((j-1)L+2) + \ldots + y(jL))}{L} \]

3. Then, the initial periodicity factor is given by the following formula, where N is the number of full periods available in the data, L is the number of points per period and n is a point within the period (1<= n <= L):

   \[ I(n) = \frac{y(n)/A(1) + y(n+L)/A(2) + \ldots + y((N-1)L+n)/A(N)}{N} \]
The three constants must be chosen carefully. The best results are usually obtained by choosing the constants to minimize the mean-squared error (MSE) between the data values and the calculated averages. Varying the values of npoint1 and npoint2 affect the results, and some values may produce a better approximation. To search for a better approximation, you may want to find values that minimize the MSE.

The equation used to forecast beyond the last data point with triple exponential smoothing is:

$$\text{forecast}(t+m) = \left( \text{SEASONAL}(t) + m \times b(t) \right) / I(t-L+\text{MOD}(m/L))$$

where:

$m$

Is the number of periods ahead for the forecast.

**Syntax:**

**How to Calculate a Triple Exponential Smoothing Column**

$$\text{FORECAST_SEASONAL}((\text{display}, \text{infield}, \text{interval}, \text{npredict}, \text{nperiod}, \text{npoint1}, \text{npoint2}, \text{npoint3}))$$

where:

**display**

Keyword

Specifies which values to display for rows of output that represent existing data. Valid values are:

- **INPUT_FIELD.** This displays the original field values for rows that represent existing data.
- **MODEL_DATA.** This displays the calculated values for rows that represent existing data.

**Note:** You can show both types of output for any field by creating two independent COMPUTE commands in the same request, each with a different display option.

**infield**

Is any numeric field. It can be the same field as the result field, or a different field. It cannot be a date-time field or a numeric field with date display options.

**interval**

Is the increment to add to each sort field value (after the last data point) to create the next value. This must be a positive integer. To sort in descending order, use the BY HIGHEST phrase. The result of adding this number to the sort field values is converted to the same format as the sort field.
For date fields, the minimal component in the format determines how the number is interpreted. For example, if the format is YMD, MDY, or DMY, an interval value of 2 is interpreted as meaning two days. If the format is YM, the 2 is interpreted as meaning two months.

npredict

Is the number of predictions for FORECAST to calculate. It must be an integer greater than or equal to zero. Zero indicates that you do not want predictions, and is only supported with a non-recursive FORECAST. For the SEASONAL method, npredict is the number of periods to calculate. The number of points generated is:

\[ n_{\text{period}} \times \text{npredict} \]

nperiod

For the SEASONAL method, is a positive whole number that specifies the number of data points in a period.

npoint1

For SEASONAL, this number is used to calculate the weights for each component in the average. This value must be a positive whole number. The weight, k, is calculated by the following formula:

\[ k = \frac{2}{1 + \text{npoint1}} \]

npoint2

For SEASONAL, this positive whole number is used to calculate the weights for each term in the trend. The weight, g, is calculated by the following formula:

\[ g = \frac{2}{1 + \text{npoint2}} \]

npoint3

For SEASONAL, this positive whole number is used to calculate the weights for each term in the seasonal adjustment. The weight, p, is calculated by the following formula:

\[ p = \frac{2}{1 + \text{npoint3}} \]
Example: Calculating a Triple Exponential Smoothing Column

In the following, the data has seasonality but no trend. Therefore, \( npoint2 \) is set high (1000) to make the trend factor negligible in the calculation:

```sql
TABLE FILE VIDEOTRK
SUM TRANSTOT
COMPUTE SEASONAL/D10.1 = FORECAST_SEASONAL(MODEL_DATA, TRANSTOT, 1,3,3,3,1000,1);
BY TRANSDATE
WHERE TRANSDATE NE '19910617'
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

In the output, \( npredict \) is 3. Therefore, three periods (nine points, \( nperiod \times npredict \)) are generated.

<table>
<thead>
<tr>
<th>TRANSDATE</th>
<th>TRANSTOT</th>
<th>SEASONAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>91/06/18</td>
<td>21.25</td>
<td>21.3</td>
</tr>
<tr>
<td>91/06/19</td>
<td>38.17</td>
<td>31.0</td>
</tr>
<tr>
<td>91/06/20</td>
<td>14.23</td>
<td>34.6</td>
</tr>
<tr>
<td>91/06/21</td>
<td>44.72</td>
<td>53.2</td>
</tr>
<tr>
<td>91/06/24</td>
<td>126.28</td>
<td>75.3</td>
</tr>
<tr>
<td>91/06/25</td>
<td>47.74</td>
<td>82.7</td>
</tr>
<tr>
<td>91/06/26</td>
<td>40.97</td>
<td>73.7</td>
</tr>
<tr>
<td>91/06/27</td>
<td>60.24</td>
<td>62.9</td>
</tr>
<tr>
<td>91/06/28</td>
<td>31.00</td>
<td>66.3</td>
</tr>
<tr>
<td>91/06/29</td>
<td></td>
<td>45.7</td>
</tr>
<tr>
<td>91/06/30</td>
<td></td>
<td>94.1</td>
</tr>
<tr>
<td>91/07/01</td>
<td></td>
<td>53.4</td>
</tr>
<tr>
<td>91/07/02</td>
<td></td>
<td>72.3</td>
</tr>
<tr>
<td>91/07/03</td>
<td></td>
<td>140.0</td>
</tr>
<tr>
<td>91/07/04</td>
<td></td>
<td>75.8</td>
</tr>
<tr>
<td>91/07/05</td>
<td></td>
<td>98.9</td>
</tr>
<tr>
<td>91/07/06</td>
<td></td>
<td>185.8</td>
</tr>
<tr>
<td>91/07/07</td>
<td></td>
<td>98.2</td>
</tr>
</tbody>
</table>
FORECAST_LINEAR: Using a Linear Regression Equation

The linear regression equation estimates values by assuming that the dependent variable (the new calculated values) and the independent variable (the sort field values) are related by a function that represents a straight line:

\[ y = mx + b \]

where:

- \( y \) is the dependent variable.
- \( x \) is the independent variable.
- \( m \) is the slope of the line.
- \( b \) is the y-intercept.

FORECAST_LINEAR uses a technique called Ordinary Least Squares to calculate values for \( m \) and \( b \) that minimize the sum of the squared differences between the data and the resulting line.

The following formulas show how \( m \) and \( b \) are calculated.

\[
    m = \frac{(\sum xy - (\sum x \cdot \sum y)/n)}{\left(\sum x^2 - (\sum x)^2/n\right)}
\]

\[
    b = \frac{(\sum y)/n - (m \cdot (\sum x)/n)}{n}
\]

where:

- \( n \) is the number of data points.
- \( y \) is the data values (dependent variables).
- \( x \) is the sort field values (independent variables).

Trend values, as well as predicted values, are calculated using the regression line equation.
How to Calculate a Linear Regression Column

FORECAST_LINEAR(display, infield, interval, npredict)

where:

display
Keyword

Specifies which values to display for rows of output that represent existing data. Valid values are:

- **INPUT_FIELD**. This displays the original field values for rows that represent existing data.
- **MODEL_DATA**. This displays the calculated values for rows that represent existing data.

**Note:** You can show both types of output for any field by creating two independent COMPUTE commands in the same request, each with a different display option.

infield
Is any numeric field. It can be the same field as the result field, or a different field. It cannot be a date-time field or a numeric field with date display options.

interval
Is the increment to add to each sort field value (after the last data point) to create the next value. This must be a positive integer. To sort in descending order, use the BY HIGHEST phrase. The result of adding this number to the sort field values is converted to the same format as the sort field.

For date fields, the minimal component in the format determines how the number is interpreted. For example, if the format is YMD, MDY, or DMY, an interval value of 2 is interpreted as meaning two days. If the format is YM, the 2 is interpreted as meaning two months.

npredict
Is the number of predictions for FORECAST to calculate. It must be an integer greater than or equal to zero. Zero indicates that you do not want predictions, and is only supported with a non-recursive FORECAST.
Example: Calculating a New Linear Regression Field

The following request calculates a regression line using the VIDEOTRK data source of QUANTITY by TRANSDATE. The interval is one day, and three predicted values are calculated.

```
TABLE FILE VIDEOTRK
SUM QUANTITY
COMPUTE FORTOT=FORECAST_LINEAR(MODEL_DATA,QUANTITY,1,3);
BY TRANSDATE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:

<table>
<thead>
<tr>
<th>TRANSDATE</th>
<th>QUANTITY</th>
<th>FORTOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>06/17/91</td>
<td>12</td>
<td>6.63</td>
</tr>
<tr>
<td>06/18/91</td>
<td>2</td>
<td>6.57</td>
</tr>
<tr>
<td>06/19/91</td>
<td>5</td>
<td>6.51</td>
</tr>
<tr>
<td>06/20/91</td>
<td>3</td>
<td>6.45</td>
</tr>
<tr>
<td>06/21/91</td>
<td>7</td>
<td>6.39</td>
</tr>
<tr>
<td>06/24/91</td>
<td>12</td>
<td>6.21</td>
</tr>
<tr>
<td>06/25/91</td>
<td>8</td>
<td>6.15</td>
</tr>
<tr>
<td>06/26/91</td>
<td>2</td>
<td>6.09</td>
</tr>
<tr>
<td>06/27/91</td>
<td>9</td>
<td>6.03</td>
</tr>
<tr>
<td>06/28/91</td>
<td>3</td>
<td>5.97</td>
</tr>
<tr>
<td>06/29/91</td>
<td></td>
<td>5.91</td>
</tr>
<tr>
<td>06/30/91</td>
<td></td>
<td>5.85</td>
</tr>
<tr>
<td>07/01/91</td>
<td></td>
<td>5.79</td>
</tr>
</tbody>
</table>

Note:

- Three predicted values of FORTOT are calculated. For values outside the range of the data, new TRANSDATE values are generated by adding the interval value (1) to the prior TRANSDATE value.
- There are no QUANTITY values for the generated FORTOT values.
- Each FORTOT value is computed using a regression line, calculated using all of the actual data values for QUANTITY.
TRANSDATE is the independent variable (x) and QUANTITY is the dependent variable (y). The equation is used to calculate QUANTITY FORECAST trend and predicted values.

The following version of the request charts the data values and the regression line.

```
GRAPH FILE VIDEOTRK
SUM QUANTITY
COMPUTE FORTOT=FORECAST_LINEAR(MODEL_DATA,QUANTITY,1,3);
BY TRANSDATE
ON GRAPH PCHOLD FORMAT JSCHART
ON GRAPH SET LOOKGRAPH VLINE
END
```

The output is shown in the following image.

![Graph showing the relationship between TRANSDATE and QUANTITY.](image)

**PARTITION_AGGR: Creating Rolling Calculations**

Using the PARTITION_AGGR function, you can generate rolling calculations based on a block of rows from the internal matrix of a TABLE request. In order to determine the limits of the rolling calculations, you specify a partition of the data based on either a sort field or the entire TABLE. Within either type of break, you can start calculating from the beginning of the break or a number of rows prior to or subsequent to the current row. You can stop the rolling calculation at the current row, a row past the start point, or the end of the partition.
By default, the field values used in the calculations are the summed values of a measure in the request. Certain prefix operators can be used to add a column to the internal matrix and use that column in the rolling calculations. The rolling calculation can be SUM, AVE, CNT, MIN, MAX, FST, or LST.

**Syntax:**

**How to Generate Rolling Calculations Using PARTITION_AGGR**

\[
\text{PARTITION\_AGGR}([\text{prefix.}]\text{measure, reset\_key, lower, upper, operation})
\]

where:

<table>
<thead>
<tr>
<th>prefix.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defines an aggregation operator to apply to the measure before using it in the rolling calculation. Valid operators are:</td>
</tr>
<tr>
<td>- <strong>SUM.</strong> which calculates the sum of the measure field values. SUM is the default operator.</td>
</tr>
<tr>
<td>- <strong>CNT.</strong> which calculates a count of the measure field values.</td>
</tr>
<tr>
<td>- <strong>AVE.</strong> which calculates the average of the measure field values.</td>
</tr>
<tr>
<td>- <strong>MIN.</strong> which calculates the minimum of the measure field values.</td>
</tr>
<tr>
<td>- <strong>MAX.</strong> which calculates the maximum of the measure field values.</td>
</tr>
<tr>
<td>- <strong>FST.</strong> which retrieves the first value of the measure field.</td>
</tr>
<tr>
<td>- <strong>LST.</strong> which retrieves the last value of the measure field.</td>
</tr>
</tbody>
</table>

**Note:** The operators PCT., RPCT., TOT., MDN., and DST. are not supported. COMPUTEs that reference those unsupported operators are also not supported.

<table>
<thead>
<tr>
<th>measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is the measure field to be aggregated. It can be a real field in the request or a calculated value generated with the COMPUTE command, as long as the COMPUTE does not reference an unsupported prefix operator.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>reset_key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifies the point at which the calculation restarts. Valid values are:</td>
</tr>
<tr>
<td>- The name of a sort field in the request.</td>
</tr>
<tr>
<td>- PRESET, which uses the value of the PARTITION_ON parameter, as described in <em>How to Specify the Partition Size for Simplified Statistical Functions</em> on page 469.</td>
</tr>
</tbody>
</table>
TABLE, which indicates that there is no break on a sort field.

The sort field may use BY HIGHEST to indicate a HIGH-TO-LOW sort. ACROSS COLUMNS
AND is supported. BY ROWS OVER and FOR are not supported.

lower
Identifies the starting point for the rolling calculation. Valid values are:

- \( n, -n \), which starts the calculation \( n \) rows forward or back from the current row.
- \( B \), which starts the calculation at the beginning of the current sort break (the first line
  with the same sort field value as the current line).

upper
Identifies the ending point of the rolling calculation. The lower row value must precede
upper row value.

Valid values are:

- \( C \), which ends the rolling calculation at the current row in the internal matrix.
- \( n, -n \), which ends the calculation \( n \) rows forward or back from the current row.
- \( E \), which ends the rolling calculation at the end of the sort break (the last line with the
  same sort value as the current row.)

**Note:** The values used in the calculations depend on the sort sequence (ascending or
descending) specified in the request. Be aware that displaying a date or time dimension in
descending order may produce different results than those you may expect.

operation
Specifies the rolling calculation used on the values in the internal matrix. Supported
operations are:

- **SUM.** which calculates a rolling sum.
- **AVE.** which calculates a rolling average.
- **CNT.** which counts the rows in the partition.
- **MIN.** which returns the minimum value in the partition.
- **MAX,** which returns the maximum value in the partition.
- **FST.** which returns the first value in the partition.
- **LST.** which returns the last value in the partition.
The calculation is performed prior to any WHERE TOTAL tests, but after any WHERE_GROUPED tests.

**Example:** Calculating a Rolling Average

The following request calculates a rolling average of the current line and the previous line in the internal matrix, within the quarter.

```
TABLE FILE WF_RETAIL_LITE
SUM COGS_US
COMPUTE AVE1/D12.2M = PARTITION_AGGR(COGS_US, TIME_QTR, -1, C, AVE);
BY BUSINESS_REGION
BY TIME_QTR
BY TIME_MTH
WHERE BUSINESS_REGION EQ 'North America' OR 'South America'
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```
The output is shown in the following image. Within each quarter, the first average is just the value from Q1, as going back 1 would cross a boundary. The second average is calculated using the first two rows within that quarter, and the third average is calculated using rows 2 and 3 within the quarter.

<table>
<thead>
<tr>
<th>Customer Business Region</th>
<th>Sale Quarter</th>
<th>Sale Month</th>
<th>Cost of Goods</th>
<th>AVE1</th>
</tr>
</thead>
<tbody>
<tr>
<td>North America</td>
<td>1</td>
<td>1</td>
<td>$26,361,956.00</td>
<td>$26,361,956.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>$24,348,729.00</td>
<td>$25,355,342.50</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>$26,118,420.00</td>
<td>$25,233,574.50</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>$23,776,352.00</td>
<td>$23,776,352.00</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5</td>
<td>$24,717,633.00</td>
<td>$24,246,992.50</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>6</td>
<td>$24,284,736.00</td>
<td>$24,501,184.50</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>$25,317,633.00</td>
<td>$25,317,633.00</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8</td>
<td>$25,916,286.00</td>
<td>$25,616,959.50</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>9</td>
<td>$24,968,297.00</td>
<td>$25,442,291.50</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>$30,717,478.00</td>
<td>$30,717,478.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>$30,055,782.00</td>
<td>$30,386,630.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td>$32,225,143.00</td>
<td>$31,140,462.50</td>
</tr>
<tr>
<td>South America</td>
<td>1</td>
<td>1</td>
<td>$3,216,999.00</td>
<td>$3,216,999.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>$2,745,677.00</td>
<td>$2,981,338.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>$3,163,526.00</td>
<td>$2,954,601.50</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>$2,852,809.00</td>
<td>$2,852,809.00</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5</td>
<td>$2,952,020.00</td>
<td>$2,902,414.50</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>6</td>
<td>$2,918,017.00</td>
<td>$2,935,018.50</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>$2,961,406.00</td>
<td>$2,961,406.00</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8</td>
<td>$3,077,824.00</td>
<td>$3,019,615.00</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>9</td>
<td>$2,895,280.00</td>
<td>$2,986,552.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>$3,642,505.00</td>
<td>$3,642,505.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>$3,482,327.00</td>
<td>$3,562,416.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td>$3,517,651.00</td>
<td>$3,499,989.00</td>
</tr>
</tbody>
</table>

The following changes the rolling average to start from the beginning of the sort break.

```sql
COMPUTE AVE1/D12.2M = PARTITION_AGG(COGS_US, TIME_QTR ,B, C, AVE);
```
The output is shown in the following image. Within each quarter, the first average is just the value from Q1, as going back would cross a boundary. The second average is calculated using the first two rows within that quarter, and the third average is calculated using rows 1 through 3 within the quarter.

<table>
<thead>
<tr>
<th>Customer Region</th>
<th>Sale Quarter</th>
<th>Sale Month</th>
<th>Cost of Goods</th>
<th>AVE1</th>
</tr>
</thead>
<tbody>
<tr>
<td>North America</td>
<td>1</td>
<td>1</td>
<td>$26,361,956.00</td>
<td>$26,361,956.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>$24,348,729.00</td>
<td>$25,355,342.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>$26,118,420.00</td>
<td>$25,609,701.67</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>$23,776,352.00</td>
<td>$23,776,352.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>$24,717,633.00</td>
<td>$24,246,992.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>$24,284,736.00</td>
<td>$24,259,573.67</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>$25,317,633.00</td>
<td>$25,317,633.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>$25,916,286.00</td>
<td>$25,616,959.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>$24,968,297.00</td>
<td>$25,400,738.67</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>$30,717,478.00</td>
<td>$30,717,478.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>$30,055,782.00</td>
<td>$30,386,630.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td>$32,225,143.00</td>
<td>$30,999,467.67</td>
</tr>
<tr>
<td>South America</td>
<td>1</td>
<td>1</td>
<td>$3,216,999.00</td>
<td>$3,216,999.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>$2,745,677.00</td>
<td>$2,981,338.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>$3,163,526.00</td>
<td>$3,042,067.33</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>$2,852,809.00</td>
<td>$2,852,809.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>$2,952,020.00</td>
<td>$2,902,414.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>$2,918,017.00</td>
<td>$2,907,615.33</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>$2,961,406.00</td>
<td>$2,961,406.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>$3,077,824.00</td>
<td>$3,019,615.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>$2,895,280.00</td>
<td>$2,978,170.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>$3,642,505.00</td>
<td>$3,642,505.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>$3,482,327.00</td>
<td>$3,562,416.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td>$3,517,651.00</td>
<td>$3,547,494.33</td>
</tr>
</tbody>
</table>

The following command uses the partition boundary TABLE.

```
COMPUTE AVE1/D12.2M = PARTITION_AGGR(COGS_US, TABLE, B, C, AVE);
```
The output is shown in the following image. The rolling average keeps adding the next row to the average with no sort field break.

<table>
<thead>
<tr>
<th>Customer Region</th>
<th>Sale Quarter</th>
<th>Sale Month</th>
<th>Cost of Goods</th>
<th>AVE1</th>
</tr>
</thead>
<tbody>
<tr>
<td>North America</td>
<td>1</td>
<td>1</td>
<td>$26,361,956.00</td>
<td>$26,361,956.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>$24,348,729.00</td>
<td>$25,355,342.50</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>$26,118,420.00</td>
<td>$25,609,701.67</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>$23,717,633.00</td>
<td>$25,151,364.25</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>$25,317,633.00</td>
<td>$24,989,351.29</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>$30,717,478.00</td>
<td>$25,652,752.00</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>6</td>
<td>$24,284,736.00</td>
<td>$24,934,637.67</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
<td>$25,317,633.00</td>
<td>$24,989,351.29</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>8</td>
<td>$25,916,286.00</td>
<td>$25,105,218.13</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td>$24,968,297.00</td>
<td>$25,090,004.67</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>10</td>
<td>$30,055,782.00</td>
<td>$26,053,027.45</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>12</td>
<td>$32,225,143.00</td>
<td>$26,567,370.42</td>
</tr>
<tr>
<td>South America</td>
<td>1</td>
<td>1</td>
<td>$3,216,999.00</td>
<td>$24,771,188.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>10</td>
<td>$2,745,677.00</td>
<td>$23,197,937.21</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>$3,163,526.00</td>
<td>$21,862,309.80</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>$2,852,809.00</td>
<td>$20,674,216.00</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5</td>
<td>$2,952,020.00</td>
<td>$19,631,733.88</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>6</td>
<td>$2,918,017.00</td>
<td>$18,703,194.06</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>7</td>
<td>$2,961,406.00</td>
<td>$17,874,678.89</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8</td>
<td>$3,077,824.00</td>
<td>$17,134,836.15</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>9</td>
<td>$2,895,280.00</td>
<td>$16,456,762.05</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>$3,642,505.00</td>
<td>$15,874,295.82</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>11</td>
<td>$3,482,327.00</td>
<td>$15,335,514.57</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>12</td>
<td>$3,517,651.00</td>
<td>$14,843,103.58</td>
</tr>
</tbody>
</table>

**Reference:** Usage Notes for PARTITION_AGGR

- Fields referenced in the PARTITION_AGGR parameters but not previously mentioned in the request will not be counted in column notation or propagated to HOLD files.
- Using the WITHIN phrase for a sum is the same as computing PARTITION_AGGR on the WITHIN sort field from B (beginning of sort break) to E (end of sort break) using SUM, as in the following example.
TABLE FILE WF_RETAIL_LITE
SUM COGS_US WITHIN TIME_QTR AS 'WITHIN Qtr'
COMPUTE PART_WITHIN_QTR/D12.2M = PARTITION_AGGR(COGS_US, TIME_QTR, B, E, SUM);
BY BUSINESS_REGION AS Region
BY TIME_QTR
BY TIME_MTH
WHERE BUSINESS_REGION EQ 'North America' OR 'South America'
ON TABLE SET PAGE NOPAGE
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END

The output is shown in the following image.

<table>
<thead>
<tr>
<th>Region</th>
<th>Sale Quarter</th>
<th>Sale Month</th>
<th>WITHIN_Qtr</th>
<th>PART_WITHIN_QTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>North America</td>
<td>1</td>
<td>1</td>
<td>$76,829,103.00</td>
<td>$76,829,105.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>$76,829,105.00</td>
<td>$76,829,105.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>$76,829,105.00</td>
<td>$76,829,105.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>$72,778,721.00</td>
<td>$72,778,721.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>$72,778,721.00</td>
<td>$72,778,721.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>$72,778,721.00</td>
<td>$72,778,721.00</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1</td>
<td>$76,202,216.00</td>
<td>$76,202,216.00</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1</td>
<td>$76,202,216.00</td>
<td>$76,202,216.00</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>1</td>
<td>$76,202,216.00</td>
<td>$76,202,216.00</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>$76,202,216.00</td>
<td>$76,202,216.00</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>1</td>
<td>$76,202,216.00</td>
<td>$76,202,216.00</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>10</td>
<td>$92,998,403.00</td>
<td>$92,998,403.00</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>10</td>
<td>$92,998,403.00</td>
<td>$92,998,403.00</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>10</td>
<td>$92,998,403.00</td>
<td>$92,998,403.00</td>
</tr>
<tr>
<td>South America</td>
<td>1</td>
<td>1</td>
<td>$9,126,202.00</td>
<td>$9,126,202.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>$9,126,202.00</td>
<td>$9,126,202.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>$9,126,202.00</td>
<td>$9,126,202.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>$8,722,846.00</td>
<td>$8,722,846.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>$8,722,846.00</td>
<td>$8,722,846.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>$8,722,846.00</td>
<td>$8,722,846.00</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1</td>
<td>$8,934,510.00</td>
<td>$8,934,510.00</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1</td>
<td>$8,934,510.00</td>
<td>$8,934,510.00</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>1</td>
<td>$8,934,510.00</td>
<td>$8,934,510.00</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>$8,934,510.00</td>
<td>$8,934,510.00</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>1</td>
<td>$8,934,510.00</td>
<td>$8,934,510.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>$10,642,483.00</td>
<td>$10,642,483.00</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>10</td>
<td>$10,642,483.00</td>
<td>$10,642,483.00</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>10</td>
<td>$10,642,483.00</td>
<td>$10,642,483.00</td>
</tr>
</tbody>
</table>
With other types of calculations, the results are not the same. For example, the following request calculates the average within quarter using the WITHIN phrase and the average within quarter using PARTITION_AGGR.

TABLE FILE WF_RETAIL_LITE
SUM COGS_US AS Cost
CNT.COGS_US AS Count AVE.COGS_US WITHIN TIME_QTR AS 'Ave Within'
COMPUTE PART_WITHIN_QTR/D12.2M = PARTITION_AGGR(COGS_US, TIME_QTR, B, E, AVE);
BY BUSINESS_REGION AS Region
BY TIME_QTR
ON TIME_QTR SUBTOTAL COGS_US CNT.COGS_US
BY TIME_MTH
WHERE BUSINESS_REGION EQ 'North America'
ON TABLE SET PAGE NOPAGE
END

The output is shown in the following image. The average using the WITHIN phrase divides the total cost for the quarter by the total count of instances for the quarter (for example, $76,829,105.00/252850 = $303.85), while PARTITION_AGGR divides the total cost for the quarter by the number of report rows in the quarter (for example, $76,829,105.00/3 = $25,609,701.67).

<table>
<thead>
<tr>
<th>Region</th>
<th>Quarter</th>
<th>Sale</th>
<th>Cost</th>
<th>Count</th>
<th>Ave Within</th>
<th>PART_WITHIN_QTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>North America</td>
<td>1</td>
<td>1</td>
<td>$26,361,956.00</td>
<td>86369</td>
<td>$303.85</td>
<td>$25,609,701.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>$24,348,729.00</td>
<td>79791</td>
<td>$303.85</td>
<td>$25,609,701.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>$26,118,420.00</td>
<td>86690</td>
<td>$303.85</td>
<td>$25,609,701.67</td>
</tr>
</tbody>
</table>

*TOTAL TIME_QTR 1
$76,829,105.00/ 252850 = $303.85

<table>
<thead>
<tr>
<th>Region</th>
<th>Quarter</th>
<th>Sale</th>
<th>Cost</th>
<th>Count</th>
<th>Ave Within</th>
<th>PART_WITHIN_QTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>North America</td>
<td>2</td>
<td>4</td>
<td>$23,776,352.00</td>
<td>79093</td>
<td>$303.40</td>
<td>$24,259,737.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>$24,717,633.00</td>
<td>81317</td>
<td>$303.40</td>
<td>$24,259,737.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>$24,384,736.00</td>
<td>79469</td>
<td>$303.40</td>
<td>$24,259,737.67</td>
</tr>
</tbody>
</table>

*TOTAL TIME_QTR 2
$72,778,721.00/ 239879 = $301.67

<table>
<thead>
<tr>
<th>Region</th>
<th>Quarter</th>
<th>Sale</th>
<th>Cost</th>
<th>Count</th>
<th>Ave Within</th>
<th>PART_WITHIN_QTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>North America</td>
<td>3</td>
<td>7</td>
<td>$25,317,633.00</td>
<td>82158</td>
<td>$308.06</td>
<td>$25,400,738.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>$25,916,286.00</td>
<td>83941</td>
<td>$308.06</td>
<td>$25,400,738.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>$24,968,297.00</td>
<td>81262</td>
<td>$308.06</td>
<td>$25,400,738.67</td>
</tr>
</tbody>
</table>

*TOTAL TIME_QTR 3
$76,202,216.00/ 247361 = $306.26

<table>
<thead>
<tr>
<th>Region</th>
<th>Quarter</th>
<th>Sale</th>
<th>Cost</th>
<th>Count</th>
<th>Ave Within</th>
<th>PART_WITHIN_QTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>North America</td>
<td>4</td>
<td>10</td>
<td>$30,717,478.00</td>
<td>99572</td>
<td>$309.47</td>
<td>$30,999,467.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>$30,055,782.00</td>
<td>97042</td>
<td>$309.47</td>
<td>$30,999,467.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td>$32,325,143.00</td>
<td>103898</td>
<td>$309.47</td>
<td>$30,999,467.67</td>
</tr>
</tbody>
</table>

*TOTAL TIME_QTR 4
$92,998,403.00/ 300512 = $309.66

TOTAL
$318,808,445.00/ 1040602 = $309.66
If you use PARTITION_AGGR to perform operations for specific time periods using an offset, for example, an operation on the quarters for different years, you must make sure that every quarter is represented. If some quarters are missing for some years, the offset will not access the correct data. In this case, generate a HOLD file that has every quarter represented for every year (you can use BY QUARTER ROWS OVER 1 OVER 2 OVER 3 OVER 4) and use PARTITION_AGGR on the HOLD file.

PARTITION_REF: Using Prior or Subsequent Field Values in Calculations

Use of LAST in a calculation retrieves the LAST value of the specified field the last time this calculation was performed. The PARTITION_REF function enables you to specify both how many rows back or forward to go in the output in order to retrieve a value, and a sort break within which the retrieval will be contained.

Syntax: How to Retrieve Prior or Subsequent Field Values for Use in a Calculation

\[
\text{PARTITION\_REF(}[\text{prefix.}]\text{field, reset\_key, offset})
\]

where:

\text{prefix}

Is optional. If used, it can be one of the following aggregation operators:

- \text{AVE.} Average
- \text{MAX.} Maximum
- \text{MIN.} Minimum
- \text{CNT.} Count
- \text{SUM.} Sum

\text{field}

Is the field whose value is to be retrieved.

\text{reset\_key}

Identifies the point at which the retrieval break restarts. Valid values are:

- The name of a sort field in the request.
- PRESET, which uses the value of the PARTITION\_ON parameter, as described in \textit{How to Specify the Partition Size for Simplified Statistical Functions} on page 469.
- TABLE, which indicates that there is no break on a sort field.
The sort field may use BY HIGHEST to indicate a HIGH-TO-LOW sort. ACROSS COLUMNS AND is supported. BY ROWS OVER and FOR are not supported.

**Note:** The values used in the retrieval depend on the sort sequence (ascending or descending) specified in the request. Be aware that displaying a date or time dimension in descending order may produce different results than those you may expect.

**offset**

Is the integer number of records to go forward (for a positive offset) or backward (for a negative offset) to retrieve the value.

If the offset is prior to the partition boundary sort value, the return will be the default value for the field. The calculation is performed prior to any WHERE TOTAL tests, but after WHERE_GROUPED tests.

**Example:** Retrieving a Previous Record With PARTITION_REF

The following request retrieves the previous record within the sort field PRODUCT_CATEGORY.

```plaintext
TABLE FILE WF_RETAIL_LITE
SUM DAYSDELAYED
COMPUTE NEWDAYS/I5=PARTITION_REF(DAYSDELAYED, PRODUCT_CATEGORY, -1);
BY PRODUCT_CATEGORY
BY PRODUCT_SUBCATEG
ON TABLE SET PAGE NOPAGE
ON TABLE SET STYLE *
GRID=OFF, $
ENDSTYLE
END
```
The output is shown in the following image. The first value within each sort break is zero because there is no prior record to retrieve.

<table>
<thead>
<tr>
<th>Product Category</th>
<th>Product Subcategory</th>
<th>Days Delayed</th>
<th>NEWDAYS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessories</td>
<td>Charger</td>
<td>12,301</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Headphones</td>
<td>26,670</td>
<td>12301</td>
</tr>
<tr>
<td></td>
<td>Universal Remote</td>
<td>20,832</td>
<td>26670</td>
</tr>
<tr>
<td>Camcorder</td>
<td>Handheld</td>
<td>29,446</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Professional</td>
<td>1,531</td>
<td>29446</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>22,248</td>
<td>1531</td>
</tr>
<tr>
<td>Computers</td>
<td>Smartphone</td>
<td>24,113</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Tablet</td>
<td>21,293</td>
<td>24113</td>
</tr>
<tr>
<td>Media Player</td>
<td>Blu Ray</td>
<td>78,989</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>DVD Players</td>
<td>31</td>
<td>78989</td>
</tr>
<tr>
<td></td>
<td>Streaming</td>
<td>8,153</td>
<td>31</td>
</tr>
<tr>
<td>Stereo Systems</td>
<td>Home Theater Systems</td>
<td>47,214</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Receivers</td>
<td>17,999</td>
<td>47214</td>
</tr>
<tr>
<td></td>
<td>Speaker Kits</td>
<td>28,468</td>
<td>17999</td>
</tr>
<tr>
<td></td>
<td>iPod Docking Station</td>
<td>37,556</td>
<td>28468</td>
</tr>
<tr>
<td>Televisions</td>
<td>Flat Panel TV</td>
<td>10,941</td>
<td>0</td>
</tr>
<tr>
<td>Video Production</td>
<td>Video Editing</td>
<td>23,553</td>
<td>0</td>
</tr>
</tbody>
</table>

The following request retrieves the average cost of goods from two records prior to the current record within the PRODUCTCATEGORY sort field.

```sql
TABLE FILE WF_RETAIL_LITE
SUM COGS_US AVE.COGS_US AS Average
COMPUTE PartitionAve/D12.2M=PARTITION_REF(AVE.COGS_US, PRODUCT_CATEGORY, -2);
BY PRODUCTCATEGORY
BY PRODUCT_SUBCATEG
ON TABLE SET PAGE NOPAGE
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```
PARTITION_REF: Using Prior or Subsequent Field Values in Calculations

The output is shown in the following image.

<table>
<thead>
<tr>
<th>Category</th>
<th>Subcategory</th>
<th>Cost of Goods</th>
<th>Average</th>
<th>PartitionAve</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessories</td>
<td>Charger</td>
<td>$2,052,711.00</td>
<td>$271.48</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Headphones</td>
<td>$51,663,564.00</td>
<td>$319.05</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Universal Remote Controls</td>
<td>$36,037,623.00</td>
<td>$285.21</td>
<td>$27.48</td>
</tr>
<tr>
<td>Camcorder</td>
<td>Handheld</td>
<td>$20,576,916.00</td>
<td>$116.02</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Professional</td>
<td>$35,218,308.00</td>
<td>$3,897.56</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>$49,071,633.00</td>
<td>$359.54</td>
<td>$116.02</td>
</tr>
<tr>
<td>Computers</td>
<td>Smartphone</td>
<td>$44,035,774.00</td>
<td>$302.01</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Tablet</td>
<td>$25,771,890.00</td>
<td>$247.89</td>
<td>$0.00</td>
</tr>
<tr>
<td>Media Player</td>
<td>Blu Ray</td>
<td>$181,112,921.00</td>
<td>$376.11</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>DVD Players</td>
<td>$3,756,254.00</td>
<td>$281.45</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>DVD Players - Portable</td>
<td>$306,576.00</td>
<td>$77.01</td>
<td>$376.11</td>
</tr>
<tr>
<td></td>
<td>Streaming</td>
<td>$5,064,730.00</td>
<td>$104.99</td>
<td>$281.45</td>
</tr>
<tr>
<td>Stereo Systems</td>
<td>Boom Box</td>
<td>$840,373.00</td>
<td>$125.67</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Home Theater Systems</td>
<td>$56,428,589.00</td>
<td>$199.38</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Receivers</td>
<td>$40,329,668.00</td>
<td>$377.67</td>
<td>$125.67</td>
</tr>
<tr>
<td></td>
<td>Speaker Kits</td>
<td>$81,396,140.00</td>
<td>$471.02</td>
<td>$199.38</td>
</tr>
<tr>
<td></td>
<td>iPod Docking Station</td>
<td>$26,119,093.00</td>
<td>$118.66</td>
<td>$377.67</td>
</tr>
<tr>
<td>Televisions</td>
<td>CRT TV</td>
<td>$1,928,416.00</td>
<td>$590.09</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Flat Panel TV</td>
<td>$59,077,345.00</td>
<td>$900.19</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Portable TV</td>
<td>$545,348.00</td>
<td>$93.74</td>
<td>$390.09</td>
</tr>
<tr>
<td>Video Production</td>
<td>Video Editing</td>
<td>$40,105,657.00</td>
<td>$283.23</td>
<td>$0.00</td>
</tr>
</tbody>
</table>
Replacing the function call with the following syntax changes the partition boundary to TABLE.

```sql
COMPUTE PartitionAve/D12.2M=PARTITION_REF(AVE.COGS_US, TABLE, -2);
```

The output is shown in the following image.

<table>
<thead>
<tr>
<th>Product Category</th>
<th>Product Subcategory</th>
<th>Cost of Goods</th>
<th>Average</th>
<th>PartitionAve</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessories</td>
<td>Charger</td>
<td>$2,032,711.00</td>
<td>$27.48</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Headphones</td>
<td>$1,663,564.00</td>
<td>$319.05</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Universal Remote</td>
<td>$36,037,623.00</td>
<td>$285.21</td>
<td>$27.48</td>
</tr>
<tr>
<td></td>
<td>Controls</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Camcorder</td>
<td>Handheld</td>
<td>$20,576,916.00</td>
<td>$116.02</td>
<td>$319.05</td>
</tr>
<tr>
<td></td>
<td>Professional</td>
<td>$35,218,308.00</td>
<td>$3,897.56</td>
<td>$285.21</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>$49,071,633.00</td>
<td>$359.54</td>
<td>$116.02</td>
</tr>
<tr>
<td>Computers</td>
<td>Smartphone</td>
<td>$44,035,774.00</td>
<td>$302.01</td>
<td>$3,897.56</td>
</tr>
<tr>
<td></td>
<td>Tablet</td>
<td>$25,771,890.00</td>
<td>$247.89</td>
<td>$359.54</td>
</tr>
<tr>
<td>Media Player</td>
<td>Blu Ray</td>
<td>$181,112,921.00</td>
<td>$376.11</td>
<td>$302.01</td>
</tr>
<tr>
<td></td>
<td>DVD Players</td>
<td>$3,756,254.00</td>
<td>$281.45</td>
<td>$247.89</td>
</tr>
<tr>
<td></td>
<td>DVD Players - Portable</td>
<td>$306,576.00</td>
<td>$77.01</td>
<td>$376.11</td>
</tr>
<tr>
<td></td>
<td>Streaming</td>
<td>$5,064,730.00</td>
<td>$104.99</td>
<td>$281.45</td>
</tr>
<tr>
<td>Stereo Systems</td>
<td>Boom Box</td>
<td>$840,373.00</td>
<td>$125.67</td>
<td>$77.01</td>
</tr>
<tr>
<td></td>
<td>Home Theater Systems</td>
<td>$56,428,589.00</td>
<td>$199.38</td>
<td>$104.99</td>
</tr>
<tr>
<td></td>
<td>Receivers</td>
<td>$40,329,668.00</td>
<td>$377.67</td>
<td>$125.67</td>
</tr>
<tr>
<td></td>
<td>Speaker Kits</td>
<td>$81,396,140.00</td>
<td>$471.02</td>
<td>$199.38</td>
</tr>
<tr>
<td></td>
<td>iPod Docking Station</td>
<td>$26,119,093.00</td>
<td>$118.66</td>
<td>$377.67</td>
</tr>
<tr>
<td>Televisions</td>
<td>CRT TV</td>
<td>$1,928,416.00</td>
<td>$590.09</td>
<td>$471.02</td>
</tr>
<tr>
<td></td>
<td>Flat Panel TV</td>
<td>$39,077,345.00</td>
<td>$900.19</td>
<td>$118.66</td>
</tr>
<tr>
<td></td>
<td>Portable TV</td>
<td>$345,348.00</td>
<td>$95.74</td>
<td>$590.09</td>
</tr>
<tr>
<td>Video Production</td>
<td>Video Editing</td>
<td>$40,105,657.00</td>
<td>$283.23</td>
<td>$900.19</td>
</tr>
</tbody>
</table>

**Reference:** Usage Notes for PARTITION_REF

- Fields referenced in the PARTITION_REF parameters but not previously mentioned in the request, will **not** be counted in column notation or propagated to HOLD files.

**INCREASE: Calculating the Difference Between the Current and a Prior Value of a Field**

Given an aggregated input field and a negative offset, INCREASE calculates the difference between the value in the current row of the report output and a prior row, within a sort break or the entire table. The reset point for the calculation is determined by the value of the PARTITION_ON parameter described in How to Specify the Partition Size for Simplified Statistical Functions on page 469.

Functions Reference 63
**Note:** The values used in the calculations depend on the sort sequence (ascending or descending) specified in the request. Be aware that displaying a date or time dimension in descending order may produce different results than those you may expect.

**Syntax:**

How to Calculate the Difference Between the Current and a Prior Value of a Field

\[
\text{INCREASE}([\text{prefix.}] \text{field}, \ offset)
\]

where:

- **prefix**
  Is one of the following optional aggregation operators to apply to the field before using it in the calculation:

  - **SUM.** which calculates the sum of the field values. SUM is the default value.
  - **CNT.** which calculates a count of the field values.
  - **AVE.** which calculates the average of the field values.
  - **MIN.** which calculates the minimum of the field values.
  - **MAX.** which calculates the maximum of the field values.
  - **FST.** which retrieves the first value of the field.
  - **LST.** which retrieves the last value of the field.

- **field**
  Numeric
  Is the field to be used in the calculation.

- **offset**
  Numeric
  Is a negative number indicating the number of rows back from the current row to use for the calculation.
**Example:** Calculating the Increase Between the Current and a Prior Value of a Field

The following request uses the default value of SET PARTITION_ON (PENULTIMATE) to calculate the increase within the PRODUCT_CATEGORY sort field between the current row and the previous row.

```
SET PARTITION_ON=PENULTIMATE
TABLE FILE wf_retail_lite
SUM QUANTITY_SOLD
COMPUTE INC = INCREASE(QUANTITY_SOLD, -1);
BY PRODUCT_CATEGORY
BY PRODUCT_SUBCATEG
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF, $
ENDSTYLE
END
```
The output is shown in the following image. The first value for INC is the value in the Accessories category for Quantity Sold, as there is no prior value. The second value for INC is the difference between the values for Headphones and Charger, the third is the difference between Universal Remote Controls and Headphones. Then, the calculations start over for Camcorder, which is the reset point.
PCT_INCREASE: Calculating the Percentage Difference Between the Current and a Prior Value of a Field

Given an aggregated input field and a negative offset, PCT_INCREASE calculates the percentage difference between the value in the current row of the report output and a prior row, within a sort break or the entire table. The reset point for the calculation is determined by the value of the PARTITION_ON parameter described in *How to Specify the Partition Size for Simplified Statistical Functions* on page 469.

The percentage increase is calculated using the following formula:

\[
\frac{\text{current\_value} - \text{prior\_value}}{\text{prior\_value}}
\]

**Note:** The values used in the calculations depend on the sort sequence (ascending or descending) specified in the request. Be aware that displaying a date or time dimension in descending order may produce different results than those you may expect.

**Syntax:**

How to Calculate the Percentage Difference Between the Current and a Prior Value of a Field

\[
\text{PCT\_INCREASE([prefix.]field, offset)}
\]

where:

- **prefix**
  - Is one of the following optional aggregation operators to apply to the field before using it in the calculation:
    - **SUM.** which calculates the sum of the field values. SUM is the default value.
    - **CNT.** which calculates a count of the field values.
    - **AVE.** which calculates the average of the field values.
    - **MIN.** which calculates the minimum of the field values.
    - **MAX.** which calculates the maximum of the field values.
    - **FST.** which retrieves the first value of the field.
    - **LST.** which retrieves the last value of the field.

- **field**
  - Numeric
  - The field to be used in the calculation.
offset
Numeric

Is a negative number indicating the number of rows back from the current row to use for the calculation.

Example: PCT_INCREASE: Calculating the Percent Increase Between the Current and a Prior Value of a Field

The following request uses the default value of SET PARTITION_ON (PENULTIMATE) to calculate the percent increase within the PRODUCT_CATEGORY sort field between the current row and the previous row.

```
SET PARTITION_ON=PENULTIMATE
TABLE FILE wf_retail_lite
SUM QUANTITY_SOLD
COMPUTE PCTINC/D8.2p = PCT_INCREASE(QUANTITY_SOLD,-1);
BY PRODUCT_CATEGORY
BY PRODUCT_SUBCATEG
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```
The output is shown in the following image. The first value for PCTINC is zero percent, as there is no prior value. The second value for PCTINC is the percent difference between the values for Headphones and Charger, the third is the percent difference between Universal Remote Controls and Headphones. Then, the calculations start over for Camcorder, which is the reset point.

<table>
<thead>
<tr>
<th>Product Category</th>
<th>Product Subcategory</th>
<th>Quantity Sold</th>
<th>PCTINC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessories</td>
<td>Charger</td>
<td>105,257</td>
<td>.00%</td>
</tr>
<tr>
<td></td>
<td>Headphones</td>
<td>228,349</td>
<td>116.94%</td>
</tr>
<tr>
<td></td>
<td>Universal Remote Controls</td>
<td>178,061</td>
<td>-22.02%</td>
</tr>
<tr>
<td>Camcorder</td>
<td>Handheld</td>
<td>250,167</td>
<td>.00%</td>
</tr>
<tr>
<td></td>
<td>Professional</td>
<td>12,872</td>
<td>-94.85%</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>192,205</td>
<td>1,393.20%</td>
</tr>
<tr>
<td>Computers</td>
<td>Smartphone</td>
<td>205,049</td>
<td>.00%</td>
</tr>
<tr>
<td></td>
<td>Tablet</td>
<td>146,728</td>
<td>-28.44%</td>
</tr>
<tr>
<td>Media Player</td>
<td>Blu Ray</td>
<td>679,495</td>
<td>.00%</td>
</tr>
<tr>
<td></td>
<td>DVD Players</td>
<td>18,835</td>
<td>-97.23%</td>
</tr>
<tr>
<td></td>
<td>DVD Players - Portable</td>
<td>5,694</td>
<td>-69.77%</td>
</tr>
<tr>
<td></td>
<td>Streaming</td>
<td>67,910</td>
<td>1,092.66%</td>
</tr>
<tr>
<td>Stereo Systems</td>
<td>Boom Box</td>
<td>9,370</td>
<td>.00%</td>
</tr>
<tr>
<td></td>
<td>Home Theater Systems</td>
<td>399,092</td>
<td>4,159.25%</td>
</tr>
<tr>
<td></td>
<td>Receivers</td>
<td>150,568</td>
<td>-62.27%</td>
</tr>
<tr>
<td></td>
<td>Speaker Kits</td>
<td>244,199</td>
<td>62.19%</td>
</tr>
<tr>
<td></td>
<td>iPod Docking Station</td>
<td>311,103</td>
<td>27.40%</td>
</tr>
<tr>
<td>Televisions</td>
<td>CRT TV</td>
<td>4,638</td>
<td>.00%</td>
</tr>
<tr>
<td></td>
<td>Flat Panel TV</td>
<td>92,501</td>
<td>1,894.42%</td>
</tr>
<tr>
<td></td>
<td>Portable TV</td>
<td>8,049</td>
<td>-91.30%</td>
</tr>
<tr>
<td>Video Production</td>
<td>Video Editing</td>
<td>199,749</td>
<td>.00%</td>
</tr>
</tbody>
</table>
PREVIOUS: Retrieving a Prior Value of a Field

Given an aggregated input field and a negative offset, PREVIOUS retrieves the value in a prior row, within a sort break or the entire table. The reset point for the calculation is determined by the value of the PARTITION_ON parameter described in How to Specify the Partition Size for Simplified Statistical Functions on page 469.

**Note:** The values used in the retrieval depend on the sort sequence (ascending or descending) specified in the request. Be aware that displaying a date or time dimension in descending order may produce different results than those you may expect.

**Syntax:**

How to Retrieve a Prior Value of a Field

PREVIOUS([prefix.]field, offset)

where:

- **prefix**
  - Is one of the following optional aggregation operators to apply to the field before using it in the calculation:
    - **SUM.** which calculates the sum of the field values. SUM is the default value.
    - **CNT.** which calculates a count of the field values.
    - **AVE.** which calculates the average of the field values.
    - **MIN.** which calculates the minimum of the field values.
    - **MAX.** which calculates the maximum of the field values.
    - **FST.** which retrieves the first value of the field.
    - **LST.** which retrieves the last value of the field.

- **field**
  - Numeric
  - The field to be used in the calculation.

- **offset**
  - Numeric
  - Is a negative number indicating the number of rows back from the current row to use for the retrieval.
Example: Retrieving a Prior Value of a Field

The following request sets the PARTITION_ON parameter to TABLE and retrieves the value of the QUANTITY_SOLD field two rows back from the current row.

```sql
SET PARTITION_ON=TABLE
TABLE FILE wf_retail_lite
SUM QUANTITY_SOLD
COMPUTE PREV = PREVIOUS(QUANTITY_SOLD, -2);
BY PRODUCT_CATEGORY
BY PRODUCT_SUBCATEG
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```
The output is shown in the following image. The value of PREV in the first two rows is zero, as there are no prior rows for retrieval. From then on, each value of PREV is from the QUANTITY_SOLD value from two rows prior, with no reset points.

<table>
<thead>
<tr>
<th>Product Category</th>
<th>Product Subcategory</th>
<th>Quantity Sold</th>
<th>PREV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessories</td>
<td>Charger</td>
<td>105,257</td>
<td>.00</td>
</tr>
<tr>
<td></td>
<td>Headphones</td>
<td>228,349</td>
<td>.00</td>
</tr>
<tr>
<td></td>
<td>Universal Remote Controls</td>
<td>178,061</td>
<td>105,257.00</td>
</tr>
<tr>
<td>Camcorder</td>
<td>Handheld</td>
<td>250,167</td>
<td>228,349.00</td>
</tr>
<tr>
<td></td>
<td>Professional</td>
<td>12,872</td>
<td>178,061.00</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>192,205</td>
<td>250,167.00</td>
</tr>
<tr>
<td>Computers</td>
<td>Smartphone</td>
<td>205,049</td>
<td>12,872.00</td>
</tr>
<tr>
<td></td>
<td>Tablet</td>
<td>146,728</td>
<td>192,205.00</td>
</tr>
<tr>
<td>Media Player</td>
<td>Blu Ray</td>
<td>679,495</td>
<td>205,049.00</td>
</tr>
<tr>
<td></td>
<td>DVD Players</td>
<td>18,835</td>
<td>146,728.00</td>
</tr>
<tr>
<td></td>
<td>DVD Players - Portable</td>
<td>5,694</td>
<td>679,495.00</td>
</tr>
<tr>
<td></td>
<td>Streaming</td>
<td>67,910</td>
<td>18,835.00</td>
</tr>
<tr>
<td>Stereo Systems</td>
<td>Boom Box</td>
<td>9,370</td>
<td>5,694.00</td>
</tr>
<tr>
<td></td>
<td>Home Theater Systems</td>
<td>399,092</td>
<td>67,910.00</td>
</tr>
<tr>
<td></td>
<td>Receivers</td>
<td>150,568</td>
<td>9,370.00</td>
</tr>
<tr>
<td></td>
<td>Speaker Kits</td>
<td>244,199</td>
<td>399,092.00</td>
</tr>
<tr>
<td></td>
<td>iPod Docking Station</td>
<td>311,103</td>
<td>150,568.00</td>
</tr>
<tr>
<td>Televisions</td>
<td>CRT TV</td>
<td>4,638</td>
<td>244,199.00</td>
</tr>
<tr>
<td></td>
<td>Flat Panel TV</td>
<td>92,501</td>
<td>311,103.00</td>
</tr>
<tr>
<td></td>
<td>Portable TV</td>
<td>8,049</td>
<td>4,638.00</td>
</tr>
<tr>
<td>Video Production</td>
<td>Video Editing</td>
<td>199,749</td>
<td>92,501.00</td>
</tr>
</tbody>
</table>

**RUNNING_AVE: Calculating an Average Over a Group of Rows**

Given an aggregated input field and a negative offset, RUNNING_AVE calculates the average of the values between the current row of the report output and a prior row, within a sort break or the entire table. The reset point for the calculation is determined by the sort field specified, the entire table, or the value of the PARTITION_ON parameter described in *How to Specify the Partition Size for Simplified Statistical Functions* on page 469.
Syntax: How to Calculate Running Average Between the Current and a Prior Value of a Field

RUNNING_AVE(field, reset_key, lower)

where:

field
    Numeric
    The field to be used in the calculation.

reset_key
    Identifies the point at which the running average restarts. Valid values are:
    - The name of a sort field in the request.
    - PRESET, which uses the value of the PARTITION_ON parameter, as described in How to Specify the Partition Size for Simplified Statistical Functions on page 469.
    - TABLE, which indicates that there is no break on a sort field.

Note: The values used in the calculations depend on the sort sequence (ascending or descending) specified in the request. Be aware that displaying a date or time dimension in descending order may produce different results than those you may expect.

lower
    Is the starting point in the partition for the running average. Valid values are:
    - A negative number, which identifies the offset from the current row.
    - B, which specifies the beginning of the sort group.

Example: Calculating a Running Average

The following request calculates a running average of QUANTITY_SOLD within the PRODUCT_CATEGORY sort field, always starting from the beginning of the sort break.

```sql
TABLE FILE wf_retail_lite
SUM QUANTITY_SOLD
COMPUTE RAVE = RUNNING_AVE(QUANTITY_SOLD, PRODUCT_CATEGORY, B);
BY PRODUCT_CATEGORY
BY PRODUCT_SUBCATEG
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```
The output is shown in the following image. The first value for RAVE is the value in the Accessories category for Quantity Sold, as there is no prior value. The second value for RAVE is the average of the values for Headphones and Charger, the third is the average of the values for Headphones, Charger, and Universal Remote Controls. Then, the calculations start over for Camcorder, which is the reset point.

<table>
<thead>
<tr>
<th>Product Category</th>
<th>Product Subcategory</th>
<th>Quantity Sold</th>
<th>RAVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessories</td>
<td>Charger</td>
<td>105,257</td>
<td>105,257.00</td>
</tr>
<tr>
<td></td>
<td>Headphones</td>
<td>228,349</td>
<td>166,803.00</td>
</tr>
<tr>
<td></td>
<td>Universal Remote Controls</td>
<td>178,061</td>
<td>170,555.00</td>
</tr>
<tr>
<td>Camcorder</td>
<td>Handheld</td>
<td>250,167</td>
<td>250,167.00</td>
</tr>
<tr>
<td></td>
<td>Professional</td>
<td>12,872</td>
<td>131,519.00</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>192,205</td>
<td>151,748.00</td>
</tr>
<tr>
<td>Computers</td>
<td>Smartphone</td>
<td>205,049</td>
<td>205,049.00</td>
</tr>
<tr>
<td></td>
<td>Tablet</td>
<td>146,728</td>
<td>175,888.00</td>
</tr>
<tr>
<td>Media Player</td>
<td>Blu Ray</td>
<td>679,495</td>
<td>679,495.00</td>
</tr>
<tr>
<td></td>
<td>DVD Players</td>
<td>18,835</td>
<td>349,165.00</td>
</tr>
<tr>
<td></td>
<td>DVD Players - Portable</td>
<td>5,694</td>
<td>234,674.00</td>
</tr>
<tr>
<td></td>
<td>Streaming</td>
<td>67,910</td>
<td>192,983.00</td>
</tr>
<tr>
<td>Stereo Systems</td>
<td>Boom Box</td>
<td>9,370</td>
<td>9,370.00</td>
</tr>
<tr>
<td></td>
<td>Home Theater Systems</td>
<td>399,092</td>
<td>204,231.00</td>
</tr>
<tr>
<td></td>
<td>Receivers</td>
<td>150,568</td>
<td>186,343.00</td>
</tr>
<tr>
<td></td>
<td>Speaker Kits</td>
<td>244,199</td>
<td>200,807.00</td>
</tr>
<tr>
<td></td>
<td>iPod Docking Station</td>
<td>311,103</td>
<td>222,866.00</td>
</tr>
<tr>
<td>Televisions</td>
<td>CRT TV</td>
<td>4,638</td>
<td>4,638.00</td>
</tr>
<tr>
<td></td>
<td>Flat Panel TV</td>
<td>92,501</td>
<td>48,569.00</td>
</tr>
<tr>
<td></td>
<td>Portable TV</td>
<td>8,049</td>
<td>35,062.00</td>
</tr>
<tr>
<td>Video Production</td>
<td>Video Editing</td>
<td>199,749</td>
<td>199,749.00</td>
</tr>
</tbody>
</table>
RUNNING_MAX: Calculating a Maximum Over a Group of Rows

Given an aggregated input field and an offset, RUNNING_MAX calculates the maximum of the values between the current row of the report output and a prior row, within a sort break or the entire table. The reset point for the calculation is determined by the sort field specified, the entire table, or the value of the PARTITION_ON parameter described in How to Specify the Partition Size for Simplified Statistical Functions on page 469.

Syntax: How to Calculate Running Maximum Between the Current and a Prior Value of a Field

\[
\text{RUNNING\_MAX}(\text{field, reset\_key, lower})
\]

where:

- **field**
  Numeric
  The field to be used in the calculation.

- **reset\_key**
  Identifies the point at which the running maximum restarts. Valid values are:
  
  - The name of a sort field in the request.
  
  - PRESET, which uses the value of the PARTITION_ON parameter, as described in How to Specify the Partition Size for Simplified Statistical Functions on page 469.
  
  - TABLE, which indicates that there is no break on a sort field.

  **Note**: The values used in the calculations depend on the sort sequence (ascending or descending) specified in the request. Be aware that displaying a date or time dimension in descending order may produce different results than those you may expect.

- **lower**
  Is the starting point in the partition for the running maximum. Valid values are:
  
  - A negative number, which identifies the offset from the current row.
  
  - B, which specifies the beginning of the sort group.
**Example: Calculating a Running Maximum**

The following request calculates a running maximum for the rows from the beginning of the table to the current value of QUANTITY_SOLD, with no reset point.

```sql
TABLE FILE wf_retail_lite
SUM QUANTITY_SOLD
COMPUTE RMAX = RUNNING_MAX(QUANTITY_SOLD, TABLE, B);
BY PRODUCT_CATEGORY
BY PRODUCT_SUBCATEG
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```
The output is shown in the following image. The first value for RMAX is the value in the Accessories category for Quantity Sold, as there is no prior value. The second value for RMAX is the value for Headphones, as that value is larger. The third value for RMAX is still the value for Headphones, as that value is larger than the Quantity Sold value in the third row. Since the maximum value in the table occurs for Blu Ray, that value is repeated on all future rows, as there is no reset point.

<table>
<thead>
<tr>
<th>Product Category</th>
<th>Product Subcategory</th>
<th>Quantity Sold</th>
<th>RMAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessories</td>
<td>Charger</td>
<td>105,257</td>
<td>105,257.00</td>
</tr>
<tr>
<td></td>
<td>Headphones</td>
<td>228,349</td>
<td>228,349.00</td>
</tr>
<tr>
<td></td>
<td>Universal Remote Controls</td>
<td>178,061</td>
<td>228,349.00</td>
</tr>
<tr>
<td>Camcorder</td>
<td>Handheld</td>
<td>250,167</td>
<td>250,167.00</td>
</tr>
<tr>
<td></td>
<td>Professional</td>
<td>12,872</td>
<td>250,167.00</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>192,205</td>
<td>250,167.00</td>
</tr>
<tr>
<td>Computers</td>
<td>Smartphone</td>
<td>205,049</td>
<td>250,167.00</td>
</tr>
<tr>
<td></td>
<td>Tablet</td>
<td>146,728</td>
<td>250,167.00</td>
</tr>
<tr>
<td>Media Player</td>
<td>Blu Ray</td>
<td>679,495</td>
<td>679,495.00</td>
</tr>
<tr>
<td></td>
<td>DVD Players</td>
<td>18,835</td>
<td>679,495.00</td>
</tr>
<tr>
<td></td>
<td>DVD Players - Portable</td>
<td>5,694</td>
<td>679,495.00</td>
</tr>
<tr>
<td></td>
<td>Streaming</td>
<td>67,910</td>
<td>679,495.00</td>
</tr>
<tr>
<td>Stereo Systems</td>
<td>Boom Box</td>
<td>9,370</td>
<td>679,495.00</td>
</tr>
<tr>
<td></td>
<td>Home Theater Systems</td>
<td>399,092</td>
<td>679,495.00</td>
</tr>
<tr>
<td></td>
<td>Receivers</td>
<td>150,568</td>
<td>679,495.00</td>
</tr>
<tr>
<td></td>
<td>Speaker Kits</td>
<td>244,199</td>
<td>679,495.00</td>
</tr>
<tr>
<td></td>
<td>iPod Docking Station</td>
<td>311,103</td>
<td>679,495.00</td>
</tr>
<tr>
<td>Televisions</td>
<td>CRT TV</td>
<td>4,638</td>
<td>679,495.00</td>
</tr>
<tr>
<td></td>
<td>Flat Panel TV</td>
<td>92,501</td>
<td>679,495.00</td>
</tr>
<tr>
<td></td>
<td>Portable TV</td>
<td>8,049</td>
<td>679,495.00</td>
</tr>
<tr>
<td>Video Production</td>
<td>Video Editing</td>
<td>199,749</td>
<td>679,495.00</td>
</tr>
</tbody>
</table>
RUNNING_MIN: Calculating a Minimum Over a Group of Rows

Given an aggregated input field and an offset, RUNNING_MIN calculates the minimum of the values between the current row of the report output and a prior row, within a sort break or the entire table. The reset point for the calculation is determined by the sort field specified, the entire table, or the value of the PARTITION_ON parameter described in How to Specify the Partition Size for Simplified Statistical Functions on page 469.

Syntax: How to Calculate Running Minimum Between the Current and a Prior Value of a Field

RUNNING_MIN(field, reset_key, lower)

where:

field
Numeric
The field to be used in the calculation.

reset_key
Identifies the point at which the running minimum restarts. Valid values are:

- The name of a sort field in the request.
- PRESET, which uses the value of the PARTITION_ON parameter, as described in How to Specify the Partition Size for Simplified Statistical Functions on page 469.
- TABLE, which indicates that there is no break on a sort field.

Note: The values used in the calculations depend on the sort sequence (ascending or descending) specified in the request. Be aware that displaying a date or time dimension in descending order may produce different results than those you may expect.

lower
Is the starting point in the partition for the running minimum. Valid values are:

- A negative number, which identifies the offset from the current row.
- B, which specifies the beginning of the sort group.

Example: Calculating a Running Minimum

The following request calculates a running minimum of QUANTITY_SOLD within the PRODUCT_CATEGORY sort field (the sort break defined by SET PARTITION_ON = PENULTIMATE), always starting from the beginning of the sort break.
SET PARTITION_ON=PENULTIMATE
TABLE FILE wf_retail_lite
SUM QUANTITY_SOLD
COMPUTE RMIN = \texttt{RUNNING\_MIN(QUANTITY\_SOLD,PRESET,B)};
BY PRODUCT\_CATEGORY
BY PRODUCT\_SUBCATEG
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
The output is shown in the following image. The first value for RMIN is the value in the Accessories category for Quantity Sold, as there is no prior value. The second value for RMIN is the value from the first row again (Charger), as that is smaller than the value in the second row. The third is the same again, as it is still the smallest. Then, the calculations start over for Camcorder, which is the reset point.

<table>
<thead>
<tr>
<th>Product Category</th>
<th>Product Subcategory</th>
<th>Quantity Sold</th>
<th>RMIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessories</td>
<td>Charger</td>
<td>105,257</td>
<td>105,257.00</td>
</tr>
<tr>
<td></td>
<td>Headphones</td>
<td>228,349</td>
<td>105,257.00</td>
</tr>
<tr>
<td></td>
<td>Universal Remote Controls</td>
<td>178,061</td>
<td>105,257.00</td>
</tr>
<tr>
<td>Camcorder</td>
<td>Handheld</td>
<td>250,167</td>
<td>250,167.00</td>
</tr>
<tr>
<td></td>
<td>Professional</td>
<td>12,872</td>
<td>12,872.00</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>192,205</td>
<td>12,872.00</td>
</tr>
<tr>
<td>Computers</td>
<td>Smartphone</td>
<td>205,049</td>
<td>205,049.00</td>
</tr>
<tr>
<td></td>
<td>Tablet</td>
<td>146,728</td>
<td>146,728.00</td>
</tr>
<tr>
<td>Media Player</td>
<td>Blu Ray</td>
<td>679,495</td>
<td>679,495.00</td>
</tr>
<tr>
<td></td>
<td>DVD Players</td>
<td>18,835</td>
<td>18,835.00</td>
</tr>
<tr>
<td></td>
<td>DVD Players - Portable</td>
<td>5,694</td>
<td>5,694.00</td>
</tr>
<tr>
<td></td>
<td>Streaming</td>
<td>67,910</td>
<td>5,694.00</td>
</tr>
<tr>
<td>Stereo Systems</td>
<td>Boom Box</td>
<td>9,370</td>
<td>9,370.00</td>
</tr>
<tr>
<td></td>
<td>Home Theater Systems</td>
<td>399,092</td>
<td>9,370.00</td>
</tr>
<tr>
<td></td>
<td>Receivers</td>
<td>150,568</td>
<td>9,370.00</td>
</tr>
<tr>
<td></td>
<td>Speaker Kits</td>
<td>244,199</td>
<td>9,370.00</td>
</tr>
<tr>
<td></td>
<td>iPod Docking Station</td>
<td>311,103</td>
<td>9,370.00</td>
</tr>
<tr>
<td>Televisions</td>
<td>CRT TV</td>
<td>4,638</td>
<td>4,638.00</td>
</tr>
<tr>
<td></td>
<td>Flat Panel TV</td>
<td>92,501</td>
<td>4,638.00</td>
</tr>
<tr>
<td></td>
<td>Portable TV</td>
<td>8,049</td>
<td>4,638.00</td>
</tr>
<tr>
<td>Video Production</td>
<td>Video Editing</td>
<td>199,749</td>
<td>199,749.00</td>
</tr>
</tbody>
</table>
RUNNING_SUM: Calculating a Sum Over a Group of Rows

Given an aggregated input field and an offset, RUNNING_SUM calculates the sum of the values between the current row of the report output and a prior row, within a sort break or the entire table. The reset point for the calculation is determined by the sort field specified, the entire table, or the value of the PARTITION_ON parameter described in *How to Specify the Partition Size for Simplified Statistical Functions* on page 469.

**Syntax:** How to Calculate Running Sum Between the Current and a Prior Value of a Field

\[
\text{RUNNING\_SUM}(\text{field}, \text{reset\_key}, \text{lower})
\]

where:

- **field**
  - Numeric
  - The field to be used in the calculation.

- **reset_key**
  - Identifies the point at which the running sum restarts. Valid values are:
    - The name of a sort field in the request.
    - PRESET, which uses the value of the PARTITION_ON parameter, as described in *How to Specify the Partition Size for Simplified Statistical Functions* on page 469.
    - TABLE, which indicates that there is no break on a sort field.

  **Note:** The values used in the calculations depend on the sort sequence (ascending or descending) specified in the request. Be aware that displaying a date or time dimension in descending order may produce different results than those you may expect.

- **lower**
  - Is the starting point in the partition for the running sum. Valid values are:
    - A negative number, which identifies the offset from the current row.
    - B, which specifies the beginning of the sort group.
**Example: Calculating a Running Sum**

The following request calculates a running sum of the current value and previous value of QUANTITY_SOLD within the reset point set by the PARTITION_ON parameter, which is the sort field PRODUCT_CATEGORY.

```
SET PARTITION_ON=PENDULTIMATE
TABLE FILE w_f_retail_lite
SUM QUANTITY_SOLD
COMPUTE RSUM = RUNNING_SUM(QUANTITY_SOLD, PRESET, -1);
BY PRODUCTCATEGORY
BY PRODUCT_SUBCATEG
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```
The output is shown in the following image. The first value for RSUM is the value in the Accessories category for Quantity Sold, as there is no prior value. The second value for RSUM is the sum of the values for Headphones and Charger, the third is the sum of the values for Charger and Universal Remote Controls. Then, the calculations start over for Camcorder, which is the reset point.

<table>
<thead>
<tr>
<th>Product Category</th>
<th>Product Subcategory</th>
<th>Quantity Sold</th>
<th>RSUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessories</td>
<td>Charger</td>
<td>105,257</td>
<td>105,257.00</td>
</tr>
<tr>
<td></td>
<td>Headphones</td>
<td>228,349</td>
<td>333,606.00</td>
</tr>
<tr>
<td></td>
<td>Universal Remote Controls</td>
<td>178,061</td>
<td>406,410.00</td>
</tr>
<tr>
<td>Camcorder</td>
<td>Handheld</td>
<td>250,167</td>
<td>250,167.00</td>
</tr>
<tr>
<td></td>
<td>Professional</td>
<td>12,872</td>
<td>263,039.00</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>192,205</td>
<td>205,077.00</td>
</tr>
<tr>
<td>Computers</td>
<td>Smartphone</td>
<td>205,049</td>
<td>205,049.00</td>
</tr>
<tr>
<td></td>
<td>Tablet</td>
<td>146,728</td>
<td>351,777.00</td>
</tr>
<tr>
<td>Media Player</td>
<td>Blu Ray</td>
<td>679,495</td>
<td>679,495.00</td>
</tr>
<tr>
<td></td>
<td>DVD Players</td>
<td>18,835</td>
<td>698,330.00</td>
</tr>
<tr>
<td></td>
<td>DVD Players - Portable</td>
<td>5,694</td>
<td>24,529.00</td>
</tr>
<tr>
<td></td>
<td>Streaming</td>
<td>67,910</td>
<td>73,604.00</td>
</tr>
<tr>
<td>Stereo Systems</td>
<td>Boom Box</td>
<td>9,370</td>
<td>9,370.00</td>
</tr>
<tr>
<td></td>
<td>Home Theater Systems</td>
<td>399,092</td>
<td>408,462.00</td>
</tr>
<tr>
<td></td>
<td>Receivers</td>
<td>150,568</td>
<td>549,660.00</td>
</tr>
<tr>
<td></td>
<td>Speaker Kits</td>
<td>244,199</td>
<td>394,767.00</td>
</tr>
<tr>
<td></td>
<td>iPod Docking Station</td>
<td>311,103</td>
<td>555,302.00</td>
</tr>
<tr>
<td>Televisions</td>
<td>CRT TV</td>
<td>4,638</td>
<td>4,638.00</td>
</tr>
<tr>
<td></td>
<td>Flat Panel TV</td>
<td>92,501</td>
<td>97,139.00</td>
</tr>
<tr>
<td></td>
<td>Portable TV</td>
<td>8,049</td>
<td>100,550.00</td>
</tr>
<tr>
<td>Video Production</td>
<td>Video Editing</td>
<td>199,749</td>
<td>199,749.00</td>
</tr>
</tbody>
</table>
Simplified Character Functions

Simplified character functions have streamlined parameter lists, similar to those used by SQL functions. In some cases, these simplified functions provide slightly different functionality than previous versions of similar functions.

The simplified functions do not have an output argument. Each function returns a value that has a specific data type.

When used in a request against a relational data source, these functions are optimized (passed to the RDBMS for processing).

**Note:** The simplified character functions are not supported in Maintain Data.

**In this chapter:**

- **CHAR_LENGTH:** Returning the Length in Characters of a String
- **CONCAT:** Concatenating Strings After Removing Trailing Blanks From the First
- **DIGITS:** Converting a Number to a Character String
- **GET_TOKEN:** Extracting a Token Based on a String of Delimiters
- **INITCAP:** Capitalizing the First Letter of Each Word in a String
- **LAST_NONBLANK:** Retrieving the Last Field Value That is Neither Blank nor Missing
- **LOWER:** Returning a String With All Letters Lowercase
- **LPAD:** Left-Padding a Character String
- **LTRIM:** Removing Blanks From the Left End of a String
- **POSITION:** Returning the First Position of a Substring in a Source String
- **REGEX:** Matching a String to a Regular Expression
- **REPLACE:** Replacing a String
- **RPAD:** Right-Padding a Character String
- **RTRIM:** Removing Blanks From the Right End of a String
- **SPLIT:** Extracting an Element From a String
- **SUBSTRING:** Extracting a Substring From a Source String
- **TOKEN:** Extracting a Token From a String
- **TRIM_:** Removing a Leading Character, Trailing Character, or Both From a String
- **UPPER:** Returning a String With All Letters Uppercase
**CHAR_LENGTH: Returning the Length in Characters of a String**

The CHAR_LENGTH function returns the length, in characters, of a string. In Unicode environments, this function uses character semantics, so that the length in characters may not be the same as the length in bytes. If the string includes trailing blanks, these are counted in the returned length. Therefore, if the format source string is type An, the returned value will always be n.

**Syntax:**

**How to Return the Length of a String in Characters**

\[
\text{CHAR_LENGTH}(\text{string})
\]

where:

- **string**
  - Alphanumeric
  - Is the string whose length is returned.

The data type of the returned length value is Integer.

**Example:**

**Returning the Length of a String**

The following request against the EMPLOYEE data source creates a virtual field named LASTNAME of type A15V that contains the LAST_NAME with the trailing blanks removed. It then uses CHAR_LENGTH to return the number of characters.

```
DEFINE FILE EMPLOYEE
LASTNAME/A15V = RTRIM(LAST_NAME);
END

TABLE FILE EMPLOYEE
SUM LAST_NAME NOPRINT AND COMPUTE
NAME_LEN/I3 = CHAR_LENGTH(LASTNAME);
BY LAST_NAME
ON TABLE SET PAGE NOPAGE
END
```
The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>NAME_LEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>7</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>9</td>
</tr>
<tr>
<td>CROSS</td>
<td>5</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>9</td>
</tr>
<tr>
<td>IRVING</td>
<td>6</td>
</tr>
<tr>
<td>JONES</td>
<td>5</td>
</tr>
<tr>
<td>MCCOY</td>
<td>5</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>8</td>
</tr>
<tr>
<td>ROMANS</td>
<td>6</td>
</tr>
<tr>
<td>SMITH</td>
<td>5</td>
</tr>
<tr>
<td>STEVENS</td>
<td>7</td>
</tr>
</tbody>
</table>

LASTNAME has format A15V and contains the last name with trailing blanks removed. CHAR_LENGTH returns the number of characters:

CHAR_LENGTH (LASTNAME)

For SMITH, the result is 5.

**CONCAT: Concatenating Strings After Removing Trailing Blanks From the First**

CONCAT removes trailing blanks from a string and then concatenates another string to it. The output is returned as variable length alphanumeric.

**Syntax:**

How to Concatenate Strings After Removing Trailing Blanks From the First

CONCAT(string1, string2)

where:

**string2**

Alphanumeric

Is a string whose trailing blanks will be removed.

**string1**

Alphanumeric

Is a string whose leading and trailing blanks will be preserved.
Example: Concatenating Strings After Removing Blanks From the First

The following request concatenates city names with state names. Note that the city and state names are converted to fixed length alphanumeric fields before concatenation.

```
DEFINE FILE WF_RETAIL_LITE
CITY/A50 = CITY_NAME;
STATE/A50 = STATE_PROV_NAME;
CONCAT_CS/A100 = CONCAT(CITY,STATE);
END

TABLE FILE WF_RETAIL_LITE
SUM CITY AS City STATE AS State CONCAT_CS AS Concatenation
BY STATE_PROV_NAME NOPRINT
WHERE COUNTRY_NAME EQ 'United States'
WHERE STATE LE 'Louisiana'
ON TABLE SET PAGE NOLEAD
END
```
The output is shown in the following image.

<table>
<thead>
<tr>
<th>City</th>
<th>State</th>
<th>Concatenation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Montgomery</td>
<td>Alabama</td>
<td>MontgomeryAlabama</td>
</tr>
<tr>
<td>Anchorage</td>
<td>Alaska</td>
<td>AnchorageAlaska</td>
</tr>
<tr>
<td>Phoenix</td>
<td>Arizona</td>
<td>PhoenixArizona</td>
</tr>
<tr>
<td>Little Rock</td>
<td>Arkansas</td>
<td>Little RockArkansas</td>
</tr>
<tr>
<td>Saratoga</td>
<td>California</td>
<td>SaratogaCalifornia</td>
</tr>
<tr>
<td>Colorado Springs</td>
<td>Colorado</td>
<td>Colorado SpringsColorado</td>
</tr>
<tr>
<td>Old Lyme</td>
<td>Connecticut</td>
<td>Old LymeConnecticut</td>
</tr>
<tr>
<td>Lewes</td>
<td>Delaware</td>
<td>LewesDelaware</td>
</tr>
<tr>
<td>Washington</td>
<td>District of Columbia</td>
<td>WashingtonDistrict of Columbia</td>
</tr>
<tr>
<td>Miami</td>
<td>Florida</td>
<td>MiamiFlorida</td>
</tr>
<tr>
<td>Atlanta</td>
<td>Georgia</td>
<td>AtlantaGeorgia</td>
</tr>
<tr>
<td>Honolulu</td>
<td>Hawaii</td>
<td>HonoluluHawaii</td>
</tr>
<tr>
<td>Boise</td>
<td>Idaho</td>
<td>BoiseIdaho</td>
</tr>
<tr>
<td>Chicago</td>
<td>Illinois</td>
<td>ChicagoIllinois</td>
</tr>
<tr>
<td>Indianapolis</td>
<td>Indiana</td>
<td>IndianapolisIndiana</td>
</tr>
<tr>
<td>Des Moines</td>
<td>Iowa</td>
<td>Des MoinesIowa</td>
</tr>
<tr>
<td>Wichita</td>
<td>Kansas</td>
<td>WichitaKansas</td>
</tr>
<tr>
<td>Lexington</td>
<td>Kentucky</td>
<td>LexingtonKentucky</td>
</tr>
<tr>
<td>New Orleans</td>
<td>Louisiana</td>
<td>New OrleansLouisiana</td>
</tr>
</tbody>
</table>

**CONCAT** concatenates CITY and STATE.

**CONCAT (CITY, STATE)**

For Montgomery Alabama, the result is MontgomeryAlabama.
**DIGITS: Converting a Number to a Character String**

Given a number, DIGITS converts it to a character string of the specified length. The format of the field that contains the number must be Integer.

**Syntax:**

How to Convert a Number to a Character String

```
DIGITS(number, length)
```

where:

- **number**
  
  Integer
  
  Is the number to be converted, stored in a field with data type Integer.

- **length**
  
  Integer between 1 and 10
  
  Is the length of the returned character string. If **length** is longer than the number of digits in the number being converted, the returned value is padded on the left with zeros. If **length** is shorter than the number of digits in the number being converted, the returned value is truncated on the left.

**Example:**

Converting a Number to a Character String

The following request against the WF_RETAIL_LITE data source converts -123.45 and ID_PRODUCT to character strings:

```
DEFINE FILE WF_RETAIL_LITE
MEAS1/I8=-123.45;
DIG1/A6=DIGITS(MEAS1,6) ;
DIG2/A6=DIGITS(ID_PRODUCT,6) ;
END
TABLE FILE WF_RETAIL_LITE
PRINT MEAS1 DIG1
ID_PRODUCT DIG2
BY PRODUCT_SUBCATEG
WHERE PRODUCT_SUBCATEG EQ 'Flat Panel TV'
ON TABLE SET PAGE NOPAGE
END
```
The output is:

<table>
<thead>
<tr>
<th>Product Subcategory</th>
<th>MEAS1</th>
<th>DIG1</th>
<th>ID Product</th>
<th>DIG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flat Panel TV</td>
<td>-123</td>
<td>000123</td>
<td>4012</td>
<td>004012</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4017</td>
<td>004017</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4018</td>
<td>004018</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4017</td>
<td>004017</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4017</td>
<td>004017</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4018</td>
<td>004018</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4018</td>
<td>004018</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4017</td>
<td>004017</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4014</td>
<td>004014</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4016</td>
<td>004016</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4016</td>
<td>004016</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4018</td>
<td>004018</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4017</td>
<td>004017</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4018</td>
<td>004018</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4016</td>
<td>004016</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4018</td>
<td>004018</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4017</td>
<td>004017</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4018</td>
<td>004018</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4017</td>
<td>004017</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4017</td>
<td>004017</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4014</td>
<td>004014</td>
</tr>
<tr>
<td></td>
<td>-123</td>
<td>000123</td>
<td>4018</td>
<td>004018</td>
</tr>
</tbody>
</table>
DIGITS converts the integer expression ID_PRODUCT+1 to a six-character string:

\[
\text{DIGITS}(\text{ID\_PRODUCT, 6})
\]

For the number 1106, the result is the character string '001106'.

**Reference:** Usage Notes for DIGITS

- Only I format numbers will be converted. D, P, and F formats generate error messages and should be converted to I before using the DIGITS function. The limit for the number that can be converted is 2 GB.
- Negative integers are turned into positive integers.
- Integer formats with decimal places are truncated.
- DIGITS is not supported in Dialogue Manager.

**GET_TOKEN: Extracting a Token Based on a String of Delimiters**

GET_TOKEN extracts a token (substring) based on a string that can contain multiple characters, each of which represents a single-character delimiter.

GET_TOKEN can be optimized if there is a single delimiter character, not a string containing multiple delimiter characters.

**Syntax:** How to Extract a Token Based on a String of Delimiters

\[
\text{GET\_TOKEN}(\text{string, delimiter\_string, occurrence})
\]

where:

- **string**
  
  Alphanumeric

  Is the input string from which the token will be extracted. This can be an alphanumeric field or constant.

- **delimiter\_string**
  
  Alphanumeric constant

  Is a string that contains the list of delimiter characters. For example, '; ;' contains three delimiter characters, semi-colon, blank space, and comma.
**occurrence**

Integer constant

Is a positive integer that specifies the token to be extracted. A negative integer will be accepted in the syntax, but will not extract a token. The value zero (0) is not supported.

**Example:** Extracting a Token Based on a String of Delimiters

The following request defines an input string and two tokens based on a list of delimiters that contains the characters comma (,), semicolon (;), and slash (/).

```plaintext
DEFINE FILE EMPLOYEE
InputString/A20 = 'ABC,DEF;GHI/JKL';
FirstToken/A20 WITH DEPARTMENT = GET_TOKEN(InputString, ',;/', 1);
FourthToken/A20 WITH DEPARTMENT = GET_TOKEN(InputString, ',;/', 4);
ENDTABLE FILE EMPLOYEE
PRINT InputString FirstToken FourthToken
WHERE READLIMIT EQ 1
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID = OFF,$
END
```

The output is shown in the following image. The first token was extracted using the comma (,) as the delimiter. The fourth token was extracted using the slash (/) as the delimiter.

<table>
<thead>
<tr>
<th>InputString</th>
<th>FirstToken</th>
<th>FourthToken</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC,DEF;GHI/JKL</td>
<td>ABC</td>
<td>JKL</td>
</tr>
</tbody>
</table>

GET_TOKEN extracts a token based on a string of delimiters.

```plaintext
GET_TOKEN(InputString, ',;/', 4)
```

For input string 'ABC,DEF;GHI/JKL', the result is JKL.

**INITCAP: Capitalizing the First Letter of Each Word in a String**

INITCAP capitalizes the first letter of each word in an input string and makes all other letters lowercase. A word starts at the beginning of the string, after a blank space, or after a special character.

**Syntax:** How to Capitalize the First Letter of Each Word in a String

```plaintext
INITCAP(input_string)
```
INITCAP: Capitalizing the First Letter of Each Word in a String

where:

\[ \text{input\_string} \]

Alphanumeric

Is the string to capitalize.

**Example: Capitalizing the First Letter of Each Word in a String**

The following request changes the last names in the EMPLOYEE data source to initial caps and capitalizes the first letter after each blank or special character in the NewName field.

```plaintext
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND COMPUTE
Caps1/A30 = INITCAP(LAST_NAME);
NewName/A30 = 'abc,def!ghi'jKL MNO';
Caps2/A30 = INITCAP(NewName);
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>Caps1</th>
<th>NewName</th>
<th>Caps2</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEVENS</td>
<td>Stevens</td>
<td>abc,def!ghi'jKL MNO</td>
<td>Abc,Def!Ghi'Jkl Mno</td>
</tr>
<tr>
<td>SMITH</td>
<td>Smith</td>
<td>abc,def!ghi'jKL MNO</td>
<td>Abc,Def!Ghi'Jkl Mno</td>
</tr>
<tr>
<td>JONES</td>
<td>Jones</td>
<td>abc,def!ghi'jKL MNO</td>
<td>Abc,Def!Ghi'Jkl Mno</td>
</tr>
<tr>
<td>SMITH</td>
<td>Smith</td>
<td>abc,def!ghi'jKL MNO</td>
<td>Abc,Def!Ghi'Jkl Mno</td>
</tr>
<tr>
<td>BANNING</td>
<td>Banning</td>
<td>abc,def!ghi'jKL MNO</td>
<td>Abc,Def!Ghi'Jkl Mno</td>
</tr>
<tr>
<td>IRVING</td>
<td>Irving</td>
<td>abc,def!ghi'jKL MNO</td>
<td>Abc,Def!Ghi'Jkl Mno</td>
</tr>
<tr>
<td>ROMANS</td>
<td>Romans</td>
<td>abc,def!ghi'jKL MNO</td>
<td>Abc,Def!Ghi'Jkl Mno</td>
</tr>
<tr>
<td>MCCOY</td>
<td>Mccoy</td>
<td>abc,def!ghi'jKL MNO</td>
<td>Abc,Def!Ghi'Jkl Mno</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>Blackwood</td>
<td>abc,def!ghi'jKL MNO</td>
<td>Abc,Def!Ghi'Jkl Mno</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>Mcknight</td>
<td>abc,def!ghi'jKL MNO</td>
<td>Abc,Def!Ghi'Jkl Mno</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>Greenspan</td>
<td>abc,def!ghi'jKL MNO</td>
<td>Abc,Def!Ghi'Jkl Mno</td>
</tr>
<tr>
<td>CROSS</td>
<td>Cross</td>
<td>abc,def!ghi'jKL MNO</td>
<td>Abc,Def!Ghi'Jkl Mno</td>
</tr>
</tbody>
</table>

INITCAP capitalizes the first letter of each word.

\[ \text{INITCAP} (\text{NewName}) \]
For the string abc,def!ghi'jKL MNO, the result is Abc,Def!Ghi'Jkl Mno.
For MCKNIGHT, the result is Mcknight.

**LAST_NONBLANK: Retrieving the Last Field Value That is Neither Blank nor Missing**

LAST_NONBLANK retrieves the last field value that is neither blank nor missing. If all previous values are either blank or missing, LAST_NONBLANK returns a missing value.

**Syntax:**

How to Return the Last Value That is Neither Blank nor Missing

LAST_NONBLANK(field)

where:

field

Is the field name whose last non-blank value is to be retrieved. If the current value is not blank or missing, the current value is returned.

**Note:** LAST_NONBLANK cannot be used in a compound expression, for example, as part of an IF condition.

**Example:**

Retrieving the Last Non-Blank Value

Consider the following delimited file named input1.csv that has two fields named FIELD_1 and FIELD_2.

```
, A, , B, C
```

The input1 Master File follows.

```
FILENAME=INPUT1, SUFFIX=DFIX ,
DATASET=baseapp/input1.csv (LRECL 15 RECFM V, BV_NAMESPACE=OFF, $
SEGMENT=INPUT1, SEGTYPE=S0, $
  FIELDNAME=FIELD_1, ALIAS=E01, USAGE=A1V, ACTUAL=A1V,
        MISSING=ON, $
  FIELDNAME=FIELD_2, ALIAS=E02, USAGE=A1V, ACTUAL=A1V,
        MISSING=ON, $
```
The input1 Access File follows.

```access
SEGNAME=INPUT1,
  DELIMITER=',',
  HEADER=NO,
  PRESERVESPACE=NO,
  CDN=COMMAS_DOT,
  CONNECTION=<local>, $
```

The following request displays the FIELD_1 values and computes the last non-blank value for each FIELD_1 value.

```access
TABLE FILE baseapp/INPUT1
PRINT FIELD_1 AS Input
COMPUTE
  Last_NonBlank/A1 MISSING ON = LAST_NONBLANK(FIELD_1);
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image.

```
<table>
<thead>
<tr>
<th>Input</th>
<th>Last NonBlank</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>.</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>
```

**LOWER: Returning a String With All Letters Lowercase**

The LOWER function takes a source string and returns a string of the same data type with all letters translated to lowercase.
**Syntax:**

How to Return a String With All Letters Lowercase

```sql
LOWER(string)
```

where:

- `string` Alphanumeric

  Is the string to convert to lowercase.

The returned string is the same data type and length as the source string.

**Example:** Converting a String to Lowercase

In the following request against the EMPLOYEE data source, LOWER converts the LAST_NAME field to lowercase and stores the result in LOWER_NAME:

```sql
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND COMPUTE
LOWER_NAME/A15 = LOWER(LAST_NAME);
ON TABLE SET PAGE NOPAGE
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>LOWER_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEVENS</td>
<td>stevens</td>
</tr>
<tr>
<td>SMITH</td>
<td>smith</td>
</tr>
<tr>
<td>JONES</td>
<td>jones</td>
</tr>
<tr>
<td>SMITH</td>
<td>smith</td>
</tr>
<tr>
<td>BANNING</td>
<td>banning</td>
</tr>
<tr>
<td>IRVING</td>
<td>irving</td>
</tr>
<tr>
<td>ROMANS</td>
<td>romans</td>
</tr>
<tr>
<td>MCCOY</td>
<td>mccoy</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>blackwood</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>mcknight</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>greenspan</td>
</tr>
<tr>
<td>CROSS</td>
<td>cross</td>
</tr>
</tbody>
</table>

LOWER converts LAST_NAME to lowercase.

`LOWER(LAST_NAME)`

For STEVENS, the result is stevens.
LPAD: Left-Padding a Character String

LPAD uses a specified character and output length to return a character string padded on the left with that character.

*Syntax:*  
How to Pad a Character String on the Left

\[
\text{LPAD}(\text{string}, \text{out\_length}, \text{pad\_character})
\]

where:

- \text{string}  
  Fixed length alphanumeric  
  Is a string to pad on the left side.

- \text{out\_length}  
  Integer  
  Is the length of the output string after padding.

- \text{pad\_character}  
  Fixed length alphanumeric  
  Is a single character to use for padding.

*Example:*  
Left-Padding a String

In the following request against the WF_RETAIL data source, LPAD left-pads the PRODUCT_CATEGORY column with @ symbols:

```
DEFINE FILE WF_RETAILLITE
LPAD1/A25 = LPAD(PRODUCT_CATEGORY, 25, '@');
DIG1/A4 = DIGITS(ID_PRODUCT, 4);
END

TABLE FILE WF_RETAILLITE
SUM DIG1 LPAD1
BY PRODUCT_CATEGORY
ON TABLE SET PAGE NOPAGE
ON TABLE SET STYLE *
TYPE=DATA, FONT=COURIER, SIZE=11, COLOR=BLUE,$
END
```
The output is:

<table>
<thead>
<tr>
<th>Product Category</th>
<th>DIG1</th>
<th>LPAD1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessories</td>
<td>5005</td>
<td>@@@@@@@@@@@@@@@@@@@Accessories</td>
</tr>
<tr>
<td>Camcorder</td>
<td>3006</td>
<td>@@@@@@@@@@@@@@@@@@@Camcorder</td>
</tr>
<tr>
<td>Computers</td>
<td>6016</td>
<td>@@@@@@@@@@@@@@@@@@@Computers</td>
</tr>
<tr>
<td>Media Player</td>
<td>1003</td>
<td>@@@@@@@@@@@@@@@@@@@Media Player</td>
</tr>
<tr>
<td>Stereo Systems</td>
<td>2155</td>
<td>@@@@@@@@@@@@@@@@@@@Stereo Systems</td>
</tr>
<tr>
<td>Televisions</td>
<td>4018</td>
<td>@@@@@@@@@@@@@@@@@@@Televisions</td>
</tr>
<tr>
<td>Video Production</td>
<td>7005</td>
<td>@@@@@@@@@@@@@@@@@@@Video Production</td>
</tr>
</tbody>
</table>

LPAD left-pads the PRODUCT_CATEGORY column with @ symbols:

```
LPAD(PRODUCT_CATEGORY, 25, '@')
```

For Stereo Systems, the output is @@@@@@@@@@@@@@@@@Stereo Systems.

**Reference:** Usage Notes for LPAD

- To use the single quotation mark (') as the padding character, you must double it and enclose the two single quotation marks within single quotation marks (LPAD(COUNTRY, 20, '''')). You can use an amper variable in quotation marks for this parameter, but you cannot use a field, virtual or real.

- Input can be fixed or variable length alphanumeric.

- Output, when optimized to SQL, will always be data type VARCHAR.

- If the output is specified as shorter than the original input, the original data will be truncated, leaving only the padding characters. The output length can be specified as a positive integer or an unquoted &variable (indicating a numeric).

**LTRIM: Removing Blanks From the Left End of a String**

The LTRIM function removes all blanks from the left end of a string.
Syntax: How to Remove Blanks From the Left End of a String

LTRIM(string)

where:

string

   Alphanumeric
   
   Is the string to trim on the left.

The data type of the returned string is AnV, with the same maximum length as the source string.

Example: Removing Blanks From the Left End of a String

In the following request against the MOVIES data source, the DIRECTOR field is right-justified and stored in the RDIRECTOR virtual field. Then LTRIM removes leading blanks from the RDIRECTOR field:

DEFINE FILE MOVIES
RDIRECTOR/A17 = RJUST(17, DIRECTOR, 'A17');
END
TABLE FILE MOVIES
PRINT RDIRECTOR AND
COMPUTE
TRIMDIR/A17 = LTRIM(RDIRECTOR);
WHERE DIRECTOR CONTAINS 'BR'
ON TABLE SET PAGE NOPAGE
END

The output is:

<table>
<thead>
<tr>
<th>RDIRECTOR</th>
<th>TRIMDIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABRAHAMS J.</td>
<td>ABRAHAMS J.</td>
</tr>
<tr>
<td>BROOKS R.</td>
<td>BROOKS R.</td>
</tr>
<tr>
<td>BROOKS J.L.</td>
<td>BROOKS J.L.</td>
</tr>
</tbody>
</table>

RDIRECTOR has the director name right justified. LTRIM removes the leading blanks.

LTRIM(RDIRECTOR)

For BROOKS R. the result is BROOKS R.
PATTERNS: Returning a Pattern That Represents the Structure of the Input String

PATTERNS returns a string that represents the structure of the input argument. The returned pattern includes the following characters:

- **A** is returned for any position in the input string that has an uppercase letter.
- **a** is returned for any position in the input string that has a lowercase letter.
- **9** is returned for any position in the input string that has a digit.

Note that special characters (for example, +/=/%) are returned exactly as they were in the input string.

The output is returned as variable length alphanumeric.

**Syntax:**

How to Return a String That Represents the Pattern Profile of the Input Argument

```
PATTERNS(string)
```

where:

- **string**
  - Alphanumeric
  - Is a string whose pattern will be returned.

**Example:**

Returning a Pattern Representing an Input String

The following request returns patterns that represent customer addresses.

```
DEFINE FILE WF_RETAIL_LITE
Address_Pattern/A40V = PATTERNS(ADDRESS_LINE_1);
END

TABLE FILE WF_RETAIL_LITE
PRINT FST.ADDRESS_LINE_1 OVER Address_Pattern
BY ADDRESS_LINE_1 NOPRINT SKIP-LINE
WHERE COUNTRY_NAME EQ 'United States'
WHERE CITY_NAME EQ 'Houston' OR 'Indianapolis' OR 'Chapel Hill' OR 'Bronx'
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```
The partial output is shown in the following image. Note that the special characters (#,.) in an address are represented in the pattern as is.

<table>
<thead>
<tr>
<th>FST Customer Address Line 1</th>
<th>1010 Milam St # Ifp-2352</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address_Pattern</td>
<td>9999 Aaaaa Aa # Aaa-9999</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FST Customer Address Line 1</th>
<th>10700 Richmond Ave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address_Pattern</td>
<td>99999 Aaaaaaaaa Aaa</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FST Customer Address Line 1</th>
<th>10777 North Fwy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address_Pattern</td>
<td>999999 Aaaaa Aaa</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FST Customer Address Line 1</th>
<th>11 E Greenway Plz Ste 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address_Pattern</td>
<td>99 A Aaaaaaaaa Aaa Aaa 999</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FST Customer Address Line 1</th>
<th>111 Monument Cir</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address_Pattern</td>
<td>999 Aaaaaaaaa Aaa</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FST Customer Address Line 1</th>
<th>111 Monument Circle - Ste 2100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address_Pattern</td>
<td>999 Aaaaaaaaa Aaaaaa - Aaa 9999</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FST Customer Address Line 1</th>
<th>1205 Dart St, Rm 219</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address_Pattern</td>
<td>99999 Aaaa Aa, Aa 999</td>
</tr>
</tbody>
</table>

PATTERNS returns the pattern representing the field ADDRESS_LINE_1.

PATTERNS (ADDRESS_LINE_1)

For 1010 Milam St # Ifp-2352

The result is 9999 Aaaaa Aa # Aaa-9999.

**POSITION: Returning the First Position of a Substring in a Source String**

The POSITION function returns the first position (in characters) of a substring in a source string.
**Syntax:** How to Return the First Position of a Substring in a Source String

```plaintext
POSITION(pattern, string)
```

where:

- **pattern**
  - Alphanumeric
  - Is the substring whose position you want to locate. The string can be as short as a single character, including a single blank.

- **string**
  - Alphanumeric
  - Is the string in which to find the pattern.

The data type of the returned value is Integer.

**Example:** Returning the First Position of a Substring

In the following request against the EMPLOYEE data source, POSITION determines the position of the first capital letter I in LAST_NAME and stores the result in I_IN_NAME:

```plaintext
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND COMPUTE
I_IN_NAME/I2 = POSITION('I', LAST_NAME);
ON TABLE SET PAGE NOPAGE
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>I_IN_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEVENS</td>
<td>0</td>
</tr>
<tr>
<td>SMITH</td>
<td>3</td>
</tr>
<tr>
<td>JONES</td>
<td>0</td>
</tr>
<tr>
<td>SMITH</td>
<td>3</td>
</tr>
<tr>
<td>BANNING</td>
<td>5</td>
</tr>
<tr>
<td>IRVING</td>
<td>1</td>
</tr>
<tr>
<td>ROMANS</td>
<td>0</td>
</tr>
<tr>
<td>MCCOY</td>
<td>0</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>0</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>5</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>0</td>
</tr>
<tr>
<td>CROSS</td>
<td>0</td>
</tr>
</tbody>
</table>

POSITION determines the position of the first capital letter I in LAST_NAME.

```plaintext
POSITION('I', LAST_NAME)
```
For STEVENS, the result is 0.
For SMITH, the result is 3.

**REGEX: Matching a String to a Regular Expression**

The REGEX function matches a string to a regular expression and returns true (1) if it matches and false (0) if it does not match.

A regular expression is a sequence of special characters and literal characters that you can combine to form a search pattern.

Many references for regular expressions exist on the web.

For a basic summary, see the section *Summary of Regular Expressions* in Chapter 2, *Security*, of the *Server Administration* manual.

**Syntax:** How to Match a String to a Regular Expression

REGEX(string, regular_expression)

where:

**string**

Alphanumeric

Is the character string to match.

**regular_expression**

Alphanumeric

Is a regular expression enclosed in single quotation marks constructed using literals and metacharacters. The following metacharacters are supported:

- . represents any single character
- * represents zero or more occurrences
- + represents one or more occurrences
- ? represents zero or one occurrence
- ^ represents beginning of line
- $ represents end of line
- [] represents any one character in the set listed within the brackets
- [^] represents any one character not in the set listed within the brackets
For example, the regular expression '^[Ste\(v|ph\)en]$' matches values starting with Ste followed by either ph or v, and ending with en.

**Note:** The output value is numeric.

### Example: Matching a String Against a Regular Expression

The following request matches the FIRSTNAME field against the regular expression '^[Sara(h?)]$', which matches Sara or Sarah:

```
TABLE FILE WF_RETAIL_LITE
PRINT FIRSTNAME AND COMPUTE
REG1/I1=REGEX(FIRSTNAME,'^[Sara(h?)]$') ;
BY LASTNAME/A10
WHERE LASTNAME EQ 'Allen'
END
```

The output is

<table>
<thead>
<tr>
<th>LASTNAME</th>
<th>Name</th>
<th>REG1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allen</td>
<td>Penny</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Rosemary</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Amber</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Julie</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Sarah</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Leo</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Margret</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Donna</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Damian</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Alexander</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Diego</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Amber</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Susan</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Amber</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Sara</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Sara</td>
<td>1</td>
</tr>
</tbody>
</table>

REGEX matches the FIRSTNAME field against the regular expression '^[Sara(h?)]$', which matches Sara or Sarah:

```
REGEX(FIRSTNAME,'^[Sara(h?)]$')
```

For Sara, the result is 1.
For Amber, the result is 0.

**REPLACE: Replacing a String**

REPLACE replaces all instances of a search string in an input string with the given replacement string. The output is always variable length alphanumeric with a length determined by the input parameters.

**Syntax:** How to Replace all Instances of a String

```sql
REPLACE(input_string, search_string, replacement)
```

where:

`input_string`
Alphanumeric or text (An, AnV, TX)

Is the input string.

`search_string`
Alphanumeric or text (An, AnV, TX)

Is the string to search for within the input string.

`replacement`
Alphanumeric or text (An, AnV, TX)

Is the replacement string to be substituted for the search string. It can be a null string ("").

**Example:** Replacing a String

REPLACE replaces the string ‘South’ in the Country Name with the string ‘S.’

```sql
SET TRACEUSER = ON
SET TRACEON = STMTRACE//CLIENT
SET TRACESTAMP=OFF
DEFINE FILE WF_RETAIL_LITE
NEWNAME/A20 = REPLACE(COUNTRY_NAME, 'SOUTH', 'S.');
END
TABLE FILE WF_RETAIL_LITE
SUM COUNTRY_NAME
BY NEWNAME AS 'New,Name'
WHERE COUNTRY_NAME LIKE 'S%'
ON TABLE SET PAGE NOLEAD
END

REPLACE(COUNTRY_NAME, 'SOUTH', 'S.');
```

For South Africa, the result is S. Africa.

The generated SQL passes the REPLACE function to the DBMS REPLACE function.
SELECT
REPLACE(T3."COUNTRY_NAME", 'SOUTH', 'S.'),
MAX(T3."COUNTRY_NAME")
FROM
wrd_wf_retail_geography T3
WHERE
(T3."COUNTRY_NAME" LIKE 'S%')
GROUP BY
REPLACE(T3."COUNTRY_NAME", 'SOUTH', 'S.')
ORDER BY
REPLACE(T3."COUNTRY_NAME", 'SOUTH', 'S.');

The output is shown in the following image.

<table>
<thead>
<tr>
<th>New Name</th>
<th>Customer Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>S. Africa</td>
<td>South Africa</td>
</tr>
<tr>
<td>S. Korea</td>
<td>South Korea</td>
</tr>
<tr>
<td>Singapore</td>
<td>Singapore</td>
</tr>
<tr>
<td>Spain</td>
<td>Spain</td>
</tr>
<tr>
<td>Sweden</td>
<td>Sweden</td>
</tr>
<tr>
<td>Switzerland</td>
<td>Switzerland</td>
</tr>
</tbody>
</table>

**Example:** Replacing All Instances of a String

In the following request, the virtual field DAYNAME1 is the string DAY1 with all instances of the string 'DAY' replaced with the string 'day'. The virtual field DAYNAME2 has all instances of the string 'DAY' removed.

DEFINE FILE WF_RETAILLITE
DAY1/A30 = 'SUNDAY MONDAY TUESDAY';
DAYNAME1/A30 = REPLACE(DAY1, 'DAY', 'day');
DAYNAME2/A30 = REPLACE(DAY1, 'DAY', '');
END
TABLE FILE WF_RETAILLITE
PRINT DAY1 OVER
DAYNAME1 OVER
DAYNAME2
WHERE EMPLOYEE_NUMBER EQ 'AH118'
ON TABLE SET PAGE NOPAGE
END
The output is:

<table>
<thead>
<tr>
<th>DAY1</th>
<th>SUNDAY</th>
<th>MONDAY</th>
<th>TUESDAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAYNAME1</td>
<td>SUNday</td>
<td>MONday</td>
<td>TUESday</td>
</tr>
<tr>
<td>DAYNAME2</td>
<td>SUN</td>
<td>MON</td>
<td>TUES</td>
</tr>
</tbody>
</table>

REPLACE removes the characters 'DAY' from the string DAY1:

REPLACE(DAY1, 'DAY', '' )

For 'SUNDAY MONDAY TUESDAY', the result is 'SUN MON TUES'.

**RPAD: Right-Padding a Character String**

RPAD uses a specified character and output length to return a character string padded on the right with that character.

**Syntax:**

How to Pad a Character String on the Right

RPAD(string, out_length, pad_character)

where:

- **string**
  - Alphanumeric
  - Is a string to pad on the right side.

- **out_length**
  - Integer
  - Is the length of the output string after padding.

- **pad_character**
  - Alphanumeric
  - Is a single character to use for padding.
**Example:** Right-Padding a String

In the following request against the WF_RETAIL data source, RPAD right-pads the PRODUCT CATEGORY column with @ symbols:

```plaintext
DEFINE FILE WF_RETAILLITE
RPAD1/A25 = RPAD(PRODUCT CATEGORY, 25, '@');
DIG1/A4 = DIGITS(ID PRODUCT, 4);
END
TABLE FILE WF_RETAILLITE
SUM DIG1 RPAD1
BY PRODUCT CATEGORY
ON TABLE SET PAGE NOPAGE
ON TABLE SET STYLE *
TYPE=DATA, FONT=COURIER, SIZE=11, COLOR=BLUE,
END
```

The output is:

<table>
<thead>
<tr>
<th>Product Category</th>
<th>DIG1</th>
<th>RPAD1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessories</td>
<td>5005</td>
<td>Accessories@@@@@@@@@@@@@@@@@@@@@@</td>
</tr>
<tr>
<td>Camcorder</td>
<td>3006</td>
<td>Camcorder@@@@@@@@@@@@@@@@@@@@@@</td>
</tr>
<tr>
<td>Computers</td>
<td>6016</td>
<td>Computers@@@@@@@@@@@@@@@@@@@@@@</td>
</tr>
<tr>
<td>Media Player</td>
<td>1003</td>
<td>Media Player@@@@@@@@@@@@@@@@@@@@@@</td>
</tr>
<tr>
<td>Stereo Systems</td>
<td>2155</td>
<td>Stereo Systems@@@@@@@@@@@@@@@@@@@@@@</td>
</tr>
<tr>
<td>Televisions</td>
<td>4018</td>
<td>Televisions@@@@@@@@@@@@@@@@@@@@@@</td>
</tr>
<tr>
<td>Video Production</td>
<td>7005</td>
<td>Video Production@@@@@@@@@@@@@@@@@@@@@@</td>
</tr>
</tbody>
</table>

RPAD right-pads the PRODUCT CATEGORY column with @ symbols:

`RPAD(PRODUCT CATEGORY, 25, '@')`

For **Stereo Systems**, the output is **Stereo Systems@@@@@@@@@@@@@@@@@@**.

**Reference:** Usage Notes for RPAD

- The input string can be data type AnV, VARCHAR, TX, and An.
- Output can only be AnV or An.
When working with relational VARCHAR columns, there is no need to trim trailing spaces from the field if they are not desired. However, with An and AnV fields derived from An fields, the trailing spaces are part of the data and will be included in the output, with the padding being placed to the right of these positions. You can use TRIM or TRIMV to remove these trailing spaces prior to applying the RPAD function.

### RTRIM: Removing Blanks From the Right End of a String

The RTRIM function removes all blanks from the right end of a string.

#### Syntax: How to Remove Blanks From the Right End of a String

```
RTRIM(string)
```

where:

- `string`  
  - Alphanumeric  
  - Is the string to trim on the right.

The data type of the returned string is AnV, with the same maximum length as the source string.

#### Example: Removing Blanks From the Right End of a String

The following request against the MOVIES data source creates the field DIRSLASH, that contains a slash at the end of the DIRECTOR field. Then it creates the TRIMDIR field, which trims the trailing blanks from the DIRECTOR field and places a slash at the end of that field:

```
TABLE FILE MOVIES
PRINT DIRECTOR NOPRINT AND
COMPUTE
DIRSLASH/A18 = DIRECTOR|'/';
TRIMDIR/A17V = RTRIM(DIRECTOR)|'/';
WHERE DIRECTOR CONTAINS 'BR'
ON TABLE SET PAGE NOPAGE
END
```

On the output, the slashes show that the trailing blanks in the DIRECTOR field were removed in the TRIMDIR field:

<table>
<thead>
<tr>
<th>DIRSLASH</th>
<th>TRIMDIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABRAHAMS J.</td>
<td>/ ABRAHAMS J./</td>
</tr>
<tr>
<td>BROOKS R.</td>
<td>/ BROOKS R./</td>
</tr>
<tr>
<td>BROOKS J.L.</td>
<td>/ BROOKS J.L./</td>
</tr>
</tbody>
</table>
RTRIM removes trailing blanks from DIRECTOR.

RTRIM(DIRECTOR)

For BROOKS R., the result is BROOKS R.

**SPLIT: Extracting an Element From a String**

The SPLIT function returns a specific type of element from a string. The output is returned as variable length alphanumeric.

**Syntax:** How to Extract an Element From a String

SPLIT(element, string)

where:

*element*

Can be one of the following keywords:

- **EMAIL.DOMAIN.** Is the domain name portion of an email address in the string.
- **EMAIL.USERID.** Is the user ID portion of an email address in the string.
- **URL.PROTOCOL.** Is the URL protocol in the string.
- **URL.HOST.** Is the host name of the URL in the string.
- **URL.PORT.** Is the port number of the URL in the string.
- **URL.PATH.** Is the URL path in the string.
- **NAME.FIRST.** Is the first token (group of characters) in the string. Tokens are delimited by blanks.
- **NAME.LAST.** Is the last token (group of characters) in the string. Tokens are delimited by blanks.

*string*

Alphanumeric

Is the string from which the element will be extracted.
Example: Extracting an Element From a String

The following request defines strings and extracts elements from them.

```define file wf_retail_lite
string1/a50 with country_name= 'http://www.informationbuilders.com';
string2/a20 = 'user1@ibi.com';
string3/a20 = 'Louisa May Alcott';
protocol/a20 = split(url_protocol, string1);
path/a50 = split(url_path, string1);
domain/a20 = split(email_domain, string2);
user/a20 = split(email_userid, string2);
first/a10 = split(name_first, string3);
last/a10 = split(name_last, string3);
end
```

TABLE FILE WF_RETAIL_LITE
SUM protocol path user domain first last
ON TABLE SET PAGE NOLEAD
END

The output is shown in the following image.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Path</th>
<th>User</th>
<th>Domain</th>
<th>First</th>
<th>Last</th>
</tr>
</thead>
<tbody>
<tr>
<td>http</td>
<td><a href="http://www.informationbuilders.com">http://www.informationbuilders.com</a></td>
<td><a href="mailto:user1@ibi.com">user1@ibi.com</a></td>
<td>Louisa May Alcott</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SPLIT extracts the URL protocol from the string STRING1.

`split(url_protocol, string1)`

For 'http://www.informationbuilders.com', the result is http.

SUBSTRING: Extracting a Substring From a Source String

The SUBSTRING function extracts a substring from a source string. If the ending position you specify for the substring is past the end of the source string, the position of the last character of the source string becomes the ending position of the substring.

Syntax: How to Extract a Substring From a Source String

`substring(string, position, length)`

where:

- `string` is the string from which to extract the substring. It can be a field, a literal in single quotation marks ('), or a variable.
**position**
Positive Integer

Is the starting position of the substring in string.

**length**
Integer

Is the limit for the length of the substring. The ending position of the substring is calculated as position + length - 1. If the calculated position beyond the end of the source string, the position of the last character of string becomes the ending position.

The data type of the returned substring is AnV.

**Example:**  **Extracting a Substring From a Source String**

In the following request, POSITION determines the position of the first letter I in LAST_NAME and stores the result in I_IN_NAME. SUBSTRING, then extracts three characters beginning with the letter I from LAST_NAME and stores the results in I_SUBSTR.

```
TABLE FILE EMPLOYEE
PRINT
COMPUTE I_IN_NAME/I2 = POSITION('I', LAST_NAME); AND
COMPUTE I_SUBSTR/A3 = SUBSTRING(LAST_NAME, I_IN_NAME, I_IN_NAME+2);
BY LAST_NAME
ON TABLE SET PAGE NOPAGE
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>I_IN_NAME</th>
<th>I_SUBSTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>5</td>
<td>ING</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>0</td>
<td>BL</td>
</tr>
<tr>
<td>CROSS</td>
<td>0</td>
<td>CR</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>0</td>
<td>GR</td>
</tr>
<tr>
<td>IRVING</td>
<td>1</td>
<td>IRV</td>
</tr>
<tr>
<td>JONES</td>
<td>0</td>
<td>JO</td>
</tr>
<tr>
<td>MCCOY</td>
<td>0</td>
<td>MC</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>5</td>
<td>IGH</td>
</tr>
<tr>
<td>ROMANS</td>
<td>0</td>
<td>RO</td>
</tr>
<tr>
<td>SMITH</td>
<td>3</td>
<td>ITH</td>
</tr>
<tr>
<td>STEVENS</td>
<td>0</td>
<td>ST</td>
</tr>
</tbody>
</table>

POSITION determines the position of the first letter I in LAST_NAME.

**SUBSTRING(LAST_NAME, I_IN_NAME, I_IN_NAME+2)**
For BANNING, the result is 5.

**TOKEN: Extracting a Token From a String**

The token function extracts a token (substring) from a string of characters. The tokens are separated by a delimiter consisting of one or more characters and specified by a token number reflecting the position of the token in the string.

**Syntax:** How to Extract a Token From a String

\[
\text{TOKEN}(\text{string}, \text{delimiter}, \text{number})
\]

where:

- **string**
  - Fixed length alphanumeric
  - Is the character string from which to extract the token.

- **delimiter**
  - Fixed length alphanumeric
  - Is a delimiter consisting of one or more characters.
  - TOKEN can be optimized if the delimiter consists of a single character.

- **number**
  - Integer
  - Is the token number to extract.

**Example:** Extracting a Token From a String

TOKEN extracts the second token from the PRODUCT_SUBCATEG column, where the delimiter is the letter P:

```
DEFINE FILE WF_RETAIL_LITE
TOK1/A20 = TOKEN(PRODUCT_SUBCATEG,'P',2);
END
TABLE FILE WF_RETAIL_LITE
SUM TOK1 AS Token
BY PRODUCT_SUBCATEG
ON TABLE SET PAGE NOPAGE
END
```
The output is:

<table>
<thead>
<tr>
<th>Product Subcategory</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blu Ray</td>
<td></td>
</tr>
<tr>
<td>Boom Box</td>
<td></td>
</tr>
<tr>
<td>CRT TV</td>
<td></td>
</tr>
<tr>
<td>Charger</td>
<td></td>
</tr>
<tr>
<td>DVD Players</td>
<td>layers</td>
</tr>
<tr>
<td>DVD Players - Portable</td>
<td>layers -</td>
</tr>
<tr>
<td>Flat Panel TV</td>
<td>anel TV</td>
</tr>
<tr>
<td>Handheld</td>
<td></td>
</tr>
<tr>
<td>Headphones</td>
<td>phones</td>
</tr>
<tr>
<td>Home Theater Systems</td>
<td></td>
</tr>
<tr>
<td>Portable TV</td>
<td>portable TV</td>
</tr>
<tr>
<td>Professional</td>
<td>rofessional</td>
</tr>
<tr>
<td>Receivers</td>
<td></td>
</tr>
<tr>
<td>Smartphone</td>
<td>hone</td>
</tr>
<tr>
<td>Speaker Kits</td>
<td>speaker Kits</td>
</tr>
<tr>
<td>Standard</td>
<td></td>
</tr>
<tr>
<td>Streaming</td>
<td></td>
</tr>
<tr>
<td>Tablet</td>
<td></td>
</tr>
<tr>
<td>Universal Remote Controls</td>
<td></td>
</tr>
<tr>
<td>Video Editing</td>
<td></td>
</tr>
<tr>
<td>iPod Docking Station</td>
<td>Docking Station</td>
</tr>
</tbody>
</table>

TOKEN extracts the second token from the PRODUCT_SUBCATEG column, where the delimiter is a blank:

\[
\text{TOKEN(\text{PRODUCT\_SUBCATEG},' ',2)}
\]

For iPod Docking Station, the result is Docking.
**TRIM_: Removing a Leading Character, Trailing Character, or Both From a String**

The TRIM_ function removes all occurrences of a single character from either the beginning or end of a string, or both.

**Note:**

- Leading and trailing blanks count as characters. If the character you want to remove is preceded (for leading) or followed (for trailing) by a blank, the character will not be removed. Alphanumeric fields that are longer than the number of characters stored within them are padded with trailing blanks.

- The function will be optimized when run against a relational DBMS that supports trimming the character and location specified.

**Syntax:** How to Remove a Leading Character, Trailing Character, or Both From a String

```plaintext
TRIM_(where, pattern, string)
```

where:

- **where**
  - **Keyword**
  - Defines where to trim the source string. Valid values are:
    - **LEADING**, which removes leading occurrences.
    - **TRAILING**, which removes trailing occurrences.
    - **BOTH**, which removes leading and trailing occurrences.

- **pattern**
  - **Alphanumeric**
  - Is a single character, enclosed in single quotation marks ("'"), whose occurrences are to be removed from string. For example, the character can be a single blank ("' '").

- **string**
  - **Alphanumeric**
  - Is the string to be trimmed.

The data type of the returned string is AnV.
**Example:** Trimming a Character From a String

In the following request, TRIM_ removes leading occurrences of the character ‘B’ from the DIRECTOR field:

```plaintext
TABLE FILE MOVIES
PRINT DIRECTOR AND
COMPUTE
TRIMDIR/A17 = TRIM_(LEADING, 'B', DIRECTOR);
WHERE DIRECTOR CONTAINS 'BR'
ON TABLE SET PAGE NOPAGE
END
```

The output is:

```
DIRECTOR   TRIMDIR
--------   -------
ABRAHAMS J. ABRAHAMS J.
BROOKS R.   ROOKS R.
BROOKS J.L. ROOKS J.L.
```

TRIM_ removes leading occurrences of the character ‘B’ from DIRECTOR.

TRIM_(LEADING, 'B', DIRECTOR)

For BROOKS R., the result is ROOKS R.

**Example:** Trimming With Trailing Blanks

The following request trims a trailing period (.) from the director name. The field DIRECTOR has format A17, so there are trailing blanks in most of the instances of the field. To create a field (DIRECTORV) without trailing blanks, SQUEEZ converts the trailing blanks in DIRECTOR to a single blank, then TRIMV removes the remaining trailing blank and stores it with format A17V, so the length of the actual characters is known. Then TRIM_ is called against DIRECTOR and DIRECTORV, creating the fields TRIMDIR (trimmed DIRECTOR) and TRIMDIRV (trimmed DIRECTORV):

```plaintext
DEFINE FILE MOVIES
DIRECTORV/A17V = TRIMV('T', SQUEEZ(17, DIRECTOR, 'A17V'), 17, ' ', 1, DIRECTORV);
TRIMDIR/A17 = TRIM_(TRAILING, '.', DIRECTOR);
TRIMDIRV/A17V = TRIM_(TRAILING, '.', DIRECTORV);
END
TABLE FILE MOVIES
PRINT DIRECTOR TRIMDIR DIRECTORV TRIMDIRV
ON TABLE SET PAGE NOPAGE
END
```
The partial output shows that the trimmed DIRECTOR field still has the trailing periods because the period is not the last character in the field. In the trimmed DIRECTORV field, the trailing periods have been removed:

<table>
<thead>
<tr>
<th>DIRECTOR</th>
<th>TRIMDIR</th>
<th>DIRECTORV</th>
<th>TRIMDIRV</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPIELBERG S.</td>
<td>SPIELBERG S.</td>
<td>SPIELBERG S.</td>
<td>SPIELBERG S</td>
</tr>
<tr>
<td>KAZAN E.</td>
<td>KAZAN E.</td>
<td>KAZAN E.</td>
<td>KAZAN E</td>
</tr>
<tr>
<td>WELLES O.</td>
<td>WELLES O.</td>
<td>WELLES O.</td>
<td>WELLES O</td>
</tr>
<tr>
<td>LUMET S.</td>
<td>LUMET S.</td>
<td>LUMET S.</td>
<td>LUMET S</td>
</tr>
</tbody>
</table>

**UPPER: Returning a String With All Letters Uppercase**

The UPPER function takes a source string and returns a string of the same data type with all letters translated to uppercase.

**Syntax:**

How to Return a String With All Letters Uppercase

\[ \text{UPPER(string)} \]

where:

\[ \text{string} \]

- Alphanumeric

  Is the string to convert to uppercase.

The returned string is the same data type and length as the source string.

**Example:**

Converting Letters to Uppercase

In the following request, LCWORD converts LAST_NAME to mixed case. Then UPPER converts the LAST_NAME_MIXED field to uppercase:

```
DEFINE FILE EMPLOYEE
LAST_NAME_MIXED/A15=LCWORD(15, LAST_NAME, 'A15');
LAST_NAME_UPPER/A15=UPPER(LAST_NAME_MIXED) ;
END
TABLE FILE EMPLOYEE
PRINT LAST_NAME_UPPER AND FIRST_NAME
BY LAST_NAME_MIXED
WHERE CURR_JOBCODE EQ 'B02' OR 'A17' OR 'B04';
ON TABLE SET PAGE NOPAGE
END
```
The output is:

<table>
<thead>
<tr>
<th>LAST_NAME_MIXED</th>
<th>LAST_NAME_UPPER</th>
<th>FIRST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banning</td>
<td>BANNING</td>
<td>JOHN</td>
</tr>
<tr>
<td>Blackwood</td>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
</tr>
<tr>
<td>Cross</td>
<td>CROSS</td>
<td>BARBARA</td>
</tr>
<tr>
<td>McCoy</td>
<td>MCCOY</td>
<td>JOHN</td>
</tr>
<tr>
<td>Mcknight</td>
<td>MCKNIGHT</td>
<td>ROGER</td>
</tr>
<tr>
<td>Romans</td>
<td>ROMANS</td>
<td>ANTHONY</td>
</tr>
</tbody>
</table>

LAST_NAME_MIXED has the last name in mixed case. UPPER converts LAST_NAME_MIXED to uppercase.

UPPER(LAST_NAME_MIXED)

For Banning, the result is BANNING.
Chapter 4

Character Functions

Character functions manipulate alphanumeric fields and character strings.

In this chapter:

- ARGLEN: Measuring the Length of a String
- ASIS: Distinguishing Between Space and Zero
- BITSON: Determining If a Bit Is On or Off
- BITVAL: Evaluating a Bit String as an Integer
- BYTVAL: Translating a Character to Decimal
- CHKFMT: Checking the Format of a String
-CHKNUM: Checking a String for Numeric Format
- CTRAN: Translating One Character to Another
- CTRFLD: Centering a Character String
- EDIT: Extracting or Adding Characters
- GETTOK: Extracting a Substring (Token)
- LCWORD: Converting a String to Mixed-Case
- LCWORD2: Converting a String to Mixed-Case
- LCWORD3: Converting a String to Mixed-Case
- OVRLAY: Overlaying a Character String
- PARAG: Dividing Text Into Smaller Lines
- PATTERN: Generating a Pattern From a String
- POSIT: Finding the Beginning of a Substring
- REVERSE: Reversing the Characters in a String
- RJUST: Right-Justifying a Character String
- SOUNDEX: Comparing Character Strings Phonetically
- SPELLNM: Spelling Out a Dollar Amount
- SQUEEZ: Reducing Multiple Spaces to a Single Space
- STRIP: Removing a Character From a String
- STRREP: Replacing Character Strings
- SUBSTR: Extracting a Substring
- TRIM: Removing Leading and Trailing Occurrences
- UPCASE: Converting Text to Uppercase
- XMLDECOD: Decoding XML-Encoded Characters
ARGLEN: Measuring the Length of a String

Available Languages: reporting, Maintain

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

In Dialogue Manager, you can measure the length of a supplied character string using the .LENGTH suffix.

Syntax:

How to Measure the Length of a Character String

ARGLEN(length, source_string, output)

ARGLEN(inlength, infield, 'outfield')

where:

length

Integer
Is the length of the field containing the character string, or a field that contains the length.

source_string

Alphanumeric
Is the name of the field containing the character string.

output

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

Is the format of the output value enclosed in single quotation marks.
**Example:** Measuring the Length of a Character String

ARGLEN determines the length of the character string in LAST_NAME and stores the result in NAME_LEN:

```
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND COMPUTE
NAME_LEN/I3 = ARGLEN(15, LAST_NAME, NAME_LEN);
WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>NAME_LEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>5</td>
</tr>
<tr>
<td>JONES</td>
<td>5</td>
</tr>
<tr>
<td>MCCOY</td>
<td>5</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>9</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>9</td>
</tr>
<tr>
<td>CROSS</td>
<td>5</td>
</tr>
</tbody>
</table>

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.

```
COMPUTE NAME_LEN/I3 = ARGLEN(15, LAST_NAME, 'I3');
```

**ASIS: Distinguishing Between Space and Zero**

Available Languages: reporting

The ASIS function distinguishes between a space and a zero in Dialogue Manager. It differentiates between a numeric string, a constant or variable defined as a numeric string (number within single quotation marks), and a field defined simply as numeric. ASIS forces a variable to be evaluated as it is entered rather than be converted to a number. It is used in Dialogue Manager equality expressions only.
**Syntax:** How to Distinguish Between a Space and a Zero

\[ \text{ASIS}(\text{argument}) \]

where:

- **argument**
  - Alphanumeric

  Is the value to be evaluated. Supply the actual value, the name of a field that contains the value, or an expression that returns the value. An expression can call a function.

  If you specify an alphanumerical literal, enclose it in single quotation marks. If you specify an expression, use parentheses, as needed, to ensure the correct order of evaluation.

**Example:** Distinguishing Between a Space and a Zero

The first request does not use ASIS. No difference is detected between variables defined as a space and 0.

```as
-SET &VAR1 = ' ';  
-SET &VAR2 = 0;  
-IF &VAR2 EQ &VAR1 GOTO ONE;  
-TYPE VAR1 &VAR1 EQ VAR2 &VAR2 NOT TRUE  
-QUIT  
-ONE  
-TYPE VAR1 &VAR1 EQ VAR2 &VAR2 TRUE
```

The output is:

```
VAR1 EQ VAR2 0 TRUE
```

The next request uses ASIS to distinguish between the two variables.

```as
-SET &VAR1 = ' ';  
-SET &VAR2 = 0;  
-IF &VAR2 EQ \text{ASIS}(&VAR1) GOTO ONE;  
-TYPE VAR1 &VAR1 EQ VAR2 &VAR2 NOT TRUE  
-QUIT  
-ONE  
-TYPE VAR1 &VAR1 EQ VAR2 &VAR2 TRUE
```

The output is:

```
VAR1 EQ VAR2 0 NOT TRUE
```
**Reference: Usage Notes for ASIS**

In general, Dialogue Manager variables are treated as alphanumeric values. However, a Dialogue Manager variable with the value of '.' may be treated as an alphanumeric value ('.') or a number (0) depending on the context used.

- If the Dialogue Manager variable '.' is used in a mathematical expression, its value will be treated as a number. For example, in the following request, &DMVAR1 is used in an arithmetic expression and is evaluated as zero (0).

  ```
  -SET &DMVAR1='.';
  -SET &DMVAR2=10 + &DMVAR1;
  TYPE DMVAR2 = &DMVAR2
  ```

  The output is;

  ```
  DMVAR2 = 10
  ```

- If the Dialogue Manager variable value '.' is used in an IF test and is compared to the values ' ', '0', or '.', the result will be TRUE even if ASIS is used, as shown in the following example. The following IF tests all evaluate to TRUE.

  ```
  -SET &DMVAR1='.';
  -SET &DMVAR2=IF &DMVAR1 EQ ' ' THEN 'TRUE' ELSE 'FALSE';
  -SET &DMVAR3=IF &DMVAR1 EQ '.' THEN 'TRUE' ELSE 'FALSE';
  -SET &DMVAR4=IF &DMVAR1 EQ '0' THEN 'TRUE' ELSE 'FALSE';
  ```

- If the Dialogue Manager variable is used with ASIS, the result of the ASIS function will be always be considered alphanumeric and will distinguish between the space (' '), zero ('0'), or period ('.'), as in the following example. The following IF tests all evaluate to TRUE.

  ```
  -SET &DMVAR2=IF ASIS('.') EQ '.' THEN 'TRUE' ELSE 'FALSE';
  -SET &DMVAR3=IF ASIS(' ') EQ ' ' THEN 'TRUE' ELSE 'FALSE';
  -SET &DMVAR4=IF ASIS('0') EQ '0' THEN 'TRUE' ELSE 'FALSE';
  ```

- Comparing ASIS('0') to ' ' and ASIS(' ') to '0' always evaluates to FALSE.

**BITSON: Determining If a Bit Is On or Off**

Available Languages: reporting, Maintain

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**

\[ \text{BITSON}(\text{bitnumber}, \text{source_string}, \text{output}) \]

\[ \text{BITSON}(\text{bitnumber}, \text{string}, '\text{outfield}') \]

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 to be evaluated, enclosed in single quotation marks, or a field or variable that contains the character string, enclosed in single quotation marks, or a field that contains the character string. The character string is in multiple eight-bit blocks.

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

  Is the format of the output value enclosed in single quotation marks.

**Example:**

**Evaluating a Bit in a Field**

BITSON evaluates the 24th bit of LAST_NAME and stores the result in BIT_24:

\[
\begin{align*}
\text{TABLE FILE EMPLOYEE} \\
\text{PRINT LAST_NAME AND COMPUTE} \\
\text{BIT_24/I1 = BITSON(24, LAST_NAME, BIT_24);} \\
\text{WHERE DEPARTMENT EQ 'MIS';} \\
\text{END}
\end{align*}
\]

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>BIT_24</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>1</td>
</tr>
<tr>
<td>JONES</td>
<td>1</td>
</tr>
<tr>
<td>MCCOY</td>
<td>1</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>1</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>1</td>
</tr>
<tr>
<td>CROSS</td>
<td>0</td>
</tr>
</tbody>
</table>
BITSON(24, LAST_NAME, 'I1')

For SMITH, the result is 1.
For CROSS, the result is 9.

\[ \text{COMPUTE BIT}_24/\text{I1} = \text{BITSON}(24, \text{LAST_NAME}, 'I1'); \]

**BITVAL: Evaluating a Bit String as an Integer**

Available Languages: reporting, Maintain

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.

If the number of bits is:

- Less than 1, the returned value is 0.
- Between 1 and 31 (the recommended range), the returned value is a zero or positive number representing the bits specified, extended with high-order zeroes for a total of 32 bits.
- Exactly 32, the returned value is the positive, zero, or the complement value of negative two, of the specified 32 bits.
- Greater than 32 (33 or more), the returned value is the positive, zero, or the complement value of negative two, of the rightmost 32 bits specified.

**Syntax:**

How to Evaluate a Bit String

\[ \text{BITVAL}(\text{source_string}, \text{startbit}, \text{number}, \text{output}) \]

where:

- **source_string**
  
  Alphanumeric
  
  Is the character string to be evaluated, enclosed in single quotation marks, or a field or variable that contains the character string.
startbit
Integer

Is the number of the first bit in the bit string, counting from the left-most bit in the
character string. If this argument is less than or equal to 0, the function returns a value of
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
Integer

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

**Example:** Evaluating a Bit String

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

```plaintext
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND COMPUTE
STRING_VAL/I5 = BITVAL(LAST_NAME, 12, 9, 'I5');
WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>STRING_VAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>332</td>
</tr>
<tr>
<td>JONES</td>
<td>365</td>
</tr>
<tr>
<td>MCCOY</td>
<td>60</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>316</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>412</td>
</tr>
<tr>
<td>CROSS</td>
<td>413</td>
</tr>
</tbody>
</table>

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

For SMITH, the result is 332.

For JONES, the result is 365.
BYTVAL: Translating a Character to Decimal

Available Languages: reporting, Maintain

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

**Syntax:**

How to Translate a Character

\[
\text{BYTVAL(character, output)}
\]

\[
\text{BYTVAL(character, 'outfield')}
\]

where:

- **character**
  
  Alphanumeric
  
  Is the character to be translated. You can specify a field or variable that contains the character, or the character itself enclosed in single quotation marks. If you supply more than one character, the function evaluates the first.

- **output**
  
  Integer
  
  Is the name of the field that contains the corresponding decimal value, or the format of the output value enclosed in single quotation marks.

  Is the format of the output value enclosed in single quotation marks.

**Example:** Translating the First Character of a Field

BYTVAL translates the first character of LAST_NAME into its ASCII or EBCDIC decimal value and stores the result in LAST_INIT_CODE. Since the input string has more than one character, BYTVAL evaluates the first one.

\[
\text{COMPUTE LAST_INIT_CODE/I3 = BYTVAL(LAST_NAME, 'I3');}
\]
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND
COMPUTE LAST_INIT_CODE/I3 = BYTVAL(LAST_NAME, 'I3');
WHERE DEPARTMENT EQ 'MIS';
END

The output on an ASCII platform is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>LAST_INIT_CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>83</td>
</tr>
<tr>
<td>JONES</td>
<td>74</td>
</tr>
<tr>
<td>MCCOY</td>
<td>77</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>66</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>71</td>
</tr>
<tr>
<td>CROSS</td>
<td>67</td>
</tr>
</tbody>
</table>

The output on an EBCDIC platform is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>LAST_INIT_CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>226</td>
</tr>
<tr>
<td>JONES</td>
<td>209</td>
</tr>
<tr>
<td>MCCOY</td>
<td>212</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>194</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>199</td>
</tr>
<tr>
<td>CROSS</td>
<td>195</td>
</tr>
</tbody>
</table>

BYTVAL translates the first character of LAST_NAME into its ASCII decimal value 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.
Example: Returning the EBCDIC Value With Dialogue Manager

This Dialogue Manager request prompts for a character, then returns the corresponding number. The following reflects the results on the Windows platform.

```
-SET &CODE = BYTVAL(&CHAR, 'I3');
-HTMLFORM BEGIN
  <HTML>
  <BODY>
  THE EQUIVALENT VALUE IS &CODE
  </BODY>
  </HTML>
-HTMLFORM END
```

Assume the value entered for &CHAR is an exclamation point (!). The output is:

THE EQUIVALENT VALUE IS 33

Example: Returning the EBCDIC Value With Dialogue Manager

This Dialogue Manager request prompts for a character, then returns the corresponding number. The following reflects the results on the z/OS platform.

```
PROMPT &CHAR.ENTER THE CHARACTER TO BE DECODED.
-SET &CODE = BYTVAL(&CHAR, 'I3');
-TYPE
-TYPE THE EQUIVALENT VALUE IS &CODE
```

Suppose you want to know the equivalent value of the exclamation point (!). A sample execution is:

ENTER THE CHARACTER TO BE DECODED
!
THE EQUIVALENT VALUE IS 90
>

CHKFMT: Checking the Format of a String

Available Languages: reporting, Maintain

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, by 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 in the string 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)

CHKFMT(numchar, string, 'mask', 'outfield')

where:

numchar
   Integer
   Is the number of characters being compared to the mask.

string
   Alphanumeric
   Is the character string to be checked enclosed in single quotation marks, or a field or variable that contains the character string.

   Is the character string to be checked enclosed in single quotation marks, or a field that contains the character string.

'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.
outputoutfield

Integer

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

Is the format of the output value enclosed in single quotation marks.

Example: Checking the Format of a Field

CHKFMT examines EMP_ID for nine numeric characters starting with 11 and stores the result in CHK_ID:

TABLE FILE EMPLOYEE
PRINT EMP_ID AND LAST_NAME AND
COMPUTE CHK_ID/I3 = CHKFMT(9, EMP_ID, '119999999', CHK_ID);
WHERE DEPARTMENT EQ 'PRODUCTION';
END

The output is:

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>LAST_NAME</th>
<th>CHK_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>071382660</td>
<td>STEVENS</td>
<td>1</td>
</tr>
<tr>
<td>119265415</td>
<td>SMITH</td>
<td>0</td>
</tr>
<tr>
<td>119329144</td>
<td>BANNING</td>
<td>0</td>
</tr>
<tr>
<td>123764317</td>
<td>IRVING</td>
<td>2</td>
</tr>
<tr>
<td>126724188</td>
<td>ROMANS</td>
<td>2</td>
</tr>
<tr>
<td>451123478</td>
<td>MCKNIGHT</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

COMPUTE CHK_ID/I3 = CHKFMT(9, EMP_ID, '119999999', 'I3');
Example: Checking the Format of a Field With MODIFY on z/OS

The following MODIFY procedure adds records of new employees to the EMPLOYEE data source. Each transaction begins as an employee ID that is alphanumeric with the first five characters as digits. The procedure rejects records with other characters in the employee ID.

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID LAST_NAME FIRST_NAME DEPARTMENT
MATCH EMP_ID
  ON MATCH REJECT
  ON NOMATCH COMPUTE
    BAD_CHAR/I3 = CHKFMT(5, EMP_ID, '99999', BAD_CHAR);
  ON NOMATCH VALIDATE
    ID_TEST = IF BAD_CHAR EQ 0 THEN 1 ELSE 0;
    ON INVALID TYPE
      "BAD EMPLOYEE ID: <EMP_ID"
      "INVALID CHARACTER IN POSITION <BAD_CHAR"
  ON NOMATCH INCLUDE
    LOG INVALID MSG OFF

DATA

A sample execution is:

```
>
EMPLOYEEFOCUS   A ON 12/05/96 AT 15.42.03
DATA FOR TRANSACTION    1
EMP_ID      =
111w2
LAST_NAME   =
johnson
FIRST_NAME  =
greg
DEPARTMENT  =
production
BAD EMPLOYEE ID: 111W2
INVALID CHARACTER IN POSITION 4
DATA FOR TRANSACTION    2
EMP_ID      =
end
TRANSACTIONS: TOTAL =     1 ACCEPTED=     0 REJECTED=     1
SEGMENTS: INPUT =     0 UPDATED =     0 DELETED =     0
>
```

The procedure processes as follows:

1. The procedure searches the data source for the ID 111w2. If it does not find this ID, it continues processing the transaction.
2. CHKFMT checks the ID against the mask 99999, which represents five digits.
3. The fourth character in the ID, the letter w, is not a digit. The function returns the value 4 to the BAD_CHAR field.
4. The VALIDATE command tests the BAD_CHAR field. Since BAD_CHAR is not equal to 0, the procedure rejects the transaction and displays a message indicating the position of the invalid character in the ID.

CHKNUM: Checking a String for Numeric Format

The CHKNUM function checks a character string for numeric format. If the string contains a valid numeric format, CHKNUM returns the value 1. If the string contains characters that are not valid in a number, CHKNUM returns zero (0).

**Syntax:** How to Check the Format of a Character String

```plaintext
CHKNUM(numchar, source_string, output)
```

where:

- **numchar**
  Integer
  Is the number of characters in the string.

- **string**
  Alphanumeric
  Is the character string to be checked enclosed in single quotation marks, or a field or variable that contains the character string.

- **output**
  Numeric
  Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.
**Example:** Checking a String for Numeric Format

CHKNUM examines the strings STR1, STR2, and STR3 for numeric format.

```plaintext
DEFINE FILE WF_RETAIL_LITE
STR1/A8 = '12345E01';
STR2/A8 = 'ABCDEFG';
STR3/A8 = '1234.567';
CHK1/I1= CHKNUM(8,STR1,CHK1);
CHK2/I1= CHKNUM(8,STR2,CHK2);
CHK3/I1= CHKNUM(8,STR3,CHK3);
END

TABLE FILE WF_RETAIL_LITE
PRINT STR1 IN 20 CHK1 STR2 CHK2 STR3 CHK3
BY PRODUCT_CATEGORY
WHERE PRODUCT_CATEGORY EQ 'Video Production'
ON TABLE SET PAGE NOPAGE
ON TABLE PCHOLD FORMAT WP
END

The output is:

<table>
<thead>
<tr>
<th>Product Category</th>
<th>STR1</th>
<th>CHK1</th>
<th>STR2</th>
<th>CHK2</th>
<th>STR3</th>
<th>CHK3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Video Production</td>
<td>12345E01</td>
<td>1</td>
<td>ABCDEFG</td>
<td>0</td>
<td>1234.567</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>12345E01</td>
<td>1</td>
<td>ABCDEFG</td>
<td>0</td>
<td>1234.567</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>12345E01</td>
<td>1</td>
<td>ABCDEFG</td>
<td>0</td>
<td>1234.567</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>12345E01</td>
<td>1</td>
<td>ABCDEFG</td>
<td>0</td>
<td>1234.567</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>12345E01</td>
<td>1</td>
<td>ABCDEFG</td>
<td>0</td>
<td>1234.567</td>
<td>1</td>
</tr>
</tbody>
</table>

CHKNUM examines STR1 for numeric format.

CHKNUM(8, str1, 'I1')

For 12345E01, the result is 1.

For ABCDEFG, the result is 0.
CTRAN: Translating One Character to Another

Available Languages: reporting, Maintain

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 must know the decimal equivalent of the characters in internal machine representation. Note that the coding chart for conversion is platform dependent, hence your platform and configuration option determines whether ASCII, EBCDIC, or Unicode coding is used. Printable EBCDIC or ASCII characters and their decimal equivalents are listed in Character Chart for ASCII and EBCDIC on page 22.

In Unicode configurations, this function uses values in the range:

- 0 to 255 for 1-byte characters.
- 256 to 65535 for 2-byte characters.
- 65536 to 16777215 for 3-byte characters.
- 16777216 to 4294967295 for 4-byte characters (primarily for EBCDIC).

Syntax: How to Translate One Character to Another

CTRAN(length, source_string, decimal, decvalue, output)

where:

length

  Integer

  Is the number of characters in the source string, or a field that contains the length.

source_string

  Alphanumeric

  Is the character string to be translated enclosed in single quotation marks (’), or the field or variable that contains the character string.

decimal

  Integer
CTRAN: Translating One Character to Another

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

\textit{decvalue}  
Integer  

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

\textit{output}  
Alphanumeric  

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

\textbf{Example:} \textbf{Translating Spaces to Underscores on an ASCII Platform}  
CTRAN translates the spaces in ADDRESS\_LN3 (ASCII decimal value 32) to underscores (ASCII decimal value 95), and stores the result in ALT\_ADDR:

\begin{verbatim}
TABLE FILE EMPLOYEE
PRINT ADDRESS\_LN3 AND COMPUTE
ALT\_ADDR/A20 = CTRAN(20, ADDRESS\_LN3, 32, 95, ALT\_ADDR);
BY EMP\_ID
WHERE TYPE EQ 'HSM';
END
\end{verbatim}

The output is:

\begin{verbatim}
EMP\_ID     ADDRESS\_LN3           ALT\_ADDR
------     -----------           --------
117593129  RUTHERFORD NJ 07073   RUTHERFORD_NJ_07073_
119265415  NEW YORK NY 10039     NEW_YORK_NY_10039___
119329144  FREEPORT NY 11520     FREEPORT_NY_11520___
123764317  NEW YORK NY 10001     NEW_YORK_NY_10001___
126724188  FREEPORT NY 11520     FREEPORT_NY_11520___
451123478  ROSELAND NJ 07068     ROSELAND_NJ_07068___
543729165  JERSEY CITY NJ 07300  JERSEY_CITY_NJ_07300
810692173  FLUSHING NY 11354     FLUSHING_NY_11354
\end{verbatim}

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\_.

138  Information Builders
Example: Translating Spaces to Underscores on an EBCDIC Platform

CTRAN translates the spaces in ADDRESS_LN3 (EBCDIC decimal value 64) to underscores (EBCDIC decimal value 109) and stores the result in ALT_ADDR:

```
TABLE FILE EMPLOYEE
PRINT ADDRESS_LN3 AND COMPUTE
ALT_ADDR/A20 = CTRAN(20, ADDRESS_LN3, 64, 109, ALT_ADDR);
BY EMP_ID
WHERE TYPE EQ 'HSM'
END
```

The output is:

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>ADDRESS_LN3</th>
<th>ALT_ADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>117593129</td>
<td>RUTHERFORD NJ 07073</td>
<td>RUTHERFORD_NJ_07073_</td>
</tr>
<tr>
<td>119265415</td>
<td>NEW YORK NY 10039</td>
<td>NEW_YORK_NY_10039____</td>
</tr>
<tr>
<td>119329144</td>
<td>FREEPORT NY 11520</td>
<td>FREEPORT_NY_11520___</td>
</tr>
<tr>
<td>123764317</td>
<td>NEW YORK NY 10001</td>
<td>NEW_YORK_NY_10001____</td>
</tr>
<tr>
<td>126724188</td>
<td>FREEPORT NY 11520</td>
<td>FREEPORT_NY_11520___</td>
</tr>
<tr>
<td>451123478</td>
<td>ROSELAND NJ 07068</td>
<td>ROSELAND_NJ_07068___</td>
</tr>
<tr>
<td>543729165</td>
<td>JERSEY CITY NJ 07300</td>
<td>JERSEY_CITY_NJ_07300</td>
</tr>
<tr>
<td>818692173</td>
<td>FLUSHING NY 11354</td>
<td>FLUSHING_NY_11354___</td>
</tr>
</tbody>
</table>

Example: Inserting Accented Letter E's With MODIFY

This MODIFY request enables you to enter the names of new employees containing the accented letter È, as in the name Adèle Molière. The equivalent EBCDIC decimal value for “an asterisk is 92, for an È, 159.

If you are using the Hot Screen facility, some characters cannot be displayed. If Hot Screen does not support the character you need, disable Hot Screen with SET SCREEN=OFF and issue the RETYPE command. If your terminal can display the character, the character appears. The display of special characters depends upon your software and hardware; not all special characters may display.
The request is:

```
MODIFY FILE EMPLOYEE
CRTFORM
"***** NEW EMPLOYEE ENTRY SCREEN *****"
""
"ENTER EMPLOYEE'S ID: <EMP_ID"
""
"ENTER EMPLOYEE'S FIRST AND LAST NAME"
"SUBSTITUTE *'S FOR ALL ACCENTED E CHARACTERS"
""
"FIRST_NAME: <FIRST_NAME LAST_NAME: <LAST_NAME"
""
"ENTER THE DEPARTMENT ASSIGNMENT: <DEPARTMENT"
MATCH EMP_ID
ON MATCH REJECT
ON NOMATCH COMPUTE
  FIRST_NAME/A10 = CTRAN(10, FIRST_NAME, 92, 159, 'A10');
  LAST_NAME/A15 = CTRAN(15, LAST_NAME, 92, 159, 'A15');
ON NOMATCH TYPE "FIRST_NAME: <FIRST_NAME LAST_NAME: <LAST_NAME"
ON NOMATCH INCLUDE
DATA
END
```

A sample execution follows:

```
***** NEW EMPLOYEE ENTRY SCREEN *****

ENTER EMPLOYEE'S ID: 999888777

ENTER EMPLOYEE'S FIRST AND LAST NAME
SUBSTITUTE *'S FOR ALL ACCENTED E CHARACTERS

FIRST_NAME: AD*LE       LAST_NAME: MOLI*RE

ENTER THE DEPARTMENT ASSIGNMENT: SALES
```

The request processes as:

1. The CRTFORM screen prompts you for an employee ID, first name, last name, and department assignment. It requests that you substitute an asterisk (*) whenever the accented letter È appears in a name.

2. Enter the following data:

   EMPLOYEE ID: 999888777

   FIRST_NAME: AD*LE

   LAST_NAME: MOLI*RE

   DEPARTMENT: SALES

3. The procedure searches the data source for the employee ID. If it does not find it, it continues processing the request.
4. CTRAN converts the asterisks into È's in both the first and last names (ADÈLE MOLIÈRE).

***** NEW EMPLOYEE ENTRY SCREEN *****

ENTER EMPLOYEE'S ID:

ENTER EMPLOYEE'S FIRST AND LAST NAME
SUBSTITUTE *'S FOR ALL ACCENTED E CHARACTERS

FIRST_NAME:              LAST_NAME:

ENTER THE DEPARTMENT ASSIGNMENT:

FIRST_NAME: ADÈLE LAST_NAME: MOLIÈRE

5. The procedure stores the data in the data source.

Example: Inserting Commas With MODIFY

This MODIFY request adds records of new employees to the EMPLOYEE data source. The PROMPT command prompts you for data one field at a time. CTRAN enables you to enter commas in names without having to enclose the names in single quotation marks. Instead of typing the comma, you type a semicolon, which is converted by CTRAN into a comma. The equivalent EBCDIC decimal value for a semicolon is 94; for a comma, 107.

The request is:

MODIFY FILE EMPLOYEE
PROMPT EMP_ID LAST_NAME FIRST_NAME DEPARTMENT
MATCH EMP_ID
  ON MATCH REJECT
  ON NOMATCH COMPUTE
    LAST_NAME/A15 = CTRAN(15, LAST_NAME, 94, 107, 'A15');
  ON NOMATCH INCLUDE
DATA
A sample execution follows:

```
> EMPLOYEEFOCUS A ON 04/19/96 AT 16.07.29
DATA FOR TRANSACTION 1

  EMP_ID      =
224466880
  LAST_NAME   =
BRADLEY; JR.
  FIRST_NAME  =
JOHN
  DEPARTMENT  =
MIS
DATA FOR TRANSACTION 2
EMP_ID      =
end
```

The request processes as:

1. The request prompts you for an employee ID, last name, first name, and department assignment. Enter the following data:

   EMP_ID: 224466880
   LAST_NAME: BRADLEY; JR.
   FIRST_NAME: JOHN
   DEPARTMENT: MIS

2. The request searches the data source for the ID 224466880. If it does not find the ID, it continues processing the transaction.

3. CTRAN converts the semicolon in "BRADLEY; JR." to a comma. The last name is now "BRADLEY, JR."

4. The request adds the transaction to the data source.

5. This request displays the semicolon converted to a comma:

```
TABLE FILE EMPLOYEE
PRINT EMP_ID LAST_NAME FIRST_NAME DEPARTMENT
IF EMP_ID IS 224466880
END
```

The output is:

```
EMP_ID     LAST_NAME        FIRST_NAME  DEPARTMENT
------     ---------        ----------  ----------
224466880  BRADLEY, JR.     JOHN        MIS
```
CTRFLD: Centering a Character String

Available Languages: reporting, Maintain

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.

Limit: Using CTRFLD in a styled report (StyleSheets feature) generally negates the effect of CTRFLD unless the item is also styled as a centered element. Also, if you are using CTRFLD on a platform for which the default font is proportional, either use a non-proportional font, or issue SET STYLE=OFF before running the request.

Syntax: How to Center a Character String

CTRFLD(source_string, length, output)

where:

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

length
   Integer
   Is the number of characters in source_string and output, or a field that contains the length. This argument must be greater than 0. A length less than 0 can cause unpredictable results.

output
   Alphanumeric
   Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.
**Example:** Centering a Field

CTRFLD centers LAST_NAME and stores the result in CENTER_NAME:

```plaintext
SET STYLE=OFF
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND COMPUTE
CENTER_NAME/A12 = CTRFLD(LAST_NAME, 12, 'A12');
WHERE DEPARTMENT EQ 'MIS'
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>CENTER_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>SMITH</td>
</tr>
<tr>
<td>JONES</td>
<td>JONES</td>
</tr>
<tr>
<td>MCCOY</td>
<td>MCCOY</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>BLACKWOOD</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>GREENSPAN</td>
</tr>
<tr>
<td>CROSS</td>
<td>CROSS</td>
</tr>
</tbody>
</table>

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

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

**EDIT: Extracting or Adding Characters**

Available Languages: reporting

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 nine (9) in the mask, EDIT copies the corresponding character from the source field 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.

**Note:**

- EDIT does not require an output argument because the result is alphanumeric and its size is determined from the mask value.
EDIT can also convert the format of a field. For information on converting a field with EDIT, see *EDIT: Converting the Format of a Field* on page 412.

**Syntax:**

**How to Extract or Add Characters**

```plaintext
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 `source_string` must be at least as large as the number of 9's in the mask.

- **mask**
  - Alphanumeric
  - Is a string of mask characters enclosed in single quotation marks or a field containing the character string enclosed in single quotation marks. The length of the mask, excluding characters other than 9 and $, determines the length of the output field.

**Example:**

**Extracting and Adding Characters**

EDIT extracts the first initial from the FIRST_NAME field and stores the result in FIRST_INIT. EDIT also adds dashes to the EMP_ID field and stores the result in EMPIDEDIT. The mask used to extract the first initial is stored in the virtual field named MASK1:

```plaintext
DEFINE FILE EMPLOYEE
MASK1/A10 = '9$$$$$$$$$
END
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND COMPUTE
FIRST_INIT/A1 = EDIT(FIRST_NAME, MASK1);
EMPIDEDIT/A11 = EDIT(EMP_ID, '999-99-9999');
WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_INIT</th>
<th>EMPIDEDIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>M</td>
<td>112-84-7612</td>
</tr>
<tr>
<td>JONES</td>
<td>D</td>
<td>117-59-3129</td>
</tr>
<tr>
<td>MCCOY</td>
<td>J</td>
<td>219-98-4371</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>R</td>
<td>326-17-9357</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>M</td>
<td>543-72-9165</td>
</tr>
<tr>
<td>CROSS</td>
<td>B</td>
<td>818-69-2173</td>
</tr>
</tbody>
</table>
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)**

Available Languages: reporting, Maintain

The GETTOK function divides a character string into substrings, called tokens. The data must have a specific character, called a delimiter, that occurs in the string and separates the string into tokens. GETTOK returns the token specified by the `token_number` argument. GETTOK 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 158.

**Syntax:**

```
GETTOK(source_string, inlen, token_number, 'delim', outlen, output)
```

where:

*source_string*

Alphanumeric

Is the source string from which to extract the token.

*inlen*

Integer

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

*token_number*

Integer

Is the number of the token 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 the delimiter in the source string enclosed in single quotation marks. If you specify more
than one character, only the first character is used.

Note: In Dialogue Manager, to prevent the conversion of a delimiter space character (' ') to
a double precision zero, include a non-numeric character after the space (for example, '%').
GETTOK uses only the first character (the space) as a delimiter, while the extra character
(%) prevents conversion to double precision.

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
Alphanumeric
Is the name of the field that contains the token, or the format of the output value enclosed
in single quotation marks. The delimiter is not included in the token.

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

Example: Extracting a Token
GETTOK extracts the last token from ADDRESS_LN3 and stores the result in LAST_TOKEN.

The delimiter is a space:

```
TABLE FILE EMPLOYEE
PRINT ADDRESS_LN3 AND COMPUTE
LAST_TOKEN/A10 = GETTOK(ADDRESS_LN3, 20, -1, ' ', 10, LAST_TOKEN);
AS 'LAST TOKEN, (ZIP CODE)'
WHERE TYPE EQ 'HSM';
END
```

The output is:

```
ADDRESS_LN3           LAST_TOKEN
-----------           ----------
RUTHERFORD NJ 07073   07073
NEW YORK NY 10039     10039
FREEPORT NY 11520     11520
NEW YORK NY 10001     10001
FREEPORT NY 11520     11520
ROSELAND NJ 07068     07068
JERSEY CITY NJ 07300   07300
FLUSHING NY 11354     11354
```
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.

**LCWORD: Converting a String to Mixed-Case**

Available Languages: reporting, Maintain

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 new word and the first letter after a single or double quotation mark, which it converts to uppercase. 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)

where:

*length*

Integer

Is the number of characters in source_string and output.

*string*

Alphanumeric

Is the character string to be converted enclosed in single quotation marks, or a field or variable containing the character string.

*output*

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 must be greater than or equal to length.
**Example: Converting a Character String to Mixed-Case**

LCWORD converts the LAST_NAME field to mixed-case and stores the result in MIXED_CASE.

```plaintext
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND COMPUTE
MIXED_CASE/A15 = LCWORD(15, LAST_NAME, MIXED_CASE);
WHERE DEPARTMENT EQ 'PRODUCTION'
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>MIXED_CASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEVENS</td>
<td>Stevens</td>
</tr>
<tr>
<td>SMITH</td>
<td>Smith</td>
</tr>
<tr>
<td>BANNING</td>
<td>Banning</td>
</tr>
<tr>
<td>IRVING</td>
<td>Irving</td>
</tr>
<tr>
<td>ROMANS</td>
<td>Romans</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>Mcknight</td>
</tr>
</tbody>
</table>

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.

**LCWORD2: Converting a String to Mixed-Case**

Available Languages: reporting, Maintain

The LCWORD2 function converts the letters in a character string to mixed-case by converting the first letter of each word to uppercase and converting every other letter to lowercase. In addition, a double quotation mark or a space indicates that the next letter should be converted to uppercase.

For example, "SMITH" would be changed to "Smith" and "JACK S" would be changed to "Jack S".
**Syntax:** How to Convert a Character String to Mixed-Case

\[ \text{LCWORD2}(\text{length}, \text{string}, \text{output}) \]

where:

- **length**
  - Integer
  - Is the length, in characters, of the character string or field to be converted, or a field that contains the length.

- **string**
  - Alphanumeric
  - Is the character string to be converted, or a temporary field that contains the string.

- **output**
  - 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 must be greater than or equal to \( \text{length} \).

**Example:** Converting a Character String to Mixed-Case

\[ \text{LCWORD2} \text{ converts the string 'O'CONNOR's'} \]

\begin{verbatim}
DEFINE FILE EMPLOYEE
MYVAL1/A10='O'CONNOR'S';
LC2/A10 = LCWORD2(10, MYVAL1, 'A10');
END
TABLE FILE EMPLOYEE
SUM LAST_NAME NOPRINT MYVAL1 LC2
END
\end{verbatim}

The output is:

\begin{verbatim}
MYVAL1 LC2
------ ---
O'CONNOR'S O'Connor's
\end{verbatim}

The value returned is O'Connor's.
LCWORD3: Converting a String to Mixed-Case

The LCWORD3 function converts the letters in a character string to mixed-case by converting the first letter of each word to uppercase and converting every other letter to lowercase. In addition, a single quotation mark indicates that the next letter should be converted to uppercase, as long as it is neither followed by a blank nor the last character in the input string.

For example, 'SMITH' would be changed to 'Smith' and JACK’S would be changed to Jack's.

Syntax: How to Convert a Character String to Mixed-Case Using LCWORD3

\[ \text{LCWORD3}(\text{length}, \text{string}, \text{output}) \]

where:

- **length**
  - Integer
  - Is the length, in characters, of the character string or field to be converted, or a field that contains the length.

- **string**
  - Alphanumeric
  - Is the character string to be converted, or a field that contains the string.

- **output**
  - 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 must be greater than or equal to **length**.

Example: Converting a Character String to Mixed-Case Using LCWORD3

LCWORD3 converts the strings O'CONNOR's and o'connor's to mixed-case:

```plaintext
DEFINE FILE EMPLOYEE
MYVAL1/A10='O'CONNOR'S';
MYVAL2/A10='o'connor's';
LC1/A10 = LCWORD3(10, MYVAL1, 'A10');
LC2/A10 = LCWORD3(10, MYVAL2, 'A10');
END
TABLE FILE EMPLOYEE
SUM LAST_NAME NOPRINT MYVAL1 LC1 MYVAL2 LC2
END
```
On the output, the letter C after the first single quotation mark is in uppercase because it is not followed by a blank and is not the final letter in the input string. The letter s after the second single quotation mark (') is in lowercase because it is the last character in the input string:

```
MYVAL1   LC1    MYVAL2   LC2
-------   ---    -------   ---
O'CONNOR'S  O'Connor's  o'connor's  O'Connor's
```

For the string O'CONNOR's, LCWORD3 returns O'Connor's.

For the string o'connor's, LCWORD3 also returns O'Connor's.

**LJUST: Left-Justifying a String**

Available Languages: reporting

LJUST left-justifies a character string within a field. All leading spaces become trailing spaces. LJUST will not have any visible effect in a report that uses StyleSheets (SET STYLE=ON) unless you center the item.

There is a version of the LJUST function that is available only in the Maintain language. For information on this function, see *LJUST: Left-Justifying a Character String (Maintain)*.

**Syntax:**

How to Left-Justify a Character String

```
LJUST(length, source_string, output)
```

where:

- **length**
  
  Integer

  Is the number of characters in *source_string* and *output*, or a field that contains the length.

- **source_string**

  Alphanumeric

  Is the character string to be justified, or a field or variable that contains the string.

- **output**

  Alphanumeric

  Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.
**Example:** Left-Justifying a String

The following request creates the XNAME field in which the last names are not left-justified. Then, LJUST left-justifies the XNAME field and stores the result in YNAME.

```
SET STYLE=OFF
DEFINE FILE EMPLOYEE
XNAME/A25=IF LAST_NAME EQ 'BLACKWOOD' THEN '    '|LAST_NAME ELSE
   ''|LAST_NAME;
YNAME/A25=LJUST(15, XNAME, 'A25');
END

TABLE FILE EMPLOYEE
PRINT LAST_NAME XNAME YNAME
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>XNAME</th>
<th>YNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEVENS</td>
<td>STEVENS</td>
<td>STEVENS</td>
</tr>
<tr>
<td>SMITH</td>
<td>SMITH</td>
<td>SMITH</td>
</tr>
<tr>
<td>JONES</td>
<td>JONES</td>
<td>JONES</td>
</tr>
<tr>
<td>SMITH</td>
<td>SMITH</td>
<td>SMITH</td>
</tr>
<tr>
<td>BANNING</td>
<td>BANNING</td>
<td>BANNING</td>
</tr>
<tr>
<td>IRVING</td>
<td>IRVING</td>
<td>IRVING</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ROMANS</td>
<td>ROMANS</td>
</tr>
<tr>
<td>MCCOY</td>
<td>MCCOY</td>
<td>MCCOY</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>BLACKWOOD</td>
<td>BLACKWOOD</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>MCKNIGHT</td>
<td>MCKNIGHT</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>GREENSPAN</td>
<td>GREENSPAN</td>
</tr>
<tr>
<td>CROSS</td>
<td>CROSS</td>
<td>CROSS</td>
</tr>
</tbody>
</table>

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

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

**LOCASE: Converting Text to Lowercase**

Available Languages: reporting, Maintain

The LOCASE function converts alphanumeric text to lowercase.

It is useful for converting input fields from FIDEL CRTFORMs and non-FOCUS applications to lowercase.
**Syntax:**

How to Convert Text to Lowercase

\[
\text{LOCASE}(\text{length}, \text{source}\_\text{string}, \text{output})
\]

where:

- **length**
  - Integer
  - Is the number of characters in \text{source}\_\text{string} and \text{output}, or a field that contains the length.
  - The length must be greater than 0 and the same for both arguments; otherwise, an error occurs.

- **source\_string**
  - Alphanumeric
  - Is the character string to convert in single quotation marks, or a field or variable that contains the string.

- **output**
  - Alphanumeric
  - Is the name of the field in which to store the result, or the format of the output value enclosed in single quotation marks. The field name can be the same as \text{source}\_\text{string}.

**Example:**

Converting a String to Lowercase

LOCASE converts the LAST\_NAME field to lowercase and stores the result in LOWER\_NAME:

```
TABLE FILE EMPLOYEE
PRINT LAST\_NAME AND COMPUTE
LOWER\_NAME/15 = LOCASE(15, LAST\_NAME, LOWER\_NAME); WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>LOWER_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>smith</td>
</tr>
<tr>
<td>JONES</td>
<td>jones</td>
</tr>
<tr>
<td>MCCOY</td>
<td>mccoy</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>blackwood</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>greenspan</td>
</tr>
<tr>
<td>CROSS</td>
<td>cross</td>
</tr>
</tbody>
</table>

LOCASE converts LAST\_NAME to lowercase and stores the result in a column with the format A15:

\[
\text{LOCASE}(15, \text{LAST}\_\text{NAME}, 'A15')
\]

For SMITH, the result is smith.
For JONES, the result is jones.

**OVRLAY: Overlaying a Character String**

Available Languages: reporting

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.

There is a version of the OVRLAY function that is available only in the Maintain language. For information on this function, see OVRLAY: Overlaying a Character String (Maintain).

**Syntax:** How to Overlay a Character String

$$\text{OVRLAY}(source\_string, length, substring, sublen, position, output)$$

where:

- **source_string**
  - Alphanumeric
  - Is the base character string.

- **stringlen**
  - Integer
  - Is the number of characters in **source_string** and **output**, or a field that contains the length.
  - If this argument is less than or equal to 0, unpredictable results occur.

- **substring**
  - Alphanumeric
  - Is the substring that will overlay **source_string**.

- **sublen**
  - Integer
  - Is the number of characters in **substring**, or a field that contains the length. If this argument is less than or equal to 0, the function returns spaces.

- **position**
  - Integer
  - Is the position in **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 **stringlen**, the function returns the source string.
Alphanumeric

Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. If the overlaid string is longer than the output field, the string is truncated to fit the field.

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 security identification code and stores the result in NEW_ID:

```
TABLE FILE EMPLOYEE
PRINT EMP_ID AND CURR_JOBCODE AND COMPUTE
NEW_ID/A9 = OVRLAY(EMP_ID, 9, CURR_JOBCODE, 3, 7, NEW_ID);
BY LAST_NAME BY FIRST_NAME
WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>EMP_ID</th>
<th>CURR_JOBCODE</th>
<th>NEW_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>326179357</td>
<td>B04</td>
<td>326179B04</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>818692173</td>
<td>A17</td>
<td>818692A17</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>543729165</td>
<td>A07</td>
<td>543729A07</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>117593129</td>
<td>B03</td>
<td>117593B03</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>219984371</td>
<td>B02</td>
<td>219984B02</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>112847612</td>
<td>B14</td>
<td>112847B14</td>
</tr>
</tbody>
</table>

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.
**Example:** Overlaying a Character in a String With MODIFY

This MODIFY procedure prompts for input using a CRTFORM screen and updates first names in the EMPLOYEE data source. The CRTFORM LOWER option enables you to update the names in lowercase, but the procedure ensures that the first letter of each name is capitalized.

```sql
MODIFY FILE EMPLOYEE
CRTFORM LOWER
"ENTER EMPLOYEE'S ID: <EMP_ID"
"ENTER FIRST_NAME IN LOWER CASE: <FIRST_NAME"
MATCH EMP_ID
ON NOMATCH REJECT
ON MATCH COMPUTE
  F_UP/A1 = UPCASE(1, FIRST_NAME, 'A1');
  FIRST_NAME/A10 = OVRLAY(FIRST_NAME, 10, F_UP, 1, 1, 'A10');
ON MATCH TYPE "CHANGING FIRST NAME TO <FIRST_NAME"
DATA
END
```

The COMPUTE command invokes two functions:

- **UPCASE** extracts the first letter and converts it to uppercase.
- **OVRLAY** replaces the original first letter in the name with the uppercase initial.

The procedure processes as:

1. The procedure prompts you from a CRTFORM screen for an employee ID and a first name. Type the following data and press Enter:
   
   Enter the employee’s ID: 071382660
   
   Enter the first name in lowercase: alfred

2. The procedure searches the data source for the ID 071382660. If it finds the ID, it continues processing the transaction. In this case, the ID exists and belongs to Alfred Stevens.

3. **UPCASE** extracts the letter a from alfred and converts it to the letter A.

4. **OVRLAY** overlays the letter A on alfred. The first name is now Alfred.

   ENTER EMPLOYEE'S ID:
   ENTER FIRST_NAME IN LOWER CASE:
   CHANGING FIRST NAME TO Alfred

5. The procedure updates the first name in the data source.

6. When you exit the procedure with PF3, the transaction message indicates that one update occurred:
Available Languages: reporting, Maintain

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 146). 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 substrings 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)
where:

\textit{length} \\
\text{Integer} \\
\text{Is the number of characters in source\_string and output, or a field that contains the length.}

\textit{source\_string} \\
\text{Alphanumeric} \\
\text{Is a string to divide into tokens enclosed in single quotation marks, or a field or variable that contains the text.}

\textit{delimiter} \\
\text{Alphanumeric} \\
\text{Is the delimiter enclosed in single quotation marks. Choose a character that does not appear in the text.}

\textit{max\_token\_size} \\
\text{Integer} \\
\text{Is the upper limit for the size of each token.}

\textit{output} \\
\text{Alphanumeric} \\
\text{Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.}

\textbf{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. It then stores the result in PARA\_ADDR:

\begin{verbatim}
TABLE FILE EMPLOYEE
PRINT ADDRESS\_LN2 AND COMPUTE
PARA\_ADDR/A20 = PARAG(20, ADDRESS\_LN2, '\'', 10, PARA\_ADDR);
BY LAST\_NAME
WHERE TYPE EQ 'HSM';
END
\end{verbatim}
The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>ADDRESS_LN2</th>
<th>PARA_ADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>APT 4C</td>
<td>APT 4C,</td>
</tr>
<tr>
<td>CROSS</td>
<td>147-15 NORTHERN BLD</td>
<td>147-15,NORTHERN,BLD</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>13 LINDEN AVE.</td>
<td>13 LINDEN,AVE.</td>
</tr>
<tr>
<td>IRVING</td>
<td>123 E 32 ST.</td>
<td>123 E 32,ST.</td>
</tr>
<tr>
<td>JONES</td>
<td>235 MURRAY HIL PKWY</td>
<td>235 MURRAY,HIL PKWY</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>117 HARRISON AVE.</td>
<td>117,HARRISON,AVE.</td>
</tr>
<tr>
<td>ROMANS</td>
<td>271 PRESIDENT ST.</td>
<td>271,PRESIDENT,ST.</td>
</tr>
<tr>
<td>SMITH</td>
<td>136 E 161 ST.</td>
<td>136 E 161,ST.</td>
</tr>
</tbody>
</table>

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 a 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 source string. This function is useful for examining data to make sure that it follows a standard pattern.

In the output pattern:

- Any character from the input that represents a single-byte digit becomes the character 9.

- Any character that represents an uppercase letter becomes A, and any character that represents a lowercase letter becomes a. For European NLS mode (Western Europe, Central Europe), A and a are extended to apply to accented alphabets.

- For Japanese, double-byte characters and Hankaku-katakana become C (uppercase). Note that double-byte includes Hiragana, Katakana, Kanji, full-width alphabets, full-width numbers, and full-width symbols. This means that all double-byte letters such as Chinese and Korean are also represented as C.

- Special characters remain unchanged.

- An unprintable character becomes the character X.
**Syntax:**

How to Generate a Pattern From an Input String

```
PATTERN (length, source_string, output)
```

where:

- **length**
  - Numeric
  - Is the length of `source_string`.

- **source_string**
  - Alphanumeric
  - Is the source string enclosed in single quotation marks, or a field containing the source string.

- **output**
  - 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
800–969–INFO
```
The Master File is:

```plaintext
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.

```plaintext
DYNAM ALLOC DD TESTFILE DA USER1.TESTFILE.FTMFILEDEF TESTFILE DISK
  testfile.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:

```
<table>
<thead>
<tr>
<th>PATTERN</th>
<th>COUNT</th>
<th>PERCENT</th>
<th>TESTFLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>(999) 999 9999</td>
<td>3</td>
<td>15.79</td>
<td>(212) 736 5533</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(212) 736 5533</td>
</tr>
<tr>
<td>(999) 999–9999</td>
<td>2</td>
<td>10.53</td>
<td>(212) 736–6250</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(212) 736–4433</td>
</tr>
<tr>
<td>9 Aaaa Aaaaa</td>
<td>1</td>
<td>5.26</td>
<td>2 Penn Plaza</td>
</tr>
<tr>
<td>999 999 9999</td>
<td>1</td>
<td>5.26</td>
<td>212 736 4433</td>
</tr>
<tr>
<td>999–99–9999</td>
<td>1</td>
<td>5.26</td>
<td>123–45–6789</td>
</tr>
<tr>
<td>999–999–AAAA</td>
<td>2</td>
<td>10.53</td>
<td>800–969–INFO</td>
</tr>
<tr>
<td>999–999–9999</td>
<td>6</td>
<td>31.58</td>
<td>212–736–6250</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>917–339–6380</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>917–339–4350</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>212–736–6250</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>212–736–6250</td>
</tr>
<tr>
<td>99999</td>
<td>1</td>
<td>5.26</td>
<td>10121</td>
</tr>
<tr>
<td>99999 X</td>
<td>1</td>
<td>5.26</td>
<td>10121 &amp;</td>
</tr>
<tr>
<td>99999–9999</td>
<td>1</td>
<td>5.26</td>
<td>10121–2898</td>
</tr>
<tr>
<td>TOTAL</td>
<td>19</td>
<td>100.00</td>
<td></td>
</tr>
</tbody>
</table>
```
PATTERN generates a pattern for each instance of TESTFLD. The result is stored in a column with the format A14:

```
PATTERN (14, TESTFLD, 'A14' )
```

For 212-736-6250, the result is 999-999-9999.
For 800-969-INFO, the result is 1999-999-AAAA.

**POSIT: Finding the Beginning of a Substring**

Available Languages: reporting

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 parent string, the function returns the value 0.

There is a version of the POSIT function that is available only in the Maintain language. For information on this function, see *POSIT: Finding the Beginning of a Substring (Maintain).*

**Syntax:**  
**How to Find the Beginning of a Substring**

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

where:

- **source_string**  
  Alphanumeric
  
  Is the string to parse enclosed in single quotation marks, or a field or variable that contains the source character string.

- **length**  
  Integer
  
  Is the number of characters in the source string, or a field that contains the length. 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. This can be the substring enclosed in single quotation marks, or the field that contains the string.

- **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.
**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 I_IN_NAME:

```
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND COMPUTE
I_IN_NAME/I2 = POSIT(LAST_NAME, 15, 'I', 1, 'I2');
WHERE DEPARTMENT EQ 'PRODUCTION'
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>I_IN_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEVENS</td>
<td>0</td>
</tr>
<tr>
<td>SMITH</td>
<td>3</td>
</tr>
<tr>
<td>BANNING</td>
<td>5</td>
</tr>
<tr>
<td>IRVING</td>
<td>1</td>
</tr>
<tr>
<td>ROMANS</td>
<td>0</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>5</td>
</tr>
</tbody>
</table>

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 the Characters in a String**

The REVERSE function reverses the characters in a string. This reversal includes all trailing blanks, which then become leading blanks. However, in an HTML report with SET SHOWBLANKS=OFF (the default value), the leading blanks are not visible.
### Syntax: How to Reverse the Characters in a String

**REVERSE(length, source_string, output)**

where:

- **length**
  - Integer
  - Is the number of characters in `source_string` and `output`, or a field that contains the length.

- **source_string**
  - Alphanumeric
  - Is the character string to reverse enclosed in single quotation marks, or a field that contains the character string.

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

### Example: Reversing the Characters in a String

In the following request against the EMPLOYEE data source, the REVERSE function is used to reverse the characters in the `LAST_NAME` field to produce the field named `REVERSE_LAST`. In this field, the trailing blanks from `LAST_NAME` have become leading blanks. The TRIM function is used to strip the leading blanks from `REVERSE_LAST` to produce the field named `TRIM_REVERSE`:

```plaintext
DEFINE FILE EMPLOYEE
REVERSE_LAST/A15 = REVERSE(15, LAST_NAME, REVERSE_LAST);
TRIM_REVERSE/A15 = TRIM('L', REVERSE_LAST, 15, ' ', 1, 'A15');
END

TABLE FILE EMPLOYEE
PRINT REVERSE_LAST TRIM_REVERSE
BY LAST_NAME
END
```
The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>REVERSE_LAST</th>
<th>TRIM_REVERSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>GNINNAB</td>
<td>GNINNAB</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>DOOWKCALB</td>
<td>DOOWKCALB</td>
</tr>
<tr>
<td>CROSS</td>
<td>SSORC</td>
<td>SSORC</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>NAPSNEERG</td>
<td>NAPSNEERG</td>
</tr>
<tr>
<td>IRVING</td>
<td>GNIVRI</td>
<td>GNIVRI</td>
</tr>
<tr>
<td>JONES</td>
<td>SENOJ</td>
<td>SENOJ</td>
</tr>
<tr>
<td>MCCOY</td>
<td>YOCCM</td>
<td>YOCCM</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>THGINKCM</td>
<td>THGINKCM</td>
</tr>
<tr>
<td>ROMANS</td>
<td>SNAOMR</td>
<td>SNAOMR</td>
</tr>
<tr>
<td>SMITH</td>
<td>HTIMS</td>
<td>HTIMS</td>
</tr>
<tr>
<td>STEVENS</td>
<td>SNEVETS</td>
<td>SNEVETS</td>
</tr>
</tbody>
</table>

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**

Available Languages: reporting

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

RJUST does not have any visible effect in a report that uses StyleSheets (SET STYLE=ON) unless you center the item. Also, if you use RJUST on a platform on which StyleSheets are turned on by default, issue SET STYLE=OFF before running the request.

There is a version of the RJUST function that is available only in the Maintain language. For information on this function, see *RJUST: Right-Justifying a Character String (Maintain)*.

**Syntax:**

```
RJUST(length, source_string, output)
```

where:

`length`

Integer

Is the number of characters in `source_string` and `output`, or a field that contains the length. Their lengths must be the same to avoid justification problems.
source_string
Alphanumeric

Is the character string to right justify, or a field or variable that contains the character string enclosed in single quotation marks.

output
Alphanumeric

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

Example: Right-Justifying a String
RJUST right-justifies the LAST_NAME field and stores the result in RIGHT_NAME:

```
SET STYLE=OFF
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND COMPUTE
RIGHT_NAME/A15 = RJUST(15, LAST_NAME, RIGHT_NAME);
WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>RIGHT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>SMITH</td>
</tr>
<tr>
<td>JONES</td>
<td>JONES</td>
</tr>
<tr>
<td>MCCOY</td>
<td>MCCOY</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>BLACKWOOD</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>GREENSPAN</td>
</tr>
<tr>
<td>CROSS</td>
<td>CROSS</td>
</tr>
</tbody>
</table>

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
Available Languages: reporting, Maintain

The SOUNDEX function analyzes a character string phonetically, without regard to spelling. It converts character strings to four character codes. 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.

To conduct a phonetic search, do the following:

1. Use SOUNDEX to translate data values from the field you are searching for to the phonetic codes.
2. Use SOUNDEX to translate your best guess target string to a phonetic code. Remember that the spelling of your target string need be only approximate. However, the first letter must be correct.

3. Use WHERE or IF criteria to compare the temporary fields created in Step 1 to the temporary field created in Step 2.

**Syntax:**

How to Compare Character Strings Phonetically

*SOUNDEX*(length, source_string, output)

where:

*length*

Alphanumeric

Is the number of characters in *source_string*, or a field that contains the length. It can be a number enclosed in single quotation marks, or a field containing the number. The number must be from 01 to 99, expressed with two digits (for example '01'); a number larger than 99 causes the function to return asterisks (*) as output.

*source_string*

Alphanumeric

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

*output*

Alphanumeric

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

**Example:**

Comparing Character Strings Phonetically

The following request creates three fields:

- PHON_NAME contains the phonetic code of employee last names.
- PHON_COY contains the phonetic code of your guess, MICOY.
- PHON_MATCH contains YES if the phonetic codes match, NO if they do not.

The WHERE criteria selects the last name that matches your best guess.

```
DEFINE FILE EMPLOYEE
PHON_NAME/A4 = SOUNDEX('15', LAST_NAME, PHON_NAME);
PHON_COY/A4 WITH LAST_NAME = SOUNDEX('15', 'MICOY', PHON_COY);
PHON_MATCH/A3 = IF PHON_NAME IS PHON_COY THEN 'YES' ELSE 'NO';
END
```
TABLE FILE EMPLOYEE
PRINT LAST_NAME
IF PHON_MATCH IS 'YES'
END

The output is:

LAST_NAME
---------
MCCOY

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**

**Available Languages:** reporting, Maintain

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 THIRTY TWO DOLLARS AND FIFTY CENTS.

**Syntax:** How to Spell Out a Dollar Amount

SPELLNM(outlength, number, output)

where:

* outlength
  Integer
  Is the number of characters in output, or a field that contains the length.

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

<table>
<thead>
<tr>
<th>If number is less than...</th>
<th>...outlength should be</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10</td>
<td>37</td>
</tr>
<tr>
<td>$100</td>
<td>45</td>
</tr>
<tr>
<td>$1,000</td>
<td>59</td>
</tr>
<tr>
<td>$10,000</td>
<td>74</td>
</tr>
</tbody>
</table>
### SPELLNM: Spelling Out a Dollar Amount

<table>
<thead>
<tr>
<th>If number is less than...</th>
<th>...outlength should be</th>
</tr>
</thead>
<tbody>
<tr>
<td>$100,000</td>
<td>82</td>
</tr>
<tr>
<td>$1,000,000</td>
<td>96</td>
</tr>
</tbody>
</table>

**number**

Alphanumeric or Numeric (9.2)

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

**output**

Alphanumeric

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

**Example:** Spelling Out a Dollar Amount

SPELLNM spells out the values in CURR_SAL and stores the result in AMT_IN_WORDS:

```plaintext
TABLE FILE EMPLOYEE
PRINT CURR_SAL AND COMPUTE
AMT_IN_WORDS/A82 = SPELLNM(82, CURR_SAL, AMT_IN_WORDS);
WHERE DEPARTMENT EQ 'MIS' END
```

The output is:

<table>
<thead>
<tr>
<th>CURR_SAL</th>
<th>AMT_IN_WORDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$13,200.00</td>
<td>THIRTEEN THOUSAND TWO HUNDRED DOLLARS AND NO CENTS</td>
</tr>
<tr>
<td>$18,480.00</td>
<td>EIGHTEEN THOUSAND FOUR HUNDRED EIGHTY DOLLARS AND NO CENTS</td>
</tr>
<tr>
<td>$18,480.00</td>
<td>EIGHTEEN THOUSAND FOUR HUNDRED EIGHTY DOLLARS AND NO CENTS</td>
</tr>
<tr>
<td>$21,780.00</td>
<td>TWENTY-ONE THOUSAND SEVEN HUNDRED EIGHTY DOLLARS AND NO CENTS</td>
</tr>
<tr>
<td>$9,000.00</td>
<td>NINE THOUSAND DOLLARS AND NO CENTS</td>
</tr>
<tr>
<td>$27,062.00</td>
<td>TWENTY-SEVEN THOUSAND SIXTY-TWO DOLLARS AND NO CENTS</td>
</tr>
</tbody>
</table>

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

```plaintext
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

Available Languages: reporting, Maintain

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

\[
\text{SQUEEZ}(\text{length}, \text{source\_string}, \text{output})
\]

where:

\text{length}

Integer

Is the number of characters in \text{source\_string} and \text{output}, or a field that contains the length.

\text{source\_string}

Alphanumeric

Is the character string to squeeze enclosed in single quotation marks, or the field that contains the character string.

\text{output}

Alphanumeric

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

Example: Reducing Multiple Spaces to a Single Space

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

\begin{verbatim}
DEFINE FILE EMPLOYEE
NAME/A30 = FIRST_NAME | LAST_NAME;
END
TABLE FILE EMPLOYEE
PRINT NAME AND COMPUTE
SQNAME/A30 = SQUEEZ(30, NAME, 'A30');
WHERE DEPARTMENT EQ 'MIS';
END
\end{verbatim}
### The output is:

<table>
<thead>
<tr>
<th>NAME</th>
<th>SQNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>MARY</td>
<td>SMITH MARY SMITH</td>
</tr>
<tr>
<td>DIANE</td>
<td>JONES DIANE JONES</td>
</tr>
<tr>
<td>JOHN</td>
<td>MCCOY JOHN MCCOY</td>
</tr>
<tr>
<td>ROSEMARIE</td>
<td>BLACKWOOD ROSEMARIE BLACKWOOD</td>
</tr>
<tr>
<td>MARY</td>
<td>GREENSPAN MARY GREENSPAN</td>
</tr>
<tr>
<td>BARBARA</td>
<td>CROSS BARBARA CROSS</td>
</tr>
</tbody>
</table>

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

\[ \text{SQUEEZ}(30, \text{NAME}, 'A30') \]

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

**Available Languages:** reporting, Maintain

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

\[ \text{STRIP}(\text{length}, \text{source_string, char, output}) \]

where:

- **length**
  - Integer
  - Is the number of characters in \text{source_string} and \text{output}, or a field that contains the number.

- **source_string**
  - Alphanumeric
  - Is the string from which the character will be removed, or a field containing the string.
**char**

Alphanumeric

Is the character to be removed from the string. This can be an alphanumeric literal enclosed in single quotation marks, or a field that contains the character. 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**

Alphanumeric

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

**Example:** Removing Occurrences of a Character From a String

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

```
TABLE FILE MOVIES
PRINT DIRECTOR AND COMPUTE
SDIR/A17 = STRIP(17, DIRECTOR, '.', 'A17');
WHERE CATEGORY EQ 'COMEDY'
END
```

The output is:

<table>
<thead>
<tr>
<th>DIRECTORS</th>
<th>SDIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZEMECKIS R.</td>
<td>ZEMECKIS R</td>
</tr>
<tr>
<td>ABRAHAMS J.</td>
<td>ABRAHAMS J</td>
</tr>
<tr>
<td>ALLEN W.</td>
<td>ALLEN W</td>
</tr>
<tr>
<td>HALLSTROM L.</td>
<td>HALLSTROM L</td>
</tr>
<tr>
<td>MARSHALL P.</td>
<td>MARSHALL P</td>
</tr>
<tr>
<td>BROOKS J.L.</td>
<td>BROOKS JL</td>
</tr>
</tbody>
</table>

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.
Example:  Removing Single Quotation Marks From a String

STRIP removes all occurrences of a single quotation mark (’) from the TITLE field and stores the result in a field with the format A39:

TABLE FILE MOVIES
PRINT TITLE AND COMPUTE
STITLE/A39 = STRIP(39, TITLE, '''', 'A39');
WHERE TITLE CONTAINS '''
END

The output is:

<table>
<thead>
<tr>
<th>TITLE</th>
<th>STITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BABETTE’S FEAST</td>
<td>BABETTES FEAST</td>
</tr>
<tr>
<td>JANE FONDA’S COMPLETE WORKOUT</td>
<td>JANE FONDAS COMPLETE WORKOUT</td>
</tr>
<tr>
<td>JANE FONDA’S NEW WORKOUT</td>
<td>JANE FONDAS NEW WORKOUT</td>
</tr>
<tr>
<td>MICKEY MANTLE’S BASEBALLTIPS</td>
<td>MICKEY MANTLES BASEBALL TIPS</td>
</tr>
</tbody>
</table>

Example:  Removing Commas From a String (Maintain)

STRIP removes all occurrences of a comma from the TITLE field:

MAINTAIN FILE MOVIES
FOR 10 NEXT MOVIECODE INTO MOVSTK
  WHERE TITLE CONTAINS ',';
  COMPUTE I/I2=1;
  REPEAT MOVSTK.FOCINDEX
    TYPE "TITLE IS: <MOVSTK(I).TITLE"
    COMPUTE NOCOMMA/A39=STRIP(39,MOVSTK().TITLE, ',', NOCOMMA);
    TYPE "NEW TITLE IS: <NOCOMMA"
    COMPUTE I=I+1
  ENDFOR
END

The output is:

TITLE IS: SMURFS, THE
NEW TITLE IS: SMURFS THE

STRREP: Replacing Character Strings

The STRREP replaces all instances of a specified string within a source string. It also supports replacement by null strings.
### Syntax: How to Replace Character Strings

**STRREP** (inlength, instring, searchlength, searchstring, replength, repstring, outlength, output)

where:

- **inlength**
  - Numeric
  - Is the number of characters in the source string.

- **instring**
  - Alphanumeric
  - Is the source 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.

- **output**
  - Alphanumeric
  - Is the resulting output string after all replacements and padding.

### Reference: Usage Note for STRREP Function

The maximum string length is 4095.
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:

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

STRREP finds and replaces commas and then dollar signs and stores the result in field with the format A17:

```
STRREP(15,CS_ALPHA,1,’’,0,’X’,14,’A14’)
STRREP(14,CS_NOCOMMAS,1,’$’,4,’USD ’,17,’A17’)
```

For $29,700.00, the result is USD 29700.00.
For $9,000.00, the result is USD 9000.00.

SUBSTR: Extracting a Substring

Available Languages: reporting

The SUBSTR function extracts a substring based on where it begins and its length in the source string. SUBSTR can vary the position of the substring depending on the values of other fields.
There is a version of the SUBSTR function that is available only in the Maintain language. For information on this function, see SUBSTR: Extracting a Substring (Maintain).

**Syntax:**

**How to Extract a Substring**

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

where:

- **length**
  - Integer
  - Is the number of characters in `source_string`, or a field that contains the length.

- **source_string**
  - Alphanumeric
  - Is the string from which to extract a substring enclosed in single quotation marks, or the field containing the parent string.

- **start**
  - Integer
  - Is the starting position of the substring in the source string. If `start` is less than one or greater than `length`, the function returns spaces.

- **end**
  - Integer
  - Is the ending position of the substring. If this argument 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 `output`. Only `sublength` characters will be processed.

- **output**
  - 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 String**

POSIT determines the position of the first letter I in LAST_NAME and stores the result in I_IN_NAME. SUBSTR then extracts three characters beginning with the letter I from LAST_NAME, and stores the results in I_SUBSTR.

```
TABLE FILE EMPLOYEE
PRINT
COMPUTE
  I_IN_NAME/I2 = POSIT(LAST_NAME, 15, 'I', 1, 'I2'); AND
COMPUTE
  I_SUBSTR/A3 =
    SUBSTR(15, LAST_NAME, I_IN_NAME, I_IN_NAME+2, 3, I_SUBSTR);
BY LAST_NAME
WHERE DEPARTMENT EQ 'PRODUCTION'
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>I_IN_NAME</th>
<th>I_SUBSTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>5</td>
<td>ING</td>
</tr>
<tr>
<td>IRVING</td>
<td>1</td>
<td>IRV</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>5</td>
<td>IGH</td>
</tr>
<tr>
<td>ROMANS</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>SMITH</td>
<td>3</td>
<td>ITH</td>
</tr>
<tr>
<td>STEVENS</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Since Romans and Stevens have no I in their names, SUBSTR extracts a blank 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**

Available Languages: reporting

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

There is a version of the TRIM function that is available only in the Maintain language. For information on this function, see *TRIM: Removing Trailing Occurrences (Maintain)*.
Syntax: How to Remove Leading and Trailing Occurrences

\[ \text{TRIM}(\text{trim}_\text{where}, \text{source}_\text{string}, \text{length}, \text{pattern}, \text{sublength}, \text{output}) \]

where:

\text{trim}_\text{where}  
Alphanumeric  
Is one of the following, which indicates where to remove the pattern:  
'\text{L}' removes leading occurrences.  
'\text{T}' removes trailing occurrences.  
'\text{B}' removes both leading and trailing occurrences.

\text{source}_\text{string}  
Alphanumeric  
Is the string to trim enclosed in single quotation marks, or the field containing the string.

\text{string}_\text{length}  
Integer  
Is the number of characters in the source string.

\text{pattern}  
Alphanumeric  
Is the character string pattern to remove enclosed in single quotation marks.

\text{sublength}  
Integer  
Is the number of characters in the pattern.

\text{output}  
Alphanumeric  
Is the field to which the result is returned, or the format of the output value enclosed in single quotation marks.
**Example:**  **Removing Leading Occurrences**

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

```plaintext
TABLE FILE MOVIES
PRINT DIRECTOR AND
COMPUTE
  TRIMDIR/A17 = TRIM('L', DIRECTOR, 17, 'BR', 2, 'A17');
  WHERE DIRECTOR CONTAINS 'BR'
END
```

The output is:

```
DIRECTOR          TRIMDIR
--------          -------
ABRAHAMS J.       ABRAHAMS J.
BROOKS R.         OOKS R.
BROOKS J.L.       OOKS J.L.
```

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.

**Example:**  **Removing Trailing Occurrences**

TRIM removes trailing occurrences of the characters ER from the TITLE. In order to remove trailing non-blank characters, trailing spaces must be removed first. The TITLE field has trailing spaces. Therefore, TRIM does not remove the characters ER when creating field TRIMT. The SHORT field does not have trailing spaces. Therefore, TRIM removes the trailing ER characters when creating field TRIMS:

```plaintext
DEFINE FILE MOVIES
SHORT/A19 = SUBSTR(19, TITLE, 1, 19, 19, SHORT);
END
TABLE FILE MOVIES
PRINT TITLE IN 1 AS 'TITLE: ' SHORT IN 40 AS 'SHORT: ' OVER
COMPUTE
  TRIMT/A39 = TRIM('T', TITLE, 39, 'ER', 2, 'A39'); IN 1 AS 'TRIMT: '
  TRIMS/A19 = TRIM('T', SHORT, 19, 'ER', 2, 'A19'); IN 40 AS 'TRIMS: '
WHERE TITLE LIKE '%ER'
END
```
The output is:

<table>
<thead>
<tr>
<th>TITLE:</th>
<th>LEARN TO SKI BETTER</th>
<th>SHORT:</th>
<th>LEARN TO SKI BETTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRIMT:</td>
<td>LEARN TO SKI BETTER</td>
<td>TRIMS:</td>
<td>LEARN TO SKI BETT</td>
</tr>
<tr>
<td>TITLE:</td>
<td>FANNY AND ALEXANDER</td>
<td>SHORT:</td>
<td>FANNY AND ALEXANDER</td>
</tr>
<tr>
<td>TRIMT:</td>
<td>FANNY AND ALEXANDER</td>
<td>TRIMS:</td>
<td>FANNY AND ALEXAND</td>
</tr>
</tbody>
</table>

**UPCASE: Converting Text to Uppercase**

Available Languages: reporting

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 field.

In FIDEL, CRTFORM LOWER retains the case of entries exactly as they were typed. Use UPCASE to convert entries for particular fields to uppercase.

There is a version of the UPCASE function that is available only in the Maintain language. For information on this function, see *UPCASE: Converting Text to Uppercase (Maintain)*.

**Syntax:**

**UPCASE**(length, source_string, output)

where:

- **length**
  Integer
  Is the number of characters in source_string and output.

- **input**
  Alphanumeric
  Is the string to convert enclosed in single quotation marks, or the field containing the character string.

- **output**
  Alphanumeric of type AnV or An
  Is the field to which the result is returned, or the format of the output value enclosed in single quotation marks.
If the format of the output_format is AnV, then the length returned is equal to the smaller of the source_string length and the upper_limit length.

**Example: Converting a Mixed-Case String to Uppercase**

UPCASE converts the LAST_NAME_MIXED field to uppercase:

```
DEFINE FILE EMPLOYEE
LAST_NAME_MIXED/A15=IF DEPARTMENT EQ 'MIS' THEN LAST_NAME ELSE
   LCWORD(15, LAST_NAME, 'A15');
LAST_NAME_UPPER/A15=UPCASE(15, LAST_NAME_MIXED, 'A15') ;
END

TABLE FILE EMPLOYEE
PRINT LAST_NAME_MIXED AND FIRST_NAME BY LAST_NAME_UPPER
WHERE CURR_JOBCODE EQ 'B02' OR 'A17' OR 'B04';
END
```

Now, when you execute the request, the names are sorted correctly.

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME_UPPER</th>
<th>LAST_NAME_MIXED</th>
<th>FIRST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>Banning</td>
<td>JOHN</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
</tr>
<tr>
<td>CROSS</td>
<td>CROSS</td>
<td>BARBARA</td>
</tr>
<tr>
<td>MCCOY</td>
<td>MCCOY</td>
<td>JOHN</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>Mcknight</td>
<td>ROGER</td>
</tr>
<tr>
<td>ROMANS</td>
<td>Romans</td>
<td>ANTHONY</td>
</tr>
</tbody>
</table>

If you do not want to see the field with all uppercase values, you can NOPRINT it.

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.

**Example: Converting a Lowercase Field to Uppercase With MODIFY**

Suppose your company decides to store employee names in mixed case and the department assignments in uppercase.
To enter records for new employees, execute this MODIFY procedure:

```sql
MODIFY FILE EMPLOYEE
CRTFORM LOWER
"ENTER EMPLOYEE'S ID : <EMP_ID"
"ENTER LAST_NAME: <LAST_NAME FIRST_NAME: <FIRST_NAME"
"TYPE THE NAME EXACTLY AS YOU SEE IT ON THE SHEET"
""
"ENTER DEPARTMENT ASSIGNMENT: <DEPARTMENT"
MATCH EMP_ID
  ON MATCH REJECT
  ON NOMATCH COMPUTE
    DEPARTMENT = UPCASE(10, DEPARTMENT, 'A10');
  ON NOMATCH INCLUDE
    ON NOMATCH TYPE "DEPARTMENT VALUE CHANGED TO UPPERCASE: <DEPARTMENT"
DATA
END
```

The procedure processes as:

1. The procedure prompts you for an employee ID, last name, first name, and department on a CRTFORM screen. The CRTFORM LOWER option retains the case of entries exactly as typed.

2. You type the following data and press Enter:

```none
ENTER EMPLOYEE'S ID : 444555666
ENTER LAST_NAME: Cutter          FIRST_NAME: Alan
TYPE THE NAME EXACTLY AS YOU SEE IT ON THE SHEET
ENTER DEPARTMENT ASSIGNMENT: sales
```

3. The procedure searches the data source for the ID 444555666. If it does not find the ID, it continues processing the transaction.

4. UPCASE converts the DEPARTMENT entry sales to SALES:

```none
ENTER EMPLOYEE'S ID :
ENTER LAST_NAME: Cutter          FIRST_NAME: Alan
TYPE THE NAME EXACTLY AS YOU SEE IT ON THE SHEET
ENTER DEPARTMENT ASSIGNMENT: sales
DEPARTMENT VALUE CHANGED TO UPPERCASE: SALES
```

5. The procedure adds the transaction to the data source.

6. When you exit the procedure with PF3, the transaction message indicates the number of transactions accepted or rejected:

```none
TRANSACTIONS: TOTAL = 1 ACCEPTED= 1 REJECTED= 0
SEGMENTS: INPUT = 1 UPDATED = 0 DELETED = 0
```
XMLDECOD: Decoding XML-Encoded Characters

The XMLDECOD function decodes the following five standard XML-encoded characters when they are encountered in a string:

<table>
<thead>
<tr>
<th>Character Name</th>
<th>Character</th>
<th>XML-Encoded Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ampersand</td>
<td>&amp;</td>
<td>&amp;</td>
</tr>
<tr>
<td>greater than symbol</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>less than symbol</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>double quotation mark</td>
<td>&quot;</td>
<td>&quot;</td>
</tr>
<tr>
<td>single quotation mark (apostrophe)</td>
<td>'</td>
<td>'</td>
</tr>
</tbody>
</table>

Syntax: How to Decode XML-Encoded Characters

XMLDECOD(inlength, source_string, outlength, output)

where:

inlength
  Integer
  Is the length of the field containing the source character string, or a field that contains the length.

source_string
  Alphanumeric
  Is the name of the field containing the source character string or the string enclosed in single quotation marks ('').

outlength
  Integer
  Is the length of the output character string, or a field that contains the length.

output
  Integer
  Is the field that contains the result, or the format of the output value enclosed in single quotation marks.
Decoding XML-Encoded Characters

The file XMLFUNCS is a .csv file that contains some unencoded characters and some XML-encoded characters. The Master File is:

```
FILE = XMLFUNCS, SUFFIX=COM,$
SEGNAME = SEG01, SEGTYPE=S1,$
FIELD=INSTRING, ALIAS=CHARS, USAGE=A30,ACTUAL=A30,$
```

The contents of the file follow:

```
CHARS: & < > ,$
ENCODED: &amp; &gt; ,$
ENCODED: &quot; &apos; ,$
MIXED: &amp; < &gt; ,$
```

XMLDECOD decodes any of the supported XML-encoded characters. Note that some viewers automatically decode the encoded values for display, so the output is produced in a plain text format (FORMAT WP):

```
FILEDEF XMLFUNCS DISK xmlfuncs.csv
DEFINE FILE XMLFUNCS
OUTSTRING/A30=XMLDECOD(30,INSTRING,30,'A30');
END
TABLE FILE XMLFUNCS
PRINT INSTRING OUTSTRING
ON TABLE PCHOLD FORMAT WP
ON TABLE SET PAGE NOPAGE
```

In the output string, XML-encoded characters have been decoded, and characters that were not encoded have been left as they were in the input string:

```
INSTRING                  OUTSTRING
--------                  --------
CHARS: & < >             CHARS: & < >
ENCODED: &amp; &gt;        ENCODED: & >
ENCODED: &quot; &apos;        ENCODED: ' '
MIXED: &amp; < &gt;         MIXED: & < >
```

XMLDECOD decodes XML-encoded characters and stores the output in a string with format A30:

```
XMLDECOD(30, INSTRING, 30, 'A30')
```

For &amp;, the result is &.

For &gt;, the result is >.
XMLENCOD: XML-Encoding Characters

The XMLENCOD function encodes the following five standard characters when they are encountered in a string:

<table>
<thead>
<tr>
<th>Character Name</th>
<th>Character</th>
<th>Encoded Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ampersand</td>
<td>&amp;</td>
<td>&amp;</td>
</tr>
<tr>
<td>greater than symbol</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>less than symbol</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>double quotation mark</td>
<td>&quot;</td>
<td>&quot;</td>
</tr>
<tr>
<td>single quotation mark (apostrophe)</td>
<td>'</td>
<td>'</td>
</tr>
</tbody>
</table>

Syntax: How to XML-Encode Characters

XMLENCOD(inlength, source_string, option, outlength, output)

where:

inlength
Integer
Is the length of the field containing the source character string, or a field that contains the length.

source_string
Alphanumeric
Is the name of the field containing the source character string or a string enclosed in single quotation marks (').

option
Integer
Is a code that specifies whether to process a string that already contains XML-encoded characters. Valid values are:

- 0, the default, which cancels processing of a string that already contains at least one XML-encoded character.
- 1, which processes a string that contains XML-encoded characters.
**outlength**
Integer

Is the length of the output character string, or a field that contains the length.

**Note:** The output length, in the worst case, could be six times the length of the input.

**output**
Integer

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

**Example:** XML-Encoding Characters

The file XMLFUNCS is a .csv file that contains some unencoded characters and some XML-encoded characters. The Master File is:

```
FILE = XMLFUNCS, SUFFIX=COM,$
SEGNAME = SEG01, SEGTYPE=S1,$
FIELD=INSTRING, ALIAS=CHARS, USAGE=A30,ACTUAL=A30,$
```

The contents of the file follow:

```
CHARS: & < > ,$
ENCODED: &amp; &gt; ,$
ENCODED: &quot; &apos; ,$
MIXED: &amp; &lt; &gt; ,$
```
XMLENCOD XML-encodes any of the supported characters to produce OUTSTRING1, and processes every input string regardless of whether it already contains XML-encoded characters. For OUTSTRING2, it only encodes those strings that do not contain any XML-encoded characters. Note that some viewers automatically decode the encoded values for display, so the output is produced in plain text format (FORMAT WP):

FILEDEF XMLFUNCS DISK xmlfuncs.csv
DEFINE FILE XMLFUNCS
OUTSTRING1/A30=XMLENCOD(30,INSTRING,1,30,'A30');
OUTSTRING2/A30=XMLENCOD(30,INSTRING,0,30,'A30');
END
TABLE FILE XMLFUNCS
PRINT INSTRING OUTSTRING1 IN 24 OUTSTRING2 IN 48
ON TABLE SET PAGE NOPAGE
ON TABLE PCHOLD FORMAT WP
END

In OUTSTRING1, the supported characters have been XML-encoded, and output is produced even if the input string contains encoded characters. OUTSTRING2 is only produced when no XML-encoded characters exist in the input string:

<table>
<thead>
<tr>
<th>INSTRING</th>
<th>OUTSTRING1</th>
<th>OUTSTRING2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARS: &amp; &lt; &gt;</td>
<td>CHAR: &amp; &lt; &gt;</td>
<td>CHAR: &amp; &lt; &gt;</td>
</tr>
<tr>
<td>ENCODED: &amp; &gt;</td>
<td>ENCODED: &amp; &gt;</td>
<td>ENCODED: &quot; '</td>
</tr>
<tr>
<td>MIXED: &amp; &lt; &gt;</td>
<td>MIXED: &amp; &lt; &gt;</td>
<td>MIXED: &quot; '</td>
</tr>
</tbody>
</table>

XMLENCOD XML-encodes characters and stores the output in a string with format A30:

XMLENCOD(30, INSTRING, 30, 1, 'A30')

For &, the result is &amp;.
For >, the result is &gt;.
Chapter 5

Variable Length Character Functions

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

In this chapter:

- Overview
- LENV: Returning the Length of an Alphanumeric Field
- LOCASV: Creating a Variable Length Lowercase String
- POSITV: Finding the Beginning of a Variable Length Substring
- SUBSTV: Extracting a Variable Length Substring
- TRIMV: Removing Characters From a String
- UPCASV: Creating a Variable Length Uppercase String

Overview

For relational data sources, AnV 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 or XFOCUS data source, AnV 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, AnV format is not recommended for use with non-relational data sources.

AnV fields can be used as arguments to all Information Builders-supplied functions that expect alphanumerical arguments. An AnV input parameter is treated as an An parameter and is padded with blanks to its declared size (n). If the last parameter specifies an AnV format, the function result is converted to type AnV with actual length set equal to its size.
The functions described in this topic are designed to work specifically with the AnV data type parameters.

**Reference:** Usage Notes for Using an AnV Field in a Function

The following affect the use of an AnV field in a function:

- When using an AnV argument in a function, the input parameter is treated as an An parameter and is padded with blanks to its declared size (n). If the last parameter specifies an AnV format, the function result is converted to type AnV with actual length set equal to its size.

- Many functions require both an alphanumeric string and its length as input arguments. If the supplied string is stored in an AnV field, you still must supply a length argument to satisfy the requirements of the function. However, the length that will be used in the function's calculations is the actual length stored as the first two bytes of the AnV field.

- In general, any input argument can be a field or a literal. In most cases, numeric input arguments are supplied to these functions as literals, and there is no reason not to supply an integer value. However, if the value is not an integer, it is truncated to an integer value regardless of whether it was supplied as a field or a literal.

**LENV: Returning the Length of an Alphanumeric Field**

Available Languages: reporting

LENV returns the actual length of an AnV field or the size of an An field.

**Syntax:** How to Find the Length of an Alphanumeric Field

```plaintext
LENV(source_string, output)
```

where:

- **source_string**
  Alphanumeric of type An or AnV

  Is the source string or field. 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 AnV format, its length, taken from the length-in-bytes of the field, is returned.
**output**

Integer

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

**Example:** Finding the Length of an AnV Field

TRIMV creates an AnV field named TITLEV by removing trailing blanks from the TITLE value. Then LENV returns the actual length of each instance of TITLEV to the ALEN field:

```
TABLE FILE MOVIES
PRINT
COMPUTE TITLEV/A39V = TRIMV('T', TITLE, 39, ' ', 1, TITLEV);
      ALEN/I2 = LENV(TITLEV, ALEN);
BY CATEGORY NOPRINT
WHERE CATEGORY EQ 'CHILDREN'
END
```

The output is:

<table>
<thead>
<tr>
<th>TITLEV</th>
<th>ALEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMURFS, THE</td>
<td>11</td>
</tr>
<tr>
<td>SHAGGY DOG, THE</td>
<td>15</td>
</tr>
<tr>
<td>SCOOBY-DOO--A DOG IN THE RUFF</td>
<td>28</td>
</tr>
<tr>
<td>ALICE IN WONDERLAND</td>
<td>19</td>
</tr>
<tr>
<td>SESAME STREET--BEDTIME STORIES AND SONGS</td>
<td>39</td>
</tr>
<tr>
<td>ROMPER ROOM--ASK MISS MOLLY</td>
<td>26</td>
</tr>
<tr>
<td>SLEEPING BEAUTY</td>
<td>15</td>
</tr>
<tr>
<td>BAMBI</td>
<td>5</td>
</tr>
</tbody>
</table>

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**

Available Languages: reporting

The LOCASV function converts alphabetic characters in the source string to lowercase and is similar to LOCASE. LOCASV returns AnV output whose actual length is the lesser of the actual length of the AnV 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)

where:

- **upper_limit**
  - Integer
  - Is the limit for the length of the source string.

- **source_string**
  - Alphanumeric of type An or AnV
  - Is the string to be converted to lowercase in single quotation marks, or a field or variable that contains the string. If it is a field, it can have An or AnV format. If it is a field of type AnV, its length is taken from the length in bytes stored in the field. If `upper_limit` is smaller than the actual length, the source string is truncated to this upper limit.

- **output**
  - Alphanumeric of type An or AnV
  - Is the name of the field in which to store the result, or the format of the output value enclosed in single quotation marks ('). This value can be for a field that is AnV or An format.
  - If the output format is AnV, 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

In this example, LOCASV converts the LAST_NAME field to lowercase and specifies a length limit of five characters. The results are stored in the LOWCV_NAME field:

```
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND COMPUTE
LOWCV_NAME/A15V = LOCASV(5, LAST_NAME, LOWCV_NAME);
WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>LOWCV_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>smith</td>
</tr>
<tr>
<td>JONES</td>
<td>jones</td>
</tr>
<tr>
<td>MCCOY</td>
<td>mccoy</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>black</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>green</td>
</tr>
<tr>
<td>CROSS</td>
<td>cross</td>
</tr>
</tbody>
</table>
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**

Available Languages: reporting

The POSITV function finds the starting position of a substring within a larger string. For example, the starting position of the substring DUCT in the string PRODUCTION is 4. If the substring is not in the parent string, the function returns the value 0. This is similar to POSIT; however, the lengths of its AnV 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)

where:

- **source_string**
  Alphanumeric of type An or AnV
  Is the source string that contains the substring whose position you want to find. It can be the string enclosed in single quotation marks ('), or a field or variable that contains the source string. If it is a field of AnV format, its length is taken from the length bytes stored in the field. If **upper_limit** is smaller than the actual 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 An or AnV
  Is the substring whose position you want to find. This can be the substring enclosed in single quotation marks ('), or the field that contains the string. If it is a field, it can have An or AnV format. If it is a field of type AnV, its length is taken from the length bytes stored in the field. If **sub_limit** is smaller than the actual length, the source string is truncated to this limit.
**Example:** Finding the Starting Position of a Variable Length Pattern

POSITV finds the starting position of a trailing definite or indefinite article in a movie title (such as ", THE" in SMURFS, THE). First TRIMV removes the trailing blanks from the title so that the article will be the trailing pattern:

```
DEFINE FILE MOVIES
TITLEV/A39V = TRIMV('T', TITLE, 39, ' ', 1, TITLEV);
PSTART/I4 = POSITV(TITLEV, LENV(TITLEV,'I4'), ',', 1,'I4');
PLEN/I4 = IF PSTART NE 0 THEN LENV(TITLEV,'I4') - PSTART +1 ELSE 0;
END
```

```
TABLE FILE MOVIES
PRINT TITLE
   PSTART AS 'Pattern,Start' IN 25
   PLEN AS 'Pattern,Length'
BY CATEGORY NOPRINT
WHERE PLEN NE 0
END
```

The output is:

<table>
<thead>
<tr>
<th>Title</th>
<th>Pattern Start</th>
<th>Pattern Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMURFS, THE</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>SHAGGY DOG, THE</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>MALTESE FALCON, THE</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>PHILADELPHIA STORY, THE</td>
<td>19</td>
<td>5</td>
</tr>
<tr>
<td>TIN DRUM, THE</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>FAMILY, THE</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>CHORUS LINE, A</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>MORNING AFTER, THE</td>
<td>14</td>
<td>5</td>
</tr>
<tr>
<td>BIRDS, THE</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>BOY AND HIS DOG, A</td>
<td>16</td>
<td>3</td>
</tr>
</tbody>
</table>
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**

Available Languages: reporting

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 AnV field, is determined based on the substring length.

**Syntax:**

How to Extract a Variable Length Substring

\[
SUBSTV(upper_limit, source_string, start, sub_limit, output)
\]

where:

- **upper_limit**
  Integer
  Is the limit for the length of the source string.

- **source_string**
  Alphanumeric of type An or AnV
  Is the character string that contains the substring you want to extract. It can be the string
  enclosed in single quotation marks ('), or the field containing the string. If it is a field, it
  can have An or AnV format. If it is a field of type AnV, its length is taken from the length
  bytes stored in the field. If upper_limit is smaller than the actual 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 the description of the output parameter for related
  information).

- **start**
  Integer
  Is the starting position of the substring in the source string. The starting position can
  exceed the source string length, which results in spaces being returned.
**sub_limit**

Integer

Is the length, in characters, of the substring. Note that the ending position can exceed the input string length depending on the provided values for `start` and `sub_limit`.

**output**

Alphanumeric of type An or AnV

Is the field to which the result is returned, or the format of the output value enclosed in single quotation marks (''). This field can be in An or AnV format.

If the format of `output` is AnV, and assuming `end` is the ending position of the substring, the actual length, `outlen`, is computed as follows from the values for `end`, `start`, and `p_length` (see the `source_string` parameter for related information):

If `end > p_length` or `end < start`, then `outlen = 0`. Otherwise, `outlen = end - start + 1`.

**Example:** Extracting a Variable Length Substring

The following request extracts a trailing definite or indefinite article from a movie title (such as "", THE" in "SMURFS, THE"). First it trims the trailing blanks so that the article is the trailing pattern. Next it finds the starting position and length of the pattern. Then `SUBSTV` extracts the pattern and `TRIMV` trims the pattern from the title:

```
DEFINE FILE MOVIES
  TITLEV/A39V = TRIMV('T', TITLE, 39, ' ', 1, TITLEV);
  PSTART/I4 = POSITV(TITLEV, LENV(TITLEV, 'I4'), ',', 1, 'I4');
  PLEN/I4 = IF PSTART NE 0 THEN LENV(TITLEV, 'I4') - PSTART + 1
               ELSE 0;
  PATTERN/A20V = SUBSTV(39, TITLEV, PSTART, PLEN, PATTERN);
  NEWTIT/A39V = TRIMV('T', TITLEV, 39, PATTERN, LENV(PATTERN, 'I4'), NEWTIT);
END
TABLE FILE MOVIES
  PRINT TITLE
    PSTART AS 'Pattern,Start' IN 25
    PLEN AS 'Pattern,Length'
    NEWTIT AS 'Trimmed,Title' IN 55
  BY CATEGORY NOPRINT
WHERE PLEN NE 0
END
```
The output is:

<table>
<thead>
<tr>
<th>TITLE</th>
<th>Pattern Start</th>
<th>Pattern Length</th>
<th>Trimmed Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMURFS, THE</td>
<td>7</td>
<td>5</td>
<td>SMURFS</td>
</tr>
<tr>
<td>SHAGGY DOG, THE</td>
<td>11</td>
<td>5</td>
<td>SHAGGY DOG</td>
</tr>
<tr>
<td>MALTESE FALCON, THE</td>
<td>15</td>
<td>5</td>
<td>MALTESE FALCON</td>
</tr>
<tr>
<td>PHILADELPHIA STORY, THE</td>
<td>19</td>
<td>5</td>
<td>PHILADELPHIA STORY</td>
</tr>
<tr>
<td>TIN DRUM, THE</td>
<td>9</td>
<td>5</td>
<td>TIN DRUM</td>
</tr>
<tr>
<td>FAMILY, THE</td>
<td>7</td>
<td>5</td>
<td>FAMILY</td>
</tr>
<tr>
<td>CHORUS LINE, A</td>
<td>12</td>
<td>3</td>
<td>CHORUS LINE</td>
</tr>
<tr>
<td>MORNING AFTER, THE</td>
<td>14</td>
<td>5</td>
<td>MORNING AFTER</td>
</tr>
<tr>
<td>BIRDS, THE</td>
<td>6</td>
<td>5</td>
<td>BIRDS</td>
</tr>
<tr>
<td>BOY AND HIS DOG, A</td>
<td>16</td>
<td>3</td>
<td>BOY AND HIS DOG</td>
</tr>
</tbody>
</table>

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

\[ \text{SUBSTV(39, TITLEV, 1, 3, 'A20V')} \]

For SMURFS, the result is SMU.
For SHAGGY DOG, the result is SHA.

**TRIMV: Removing Characters From a String**

Available Languages: reporting

The TRIMV function removes leading and/or trailing occurrences of a pattern within a character string. TRIMV is similar to TRIM. However, TRIMV allows the source string and the pattern to be removed to have AnV format.

TRIMV is useful for converting an An field to an AnV 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

```plaintext
TRIMV(trim_where, source_string, upper_limit, pattern, pattern_limit, output)
```

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 An or AnV
  - Is the source string to be trimmed. It can be the string enclosed in single quotation marks ('), or the field containing the string. If it is a field, it can have An or AnV format. If it is a field of type AnV, its length is taken from the length in bytes stored in the field. If **upper_limit** is smaller than the actual length, the source string is truncated to this upper limit.

- **upper_limit**
  - Integer
  - Is the upper limit for the length of the source string.

- **pattern**
  - Alphanumeric of type An or AnV
  - Is the pattern to remove from the string, enclosed in single quotation marks ('). If it is a field, it can have An or AnV format. If it is a field of type AnV, its length is taken from the length in bytes stored in the field. If **pattern_limit** is smaller than the actual length, the pattern is truncated to this limit.

- **pattern_limit**
  - Integer
  - Is the limit for the length of the pattern.

- **output**
  - Alphanumeric of type An or AnV
  - Is the field to which the result is returned, or the format of the output value enclosed in single quotation marks ('). The field can be in AnV or An format.
If the output format is AnV, the length is set to the number of characters left after trimming.

**Example:** Creating an AnV Field by Removing Trailing Blanks

TRIMV creates an AnV field named TITLEV by removing trailing blanks from the TITLE value:

```
TABLE FILE MOVIES
PRINT DIRECTOR
COMPUTE TITLEV/A39V = TRIMV('T', TITLE, 39, ' ', 1, TITLEV);
BY CATEGORY
END
```

Here are the first 10 lines of the output:

<table>
<thead>
<tr>
<th>CATEGORY</th>
<th>DIRECTOR</th>
<th>TITLEV</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION</td>
<td>SPIELBERG S.</td>
<td>JAWS</td>
</tr>
<tr>
<td>VERHOVEN P.</td>
<td>ROBOCOP</td>
<td></td>
</tr>
<tr>
<td>VERHOVEN P.</td>
<td>TOTAL RECALL</td>
<td></td>
</tr>
<tr>
<td>SCOTT T.</td>
<td>TOP GUN</td>
<td></td>
</tr>
<tr>
<td>MCDONALD P.</td>
<td>RAMBO III</td>
<td></td>
</tr>
<tr>
<td>CHILDREN</td>
<td>SMURFS, THE</td>
<td></td>
</tr>
<tr>
<td>BARTON C.</td>
<td>SHAGGY DOG, THE</td>
<td></td>
</tr>
<tr>
<td>GEROMINI</td>
<td>ALICE IN WONDERLAND</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SESAME STREET—BEDTIME STORIES AND SONGS</td>
<td></td>
</tr>
</tbody>
</table>

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**

Available Languages: reporting

UPCASV converts alphabetic characters to uppercase, and is similar to UPCASE. However, UPCASV can return AnV output whose actual length is the lesser of the actual length of the AnV 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)

where:

*upper_limit*

Integer

Is the limit for the length of the source string. It can be a positive constant or a field whose integer portion represents the upper limit.

*source_string*

Alphanumeric of type An or AnV

is the string to convert to uppercase. It can be the character string enclosed in single quotation marks ("), or the field containing the character string. If it is a field, it can have An or AnV format. If it is a field of type AnV, its length is taken from the length in bytes stored in the field. If *upper_limit* is smaller than the actual length, the source string is truncated to the upper limit.

*output*

Alphanumeric of type An or AnV

Is the field to which the result is returned, or the format of the output value enclosed in single quotation marks ("'). This can be a field with AnV or An format.

If the output format is AnV, the length returned is equal to the smaller of the source string length and *upper_limit*.

**Example:**

Creating a Variable Length Uppercase String

Suppose you are sorting on a field that contains both uppercase and mixed-case values. The following request defines a field called LAST_NAME_MIXED that contains both uppercase and mixed-case values:

```plaintext
DEFINE FILE EMPLOYEE
LAST_NAME_MIXED/A15=IF DEPARTMENT EQ 'MIS' THEN LAST_NAME ELSE LCWORD(15, LAST_NAME, 'A15');
LAST_NAME_UPCASV/A15V=UPCASV(5, LAST_NAME_MIXED, 'A15');
END
```
Suppose you execute a request that sorts by this field:

```
TABLE FILE EMPLOYEE
PRINT LAST_NAME_MIXED AND FIRST_NAME BY LAST_NAME_UPCASV
WHERE CURR_JOBCODE EQ 'B02' OR 'A17' OR 'B04';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME_UPCASV</th>
<th>LAST_NAME_MIXED</th>
<th>FIRST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNI</td>
<td>Banning</td>
<td>JOHN</td>
</tr>
<tr>
<td>BLACK</td>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
</tr>
<tr>
<td>CROSS</td>
<td>CROSS</td>
<td>BARBARA</td>
</tr>
<tr>
<td>MCCOY</td>
<td>MCCOY</td>
<td>JOHN</td>
</tr>
<tr>
<td>MCKNI</td>
<td>Mcknight</td>
<td>ROGER</td>
</tr>
<tr>
<td>ROMAN</td>
<td>Romans</td>
<td>ANTHONY</td>
</tr>
</tbody>
</table>

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.
The functions in this topic manipulate strings of DBCS and SBCS characters when your configuration uses a DBCS code page.

In this chapter:
- DCTRAN: Translating A Single-Byte or Double-Byte Character to Another
- DEDIT: Extracting or Adding Characters
- DSTRIP: Removing a Single-Byte or Double-Byte Character From a String
- DSUBSTR: Extracting a Substring
- JPTRANS: Converting Japanese Specific Characters
- KKFCUT: Truncating a String
- SFTDEL: Deleting the Shift Code From DBCS Data
- SFTINS: Inserting the Shift Code Into DBCS Data

**DCTRAN: Translating 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.

The DCTRAN function can translate single-byte to double-byte characters and double-byte to single-byte characters, as well as single-byte to single-byte characters and double-byte to double-byte characters.

**Syntax:**

`DCTRAN(length, source_string, indecimal, outdecimal, output)`

where:

- `length`  
  Double
Is the number of characters in `source_string`.

`source_string`

Alphanumeric

Is the character string to be translated.

`indecimal`

Double

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

`outdecimal`

Double

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

`output`

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, 'ＡＰＡ本B語', 177, 70, A8)
```

For ＡＰＡ本B語, the result is ＡＰＡ本B語.

**DEDIT: Extracting or Adding Characters**

If your configuration uses a DBCS code page, you can use the DEDIT function to extract characters from 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, output)

where:

\textit{inlength}
   
   Integer
   
   Is the number of \textit{bytes} in \textit{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.

\textit{source_string}
   
   Alphanumeric
   
   Is the string to edit enclosed in single quotation marks ('), or the field containing the string.

\textit{mask_length}
   
   Integer
   
   Is the number of \textit{characters} in mask.

\textit{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.

\textit{output}
   
   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, 'あいうえお', 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, 'あいうえお', 16, '9$9$9$9$9$-ABCDE', 'A20')
The result is aieo-ABCDE.
```

DSTRIP: Removing 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)
```

where:

length

Double

Is the number of characters in source_string and output.

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
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:

\textit{DSTRI\textbackslash{}P(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
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 source string.

Syntax: How to Extract a Substring

\texttt{DSUBSTR(inlength, source_string, start, end, sublength, output)}

where:

\textit{inlength}
Integer

Is the length of the source 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 source string.

\textit{source_string}
Alphanumeric

Is the string from which the substring will be extracted 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 source string. If this argument is less than one or greater than end, the function returns spaces.

dend
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 output. Only sublength characters will be processed.

output
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, 'あいうえお', 4, 6, 3, 'A10')
```

The result is いっ.

**JPTRANS: Converting 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. The following table shows the single component input types:

<table>
<thead>
<tr>
<th>Conversion Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'UPCASE'</td>
<td>Converts Zenkaku (Fullwidth) alphabets to Zenkaku uppercase.</td>
</tr>
<tr>
<td>'LOCASE'</td>
<td>Converts Zenkaku alphabets to Zenkaku lowercase.</td>
</tr>
<tr>
<td>'HNZNALPHA'</td>
<td>Converts alphanumerics from Hankaku (Halfwidth) to Zenkaku.</td>
</tr>
<tr>
<td>'HNZNSIGN'</td>
<td>Converts ASCII symbols from Hankaku to Zenkaku.</td>
</tr>
<tr>
<td>'HNZNKANA'</td>
<td>Converts Katakana from Hankaku to Zenkaku.</td>
</tr>
<tr>
<td>'HNZNSPACE'</td>
<td>Converts space (blank) from Hankaku to Zenkaku.</td>
</tr>
<tr>
<td>'ZNHNALPHA'</td>
<td>Converts alphanumerics from Zenkaku to Hankaku.</td>
</tr>
<tr>
<td>'ZNHNSIGN'</td>
<td>Converts ASCII symbols from Zenkaku to Hankaku.</td>
</tr>
<tr>
<td>'ZNHNKANA'</td>
<td>Converts Katakana from Zenkaku to Hankaku.</td>
</tr>
<tr>
<td>'ZNHNSPACE'</td>
<td>Converts space from Zenkaku to Hankaku.</td>
</tr>
<tr>
<td>'HIRAKATA'</td>
<td>Converts Hiragana to Zenkaku Katakana.</td>
</tr>
<tr>
<td>'KATAHIRA'</td>
<td>Converts Zenkaku Katakana to Hiragana.</td>
</tr>
<tr>
<td>'930TO939'</td>
<td>Converts codepage from 930 to 939.</td>
</tr>
<tr>
<td>'939TO930'</td>
<td>Converts codepage from 939 to 930.</td>
</tr>
</tbody>
</table>
JPTRANS: Converting Japanese Specific Characters

- **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

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

For a b c, the result is A B C.

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

For A B C, the result is a b c.

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

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

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

For !@$%„？, the result is !@ $ % „ ？

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

For 「ぁ・ゃ・ょ・ゅ・ょ・」, the result is 「ベースボール。」

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

For アイウ, the result is ア イ ウ

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

For A a B b C c 1 2 3, the result is AaBbCc123.
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). The following EXTRA rule for those special cases is shown below:

  **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 Bracket (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)
Backslash \ (U+FF3C) -> Backslash \ (U+005C)
Halfwidth Left Corner Bracket (U+300C) -> Halfwidth Left Corner Bracket (U+FF62)
Halfwidth Right Corner Bracket (U+300D) -> Halfwidth Right Corner Bracket (U+FF63)
Halfwidth 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 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)

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.

For バンゴウ # 1 2 3 , the result is バンゴウ#123

JPTRANS('ZNHNALPHA', 20, JPTRANS('ZNHNNSIGN', 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.

**KKFCUT: Truncating a String**

If your configuration uses a DBCS code page, you can use the KKFCUT function to truncate a string.

**Syntax:** How to Truncate a String

```
KKFCUT(length, source_string, output)
```

where:

- **length**
  
  Integer

  Is the length of the source 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 source string.

- **source_string**

  Alphanumeric

  Is the string that will be truncated enclosed in single quotation marks ('), or the field containing the string.
output

Alphanumeric

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

The string will be truncated to the number of bytes in the output field.

Example: Truncating a String

In the following, KKFCUT truncates the COUNTRY field (up to 10 bytes long) to A4 format:

\[ \text{COUNTRY}\_\text{CUT}/\text{A4} = \text{KKFCUT}(10, \text{COUNTRY}, '\text{A4}'); \]

The output in ASCII environments is shown in the following image:

<table>
<thead>
<tr>
<th>国名</th>
<th>COUNTRY_CUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>イギリス</td>
<td>イギ</td>
</tr>
<tr>
<td>日本</td>
<td>日本</td>
</tr>
<tr>
<td>イタリア</td>
<td>イタ</td>
</tr>
<tr>
<td>ドイツ</td>
<td>ドイ</td>
</tr>
<tr>
<td>フランス</td>
<td>フラ</td>
</tr>
</tbody>
</table>

The output in EBCDIC environments is shown in the following image:

<table>
<thead>
<tr>
<th>国名</th>
<th>COUNTRY_CUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>イギリス</td>
<td>イ</td>
</tr>
<tr>
<td>日本</td>
<td>日</td>
</tr>
<tr>
<td>イタリア</td>
<td>イ</td>
</tr>
<tr>
<td>ドイツ</td>
<td>ド</td>
</tr>
<tr>
<td>フランス</td>
<td>フ</td>
</tr>
</tbody>
</table>

SFTDEL: Deleting the Shift Code From DBCS Data

If your configuration uses a DBCS code page, you can use the SFTDEL function to delete the shift code from DBCS data.
**Syntax:**

How to Delete the Shift Code From DBCS Data

\[ \text{SFTDEL}(\text{source\_string}, \text{length}, \text{output}) \]

where:

**source\_string**

Alphanumeric

Is the string from which the shift code will be deleted enclosed in single quotation marks ("'"), or the field containing the string.

**length**

Integer

Is the length of the source 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 source string.

**output**

Alphanumeric

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

**Example:**

Deleting the Shift Code From a String

In the following, SFTDEL deleted the shift code from the COUNTRY field (up to 10 bytes long):

\[ \text{COUNTRY\_DEL/A10 = SFTDEL(COUNTRY, 10, 'A10');} \]

The output in ASCII environments is shown in the following image:

<table>
<thead>
<tr>
<th>国名</th>
<th>COUNTRY_DEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>イギリス</td>
<td>イギリス</td>
</tr>
<tr>
<td>日本</td>
<td>日本</td>
</tr>
<tr>
<td>イタリア</td>
<td>イタリア</td>
</tr>
<tr>
<td>ドイツ</td>
<td>ドイツ</td>
</tr>
<tr>
<td>フランス</td>
<td>フランス</td>
</tr>
</tbody>
</table>
The output in EBCDIC environments is shown in the following image:

<table>
<thead>
<tr>
<th>国名</th>
<th>COUNTRY_DEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>イギリス</td>
<td>ệnhzh</td>
</tr>
<tr>
<td>日本</td>
<td>ăt</td>
</tr>
<tr>
<td>イタリア</td>
<td>ătj</td>
</tr>
<tr>
<td>ドイツ</td>
<td>ăt</td>
</tr>
<tr>
<td>フランス</td>
<td>ăt</td>
</tr>
</tbody>
</table>

SFTINS: Inserting the Shift Code Into DBCS Data

If your configuration uses a DBCS code page, you can use the SFTINS function to insert the shift code into DBCS data.

**Syntax:**

**How to Insert the Shift Code Into DBCS Data**

\[
SFTINS(source\_string, \, length, \, output)
\]

where:

- **source\_string**
  - Alphanumeric
  - Is the string into which the shift code will be inserted enclosed in single quotation marks ('), or the field containing the string.

- **length**
  - Integer
  - Is the length of the source 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 source string.

- **output**
  - Alphanumeric
  - Is the field to which the result is returned, or the format of the output value enclosed in single quotation marks (').

**Example:**

**SFTINS: Inserting the Shift Code Into a String**

In the following example, SFTINS inserts the shift code into the COUNTRY_DEL field (which is the COUNTRY field with the shift code deleted):

\[
COUNTRY\_INS/A10 = SFTINS(COUNTRY\_DEL, 10, 'A10');
\]
The output displays the original COUNTRY field, the COUNTRY_DEL field with the shift code deleted, and the COUNTRY_INS field with the shift code re-inserted.

The output in ASCII environments, is shown in the following image:

<table>
<thead>
<tr>
<th>国名</th>
<th>COUNTRY_DEL</th>
<th>COUNTRY_INS</th>
</tr>
</thead>
<tbody>
<tr>
<td>イギリス</td>
<td>イギリス</td>
<td>イギリス</td>
</tr>
<tr>
<td>日本</td>
<td>日本</td>
<td>日本</td>
</tr>
<tr>
<td>イタリア</td>
<td>イタリア</td>
<td>イタリア</td>
</tr>
<tr>
<td>ドイツ</td>
<td>ドイツ</td>
<td>ドイツ</td>
</tr>
<tr>
<td>フランス</td>
<td>フランス</td>
<td>フランス</td>
</tr>
</tbody>
</table>

The output in EBCDIC environments is shown in the following image:

<table>
<thead>
<tr>
<th>国名</th>
<th>COUNTRY_DEL</th>
<th>COUNTRY_INS</th>
</tr>
</thead>
<tbody>
<tr>
<td>イギリス</td>
<td>b'AFib'</td>
<td>イギリス</td>
</tr>
<tr>
<td>日本</td>
<td>'iA'</td>
<td>日本</td>
</tr>
<tr>
<td>イタリア</td>
<td>b'jfiA'</td>
<td>イタリア</td>
</tr>
<tr>
<td>ドイツ</td>
<td>'ibb'</td>
<td>ドイツ</td>
</tr>
<tr>
<td>フランス</td>
<td>'h[1]'</td>
<td>フランス</td>
</tr>
</tbody>
</table>
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.

The result of a data source function must be stored in a field. The result cannot be stored in a Dialogue Manager variable.

For many functions, the output argument can be supplied either as a field name or as a format enclosed in single quotation marks (‘). However, if a function is called from a Dialogue Manager command, this argument must always be supplied as a format. If a function is called from a Maintain Data procedure, this argument must always be supplied as a field name. For detailed information about calling a function and supplying arguments, see Accessing and Calling a Function.

In this chapter:

- **CHECKMD5**: Computing an MD5 Hash Check Value
- **CHECKSUM**: Computing a Hash Sum
- **COALESCE**: Returning the First Non-Missing Value
- **DB_EXPR**: Inserting an SQL Expression Into a Request
- **DB_INFILE**: Testing Values Against a File or an SQL Subquery
- **DB_LOOKUP**: Retrieving Data Source Values
- **DECODE**: Decoding Values
- **FIND**: Verifying the Existence of a Value in a Data Source
- **IMPUTE**: Replacing Missing Values With Aggregated Values
- **LAST**: Retrieving the Preceding Value
- **LOOKUP**: Retrieving a Value From a Cross-referenced Data Source
- **NULLIF**: Returning a Null Value When Parameters Are Equal
CHECKMD5: Computing an MD5 Hash Check Value

CHECKMD5 takes an alphanumeric input value and returns a 128-bit value in a fixed length alphanumeric string, using the MD5 hash function. A hash function is any function that can be used to map data of arbitrary size to data of fixed size. The values returned by a hash function are called hash values. They can be used for assuring the integrity of transmitted data.

**Syntax:** How to Compute an MD5 Hash Check Value

```plaintext
CHECKMD5(buffer)
```

where:

`buffer`

Is a data buffer whose hash value is to be calculated. It can be a set of data of different types presented as a single field, or a group field in one of the following data type formats: An, AnV, or TXn.

**Example:** Calculating an MD5 Hash Check Value

The following request calculates an MD5 hash check value and converts it to an alphanumeric hexadecimal value for display.

```plaintext
DEFINE FILE WF_RETAIL_LITE
MD5/A32 = HEXTYPE(CHECKMD5(PRODUCT_CATEGORY));
END
TABLE FILE WF_RETAIL_LITE
SUM MD5
BY PRODUCT_CATEGORY
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
TYPE=REPORT, FONT=COURIER, $
ENDSTYLE
END
```
The output is shown in the following image. The monospaced font shows that although the input values have varying length, the output has a fixed length.

<table>
<thead>
<tr>
<th>Product Category</th>
<th>MD5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessories</td>
<td>98EDB85B00D9527AD5ACEBE451B3FAE6</td>
</tr>
<tr>
<td>Camcorder</td>
<td>612A923BDD05C2231F81991B8D12A3A1</td>
</tr>
<tr>
<td>Computers</td>
<td>45888A4DA062F16A099A7F7C6CC15EE0</td>
</tr>
<tr>
<td>Media Player</td>
<td>D34BEA29F24AF9FDE2E10B3E1D857CF9</td>
</tr>
<tr>
<td>Stereo Systems</td>
<td>3AA9FFE9806E269A7EB066A84092F0A3</td>
</tr>
<tr>
<td>Televisions</td>
<td>A3B5BC99DD2B42627EF64A4FCAAAB0B2</td>
</tr>
<tr>
<td>Video Production</td>
<td>60913E95848330A2C4A5D921E7C8BB14</td>
</tr>
</tbody>
</table>

CHECKMD5 calculates a fixed length MD5 hash check value, and HEXTYPE converts it to a printable hexadecimal string.

HEXTYPE(CHECKMD5(PRODUCT_CATEGORY))

For Accessories, the result is 98EDB85B00D9527AD5ACEBE451B3FAE6.

**CHECKSUM: Computing a Hash Sum**

CHECKSUM computes a hash sum, called the checksum, of its input parameter, as a whole number in format I11. This can be used for equality search of the fields. A checksum is a hash sum used to ensure the integrity of a file after it has been transmitted from one storage device to another.

**Syntax:** How to Compute a CHECKSUM Hash Value

CHECKSUM(buffer)

where:

**buffer**

Is a data buffer whose hash index is to be calculated. It can be a set of data of different types presented as a single field, in one of the following data type formats: An, AnV, or TXn.
Example: Calculating a CHECKSUM Hash Value

The following request computes a checksum hash value.

```
DEFINE FILE WF_RETAIL_LITE
CHKSUM/I11 = (CHECKSUM(PRODUCT_CATEGORY));
END
TABLE FILE WF_RETAIL_LITE
PRINT CHKSUM
BY PRODUCTCATEGORY
WHERE PRODUCTCATEGORY NE LAST PRODUCTCATEGORY
ON TABLE SET PAGE NOLEAD
END
```

The output is shown in the following image.

<table>
<thead>
<tr>
<th>Product Category</th>
<th>CHKSUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessories</td>
<td>-830549649</td>
</tr>
<tr>
<td>Camcorder</td>
<td>-912058982</td>
</tr>
<tr>
<td>Computers</td>
<td>-469201037</td>
</tr>
<tr>
<td>Media Player</td>
<td>-1760917009</td>
</tr>
<tr>
<td>Stereo Systems</td>
<td>-1853215244</td>
</tr>
<tr>
<td>Televisions</td>
<td>810407163</td>
</tr>
<tr>
<td>Video Production</td>
<td>275494446</td>
</tr>
</tbody>
</table>

CHECKSUM calculates a checksum hash value.

```
CHECKSUM(PRODUCTCATEGORY)
```

For Accessories, the result is -830549649.

COALESCE: Returning the First Non-Missing Value

Given a list of arguments, COALESCE returns the value of the first argument that is not missing. If all argument values are missing, it returns a missing value if MISSING is ON. Otherwise it returns a default value (zero or blank).

Syntax: How to Return the First Non-Missing Value

```
COALESCE(arg1, arg2, ...)
```
where:

\[ arg1, \ arg2, \ldots \]

Any field, expression, or constant. The arguments should all be either numeric or alphanumeric.

Are the input parameters that are tested for missing values.

The output data type is the same as the input data types.

**Example: Returning the First Non-Missing Value**

This example uses the SALES data source with missing values added. The missing values are added by the following procedure named SALEMISS:

```
MODIFY FILE SALES
  FIXFORM STORE/4 DATE/5 PROD/4
  FIXFORM UNIT/3 RETAIL/5 DELIVER/3
  FIXFORM OPEN/3 RETURNS/C2 DAMAGED/C2
  MATCH STORE
    ON NOMATCH REJECT
    ON MATCH CONTINUE
  MATCH DATE
    ON NOMATCH REJECT
    ON MATCH CONTINUE
  MATCH PROD_CODE
    ON NOMATCH INCLUDE
    ON MATCH REJECT
  DATA
  14Z 1017 C13 15 1.99 35 30 6
  14Z 1017 C14 18 2.05 30 25 4
  14Z 1017 E2 33 0.99 45 40
  END
```

The following request uses COALESCE to return the first non-missing value:

```
TABLE FILE SALES
PRINT DAMAGED RETURNS RETAIL_PRICE
COMPUTE
  COAL1/D12.2 MISSING ON = COALESCE(DAMAGED, RETURNS, RETAIL_PRICE);
  BY STORE_CODE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```
The output is shown in the following image. The value of DAMAGED is returned, if it is not missing. If DAMAGED is missing, the value of RETURNS is returned, if it is not missing. If they are both missing, the value of RETAIL_PRICE is returned.

<table>
<thead>
<tr>
<th>STORE CODE</th>
<th>DAMAGED</th>
<th>RETURNS</th>
<th>RETAIL PRICE</th>
<th>COAL1</th>
</tr>
</thead>
<tbody>
<tr>
<td>14B</td>
<td>6</td>
<td>10</td>
<td>$.95</td>
<td>6.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>$1.29</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>$1.89</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>3</td>
<td>$1.99</td>
<td>.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>$2.39</td>
<td>4.00</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>$2.19</td>
<td>.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>9</td>
<td>$.99</td>
<td>4.00</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>8</td>
<td>$1.09</td>
<td>9.00</td>
</tr>
<tr>
<td>14Z</td>
<td>3</td>
<td>2</td>
<td>$8.85</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>$1.89</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>$1.99</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>.</td>
<td>$1.99</td>
<td>6.00</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>4</td>
<td>$2.05</td>
<td>4.00</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>$2.09</td>
<td>.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>$2.09</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>4</td>
<td>$.89</td>
<td>7.00</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td>$.99</td>
<td>.99</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>$1.09</td>
<td>2.00</td>
</tr>
<tr>
<td>77F</td>
<td>1</td>
<td>1</td>
<td>$2.09</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>$2.49</td>
<td>.00</td>
</tr>
<tr>
<td>K1</td>
<td>0</td>
<td>1</td>
<td>$1.49</td>
<td>.00</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>$.99</td>
<td>1.00</td>
</tr>
</tbody>
</table>

COALESCE returns the first non-missing value:

COALESCE(DAMAGED, RETURNS)

The following table shows sample inputs and results.

<table>
<thead>
<tr>
<th>DAMAGED</th>
<th>RETURNS</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>MISSING</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
DB_EXPR: Inserting an SQL Expression Into a Request

The DB_EXPR function inserts a native SQL expression exactly as entered into the native SQL generated for a FOCUS or SQL language request.

The DB_EXPR function can be used in a DEFINE command, a DEFINE in a Master File, a WHERE clause, a FILTER FILE command, a filter in a Master File, or in an SQL statement. It can be used in a COMPUTE command if the request is an aggregate request (uses the SUM, WRITE, or ADD command) and has a single display command. The expression must return a single value.

Syntax: How to Insert an SQL Expression Into a Request With DB_EXPR

DB_EXPR(native_SQL_expression)

where:

native_SQL_expression

Is a partial native SQL string that is valid to insert into the SQL generated by the request. The SQL string must have double quotation marks (") around each field reference, unless the function is used in a DEFINE with a WITH phrase.

Reference: Usage Notes for the DB_EXPR Function

- The expression must return a single value.
- Any request that includes one or more DB_EXPR functions must be for a synonym that has a relational SUFFIX.
- Field references in the native SQL expression must be within the current synonym context.
- The native SQL expression must be coded inline. SQL read from a file is not supported.
Example: Inserting the DB2 BIGINT and CHAR Functions Into a TABLE Request

The following TABLE request against the WF_RETAIL data source uses the DB_EXPR function in the COMPUTE command to call two DB2 functions. It calls the BIGINT function to convert the squared revenue to a BIGINT data type, and then uses the CHAR function to convert that value to alphanumeric.

TABLE FILE WF_RETAILLITE
SUM REVENUE NOPRINT
AND COMPUTE BIGREV/A31 = DB_EXPR(CHAR(BIGINT("REVENUE" * "REVENUE") ) ) ;
AS 'Alpha Square Revenue'
BY REGION
ON TABLE SET PAGE NOPAGE
END

WF_RETAIL is a sample data source you can create by right-clicking an application on the Reporting Server Web Console and pointing to New and then clicking Tutorials from the context menu.

The trace shows that the expression from the DB_EXPR function was inserted into the DB2 SELECT statement:

```
SELECT
  T11."REGION",
  SUM(T1."Revenue"),
  ((CHAR(BIGINT( SUM(T1."Revenue") * SUM(T1."Revenue") ) ) ))
FROM
  wrd_fact_sales T1,
  wrd_dim_customer T5,
  wrd_dim_geography T11
WHERE
  (T5."ID_CUSTOMER" = T1."ID_CUSTOMER") AND
  (T11."ID_GEOGRAPHY" = T5."ID_GEOGRAPHY")
GROUP BY
  T11."REGION"
ORDER BY
  T11."REGION"
FOR FETCH ONLY;
END
```
The output is:

<table>
<thead>
<tr>
<th>Region</th>
<th>Alpha Square Revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Central</td>
<td>459024717717929</td>
</tr>
<tr>
<td>MidEast</td>
<td>61720506151994</td>
</tr>
<tr>
<td>NorthEast</td>
<td>247772056471221</td>
</tr>
<tr>
<td>NorthWest</td>
<td>42335175855351</td>
</tr>
<tr>
<td>SouthEast</td>
<td>205820846242532</td>
</tr>
<tr>
<td>SouthWest</td>
<td>9449541537794</td>
</tr>
<tr>
<td>West</td>
<td>164356565757257</td>
</tr>
</tbody>
</table>

**DB_INFILE: Testing Values Against a File or an SQL Subquery**

The DB_INFILE function compares one or more field values in a source file to values in a target file. The comparison can be based on one or more field values. DB_INFILE returns the value 1 (TRUE) if the set of source fields matches a set of values from the target file. Otherwise, the function returns 0 (zero, FALSE). DB_INFILE can be used where a function is valid in a WebFOCUS request, such as in a DEFINE or a WHERE phrase.

The target file can be any data source that WebFOCUS can read. Depending on the data sources accessed and the components in the request, either WebFOCUS or an RDBMS will process the comparison of values.

If WebFOCUS processes the comparison, it reads the target data source and dynamically creates a sequential file containing the target data values, along with a synonym describing the data file. It then builds IF or WHERE structures in memory with all combinations of source and target values. If the target data contains characters that WebFOCUS considers wildcard characters, it will treat them as wildcard characters unless the command SET EQTEST = EXACT is in effect.

The following situations exist when a relational data source is the source file:

- **The target values are in a relational data source from the same RDBMS and connection.**
  
  In this case, the target file referenced by DB_INFILE can be:

  - An SQL file containing a subquery that retrieves the target values. A synonym must exist that describes the target SQL file. The Access File must specify the CONNECTION and DATASET for the target file.
If the subquery results in a SELECT statement supported by the RDBMS, the relational adapter inserts the subquery into the WHERE predicate of the generated SQL.

If the subquery does not result in a valid SELECT statement for the RDBMS, the relational adapter retrieves the target values. It then generates a WHERE predicate, with a list of all combinations of source and target field values.

You can create an SQL file containing a subquery and a corresponding synonym using the HOLD FORMAT SQL_SCRIPT command. For more information, see the Creating Reports manual.

A relational data source. A synonym must exist that describes the target data source.

If the data source contains only those fields referenced by DB_INFILE as target fields, the relational adapter creates a subquery that retrieves the target values. If the subquery results in a SELECT statement supported by the RDBMS, the relational adapter inserts the subquery into the WHERE predicate of the generated SQL.

If the subquery does not result in a valid SELECT statement for the RDBMS, the relational adapter retrieves a unique list of the target values. It then generates a WHERE predicate with a list of all combinations of source and target field values.

The target values are in a non-relational data source or a relational data source from a different RDBMS or connection. In this case, the target values are retrieved and passed to WebFOCUS for processing.

Syntax: How to Compare Source and Target Field Values With DB_INFILE

```
DB_INFILE(target_file, s1, t1, ... sn, tn)
```

where:

- **target_file**
  - Is the synonym for the target file.

- **s1, ..., sn**
  - Are fields from the source file.

- **t1, ..., tn**
  - Are fields from the target file.

The function returns the value 1 if a set of target values matches the set of source values. Otherwise, the function returns a zero (0).
**Reference:** Usage Notes for DB_INFILE

- If both the source and target data sources have MISSING=ON for a comparison field, then a missing value in both files is considered an equality. If MISSING=OFF in one or both files, a missing value in one or both files results in an inequality.

- Values are not padded or truncated when compared, except when comparing date and date-time values.

- If the source field is a date field and the target field is a date-time field, the time component is removed before comparison.

- If the source field is a date-time field and the target field is a date field, a zero time component is added to the target value before comparison.

- If an alphanumeric field is compared to a numeric field, an attempt will be made to convert the alphanumeric value to a number before comparison.

- If WebFOCUS processes the comparison, and the target data contains characters that WebFOCUS considers wildcard characters, it will treat them as wildcard characters unless the command SET EQTEST = EXACT is in effect.

**Example:** Comparing Source and Target Values Using an SQL Subquery File

This example uses the WF_RETAIL DB2 data source.

WF_RETAIL is a sample data source you can create by right-clicking an application on the Reporting Server Web Console, selecting New, and then Samples from the context menu.

The SQL file named retail_subquery.sql contains the following subquery that retrieves specified state codes in the Central and NorthEast regions:

```sql
SELECT MAX(T11.REGION), MAX(T11.STATECODE) FROM wrd_dim_geography T11
WHERE (T11.STATECODE IN('AR', 'IA', 'KS', 'KY', 'WY', 'CT', 'MA', 'NJ', 'NY', 'RI')) AND (T11.REGION IN('Central', 'NorthEast')) GROUP BY T11.REGION, T11.STATECODE
```

The retail_subquery.mas Master File follows:

```Dos
FILENAME=RETAIL_SUBQUERY, SUFFIX=DB2, $ SEGMENT=RETAIL_SUBQUERY, SEGTYPE=S0, $ FIELDNAME=REGION, ALIAS=E01, USAGE=A15V, ACTUAL=A15V, MISSING=ON, $ FIELDNAME=STATECODE, ALIAS=E02, USAGE=A2, ACTUAL=A2, MISSING=ON, $ $ 
```

The retail_subquery.acx Access File follows:

```Dos
SEGNAME=RETAIL_SUBQUERY, CONNECTION=CON1, DATASET=RETAIL_SUBQUERY.SQL, $ ```
**Note:** You can create an SQL subquery file, along with a corresponding synonym, using the HOLD FORMAT SQL_SCRIPT command. For more information, see the *Creating Reports With WebFOCUS Language* manual.

**Note:** You can create an SQL subquery file, along with a corresponding synonym, using the HOLD FORMAT SQL_SCRIPT command. For more information, see the *Creating Reports* manual.

The following request uses the DB_INFILE function to compare region names and state codes against the names retrieved by the subquery:

```
TABLE FILE WF_RETAILLITE
SUM REVENUE
BY REGION
BY STATECODE
WHERE DB_INFILE(RETAIL_SUBQUERY, REGION, REGION, STATECODE, STATECODE)
ON TABLE SET PAGE NOPAGE
END
```

The trace shows that the subquery was inserted into the WHERE predicate in the generated SQL:

```
SELECT
  T11."REGION",
  T11."STATECODE",
  SUM(T1."Revenue")
FROM
  wrd_fact_sales T1,
  wrd_dim_customer T5,
  wrd_dim_geography T11
WHERE
  (T5."ID_CUSTOMER" = T1."ID_CUSTOMER") AND
  (T11."ID_GEOGRAPHY" = T5."ID_GEOGRAPHY") AND
  ((T11."REGION", T11."STATECODE") IN (SELECT MAX(T11.REGION),
    MAX(T11.STATECODE) FROM wrd_dim_geography T11 WHERE
    (T11.STATECODE IN('AR', 'IA', 'KS', 'KY', 'WY', 'CT', 'MA',
      'NJ', 'NY', 'RI')) AND (T11.REGION IN('Central', 'NorthEast'))) GROUP BY T11.REGION, T11.STATECODE)
GROUP BY
  T11."REGION",
  T11."STATECODE"
ORDER BY
  T11."REGION",
  T11."STATECODE"
FOR FETCH ONLY;
END
```
Example: Comparing Source and Target Values Using a Sequential File

The empvalues.ftm sequential file contains the last and first names of employees in the MIS department:

SMITH MARY JONES DIANE MCCOY
JOHN BLACKWOOD ROSEMARIE GREENSPAN MARY
CROSS BARBARA

The empvalues.mas Master File describes the data in the empvalues.ftm file

FILENAME=EMPVALUES, SUFFIX=FIX , IOTYPE=BINARY, $ SEGMENT=EMPVALUE, SEGTYPE=S0, $ FIELDNAME=LN, ALIAS=E01, USAGE=A15, ACTUAL=A16, $ FIELDNAME=FN, ALIAS=E02, USAGE=A10, ACTUAL=A12, $

Note: You can create a sequential file, along with a corresponding synonym, using the HOLD FORMAT SQL_SCRIPT command. For more information, see the Creating Reports With WebFOCUS Language manual.
The following request against the FOCUS EMPLOYEE data source uses the DB_INFILE function to compare employee names against the names stored in the empvalues.ftm file:

```
FILEDEF EMPVALUES DISK baseapp/empvalues.ftm
TABLE FILE EMPLOYEE
SUM CURR_SAL
BY LAST_NAME BY FIRST_NAME
WHERE DB_INFILE(EMPVALUES, LAST_NAME, LN, FIRST_NAME, FN)
ON TABLE SET PAGE NOPAGE
END
```

The output is:

```
<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>CURR_SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>$21,780.00</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>$27,062.00</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>$9,000.00</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>$18,480.00</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>$18,480.00</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>$13,200.00</td>
</tr>
</tbody>
</table>
```

**Syntax:**

**How to Control DB_INFILE Optimization**

To control whether to prevent optimization of the DB_INFILE expression, issue the following command:

```
SET DB_INFILE = {DEFAULT|EXPAND_ALWAYS|EXPAND_NEVER}
```

In a TABLE request, issue the following command:

```
ON TABLE SET DB_INFILE {DEFAULT|EXPAND_ALWAYS|EXPAND_NEVER}
```

where:

- **DEFAULT**

  Enables DB_INFILE to create a subquery if its analysis determines that it is possible. This is the default value.

- **EXPAND_ALWAYS**

  Prevents DB_INFILE from creating a subquery. Instead, it expands the expression into IF and WHERE clauses in memory.
EXPAND_NEVER

Prevents DB_INFILE from expanding the expression into IF and WHERE clauses in memory. Instead, it attempts to create a subquery. If this is not possible, a FOC32585 message is generated and processing halts.

**DB_LOOKUP: Retrieving Data Source Values**

Available Languages: reporting, MODIFY

You can use the DB_LOOKUP function 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*

Is the lookup Master File.

*srcfld1, srcfld2 ...*

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

*lookfld1, lookfld2 ...*

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

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.

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, unless the ENGINE INT SET CACHE=ON command is in effect. Having this setting in effect may also improve performance if the values will be looked up more than once. The key field of the sequential file 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. 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.
Example: Retrieving a Value From a Fixed Format Sequential File in a TABLE Request

The following procedure creates a fixed format sequential file named GSALE from the GGSALES data source. The fields in this file are PRODUCT (product description), CATEGORY (product category), and PCD (product code). The file is sorted on the PCD field:

```
SET ASNAMES = ON
TABLE FILE GGSALES
SUM PRODUCT CATEGORY
BY PCD
ON TABLE HOLD AS GSALE FORMAT ALPHA
END
```

The following Master File is generated as a result of the HOLD command:

```
FILENAME=GSALE, SUFFIX=FIX, $
SEGMENT=GSALE, SEGTYPE=S1, $
    FIELDNAME=PCD, ALIAS=E01, USAGE=A04, ACTUAL=A04, $
    FIELDNAME=PRODUCT, ALIAS=E02, USAGE=A16, ACTUAL=A16, $
    FIELDNAME=CATEGORY, ALIAS=E03, USAGE=A11, ACTUAL=A11, $
```

The following TABLE request against the GGPRODS data source, sorts the report on the field that matches the key field in the lookup file. It retrieves the value of the CATEGORY field from the GSALE lookup file by matching on the product code and product description fields. Note that the DEFINE FILE command is cleared at the end of the request:

```
DEFINE FILE GGPRODS
PCAT/A11 MISSING ON = DB_LOOKUP(GSALE, PRODUCT_ID, PCD, PRODUCT_DESCRIPTION, PRODUCT, CATEGORY);
END
TABLE FILE GGPRODS
PRINT PRODUCT_DESCRIPTION PCAT
BY PRODUCT_ID
END
DEFINE FILE GGPRODS CLEAR
END
```

Because the GSALE Master File does not define the CATEGORY field with the MISSING=ON attribute, the PCAT column displays a blank in those rows that have no matching record in the lookup file:

<table>
<thead>
<tr>
<th>Product Code</th>
<th>Product</th>
<th>PCAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>B141</td>
<td>Hazelnut</td>
<td>----</td>
</tr>
<tr>
<td>B142</td>
<td>French Roast</td>
<td></td>
</tr>
<tr>
<td>B144</td>
<td>Kona</td>
<td></td>
</tr>
</tbody>
</table>
If you add the `MISSING=ON` attribute to the CATEGORY field in the GSALE Master File, the PCAT column displays a missing data symbol in rows that do not have a matching record in the lookup file:

<table>
<thead>
<tr>
<th>Product Code</th>
<th>Product</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>B141</td>
<td>Hazelnut</td>
<td></td>
</tr>
<tr>
<td>B142</td>
<td>French Roast</td>
<td></td>
</tr>
<tr>
<td>B144</td>
<td>Kona</td>
<td></td>
</tr>
<tr>
<td>F101</td>
<td>Scone</td>
<td>Food</td>
</tr>
<tr>
<td>F102</td>
<td>Biscotti</td>
<td>Food</td>
</tr>
<tr>
<td>F103</td>
<td>Croissant</td>
<td>Food</td>
</tr>
<tr>
<td>G100</td>
<td>Mug</td>
<td>Gifts</td>
</tr>
<tr>
<td>G104</td>
<td>Thermos</td>
<td>Gifts</td>
</tr>
<tr>
<td>G110</td>
<td>Coffee Grinder</td>
<td>Gifts</td>
</tr>
<tr>
<td>G121</td>
<td>Coffee Pot</td>
<td>Gifts</td>
</tr>
</tbody>
</table>

**Example:**  
**Retrieving a Value From a LOOKUP Table**

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

\[
\text{DB\_LOOKUP(dmcomp,STORE\_CODE,STORE\_CODE,STORENAME)}
\]

For 1003CA the result is Audio Expert.

For 1004MD the result is City Video For 2010AZ the result is eMart.

**DECODE: Decoding Values**

Available Languages: reporting, Maintain

The DECODE function assigns values based on the coded value of an input field. DECODE is useful for giving a more meaningful value to a coded value in a field. 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 females). DECODE expands (decodes) these values to ensure correct interpretation on a report.
You can use DECODE by supplying values directly in the function or by reading values from a separate file.

The use of DECODE with Maintain is limited. For information on decoding values with subscripted stack values, see SELECTS: Decoding a Value From a Stack.

**Syntax:**

**How to Supply Values in the Function**

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

where:

*fieldname*

  Alphanumeric or Numeric
  
  Is the name of the input field.

*code*

  Alphanumeric or Numeric
  
  Is the coded value that DECODE compares with the current value of fieldname. 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 field name, the code and field name must be in the same format.

*result*

  Alphanumeric or Numeric
  
  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 data type of the expression.

*default*

  Alphanumeric or Numeric
  
  Is the value returned as a result for non-matching codes. The format must be the same as the format of result. If you omit a default value, DECODE assigns a blank or zero to non-matching codes.

*filename*

  Alphanumeric
Is the ddname that points to name of the file in which code/result pairs are stored. Every record in the file must contain a 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 phrase. Use either a comma or blank to separate the code from the result, or one pair from another.

Note: DECODE has no output argument.

Example: Supplying Values Using the DECODE Function

EDIT extracts the first character of the CURR_JOBCODE field, then DECODE returns either ADMINISTRATIVE or DATA PROCESSING depending on the value extracted.

```
TABLE FILE EMPLOYEE
PRINT CURR_JOBCODE AND COMPUTE
DEPX_CODE/A1 = EDIT(CURR_JOBCODE, '9$$'); NOPRINT AND COMPUTE
JOB_CATEGORY/A15 = DECODE DEPX_CODE(A 'ADMINISTRATIVE'
B 'DATA PROCESSING');
BY LAST_NAME
WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

```
LAST_NAME    CURR_JOBCODE    JOB_CATEGORY
---------    ------------    ------------
BLACKWOOD    B04             DATA PROCESSING
CROSS        A17             ADMINISTRATIVE
GREENSPAN    A07             ADMINISTRATIVE
JONES        B03             DATA PROCESSING
MCCOY        B02             DATA PROCESSING
SMITH        B14             DATA PROCESSING
```

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.

Reference: Guidelines for Reading Values From a File

- Each record in the file is expected to contain pairs of elements separated by a comma or blank.

- If each record in the file consists of only one element, this element is interpreted as the code, and the result becomes either a blank or zero, as needed.
This makes it possible to use the file to hold screening literals referenced in the screening condition:

\[
\text{IF field IS (filename)}
\]

and as a file of literals for an IF criteria specified in a computational expression. For example:

\[
\begin{align*}
\text{TAKE} & = \text{DECODE SELECT (filename ELSE 1)}; \\
\text{VALUE} & = \text{IF TAKE IS 0 THEN... ELSE...;}
\end{align*}
\]

TAKE is 0 for SELECT values found in the literal file and 1 in all other cases. The VALUE computation is carried out as if the expression had been:

\[
\text{IF SELECT (filename) THEN... ELSE...;}
\]

- The file can contain up to 32,767 characters in the file.
- All data is interpreted in ASCII format on UNIX and Windows, or in EBCDIC format on z/OS, and converted to the USAGE format of the DECODE pairs.
- Leading and trailing blanks are ignored.
- The remainder of each record is ignored and can be used for comments or other data. This convention applies in all cases, except when the file name is HOLD. In that case, the file is presumed to have been created by the HOLD command, which writes fields in the internal format, and the DECODE pairs are interpreted accordingly. In this case, extraneous data in the record is ignored.

**Example:** Reading DECODE Values From a File

The following example has two parts. The first part creates a file with a list of IDs and reads the EDUCFILE data source. The second part reads the EMPLOYEE data source and assigns 0 to those employees who have taken classes and 1 to those employees who have not. The HOLD file contains only one column of values. Therefore, DECODE assigns the value 0 to an employee whose EMP_ID appears in the file and 1 when EMP_ID does not appear in the file.

```
TABLE FILE EDUCFILE
PRINT EMP_ID
ON TABLE HOLD
END

TABLE FILE EMPLOYEE
PRINT EMP_ID AND LAST_NAME AND FIRST_NAME AND COMPUTE
NOT_IN_LIST/I1 = DECODE EMP_ID(HOLD ELSE 1);
WHERE DEPARTMENT EQ 'MIS';
END
```
The output is:

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>NOT_IN_LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>112847612</td>
<td>SMITH</td>
<td>MARY</td>
<td>0</td>
</tr>
<tr>
<td>117593129</td>
<td>JONES</td>
<td>DIANE</td>
<td>0</td>
</tr>
<tr>
<td>219984371</td>
<td>MCCOY</td>
<td>JOHN</td>
<td>1</td>
</tr>
<tr>
<td>326179357</td>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>0</td>
</tr>
<tr>
<td>543729165</td>
<td>GREENSPAN</td>
<td>MARY</td>
<td>0</td>
</tr>
<tr>
<td>818692173</td>
<td>CROSS</td>
<td>BARBARA</td>
<td>0</td>
</tr>
</tbody>
</table>

**FIND: Verifying the Existence of a Value in a Data Source**

Available Languages: MODIFY, Maintain

The FIND function determines if a data value is in a data source field being searched. The function sets a temporary field to 1 (a non-zero value for MODIFY) if the data value is found in the data source field, and to 0 if it is not. FIND does not change the searched file's current database position. A value greater than zero confirms the presence of the data value, not the number of instances in the data source 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.

**Note:** For MODIFY only, the FIND function verifies the existence of an incoming data value in an indexed FOCUS 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.

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 a Data Source**

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

where:

- `fieldname` is the name of the field that contains the incoming data value.
AS *dbfield*

Is the name of the data source field whose values are compared to the incoming field values.

This field must be indexed. If the incoming field and the data source field have the same name, omit this phrase.

For Maintain - the AS field is required and the name must be qualified.

For MODIFY - the AS field must be indexed. If the incoming field and the data source field have the same name, omit this phrase.

*file*

Is the name of the indexed FOCUS data source.

For Maintain - the IN file is unnecessary since the AS field name is required and must be qualified.

For MODIFY - the IN field must be indexed.

**Note:**

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

**Example:** *Verifying the Existence of a Value in Another Data Source (Maintain)*

In the following example, FIND determines if a data value is found in another data source.

```
MAINTAIN FILE MOVIES AND VIDEOTRK
FOR ALL NEXT MOVIES.MOVIECODE INTO FILMSTK
TYPE "RC SHOULD BE 1 WHERE MOVIECODE EXISTS IN BOTH FILES";
TYPE ""
COMPUTE RC/I1;
COMPUTE I/I1=1;
REPEAT FILMSTK.FOCCOUNT
  COMPUTE RC= FIND(FILMSTK(I).MOVIECODE AS VIDEOTRK.MOVIECODE)
  TYPE "FOR MOVIECODE = <<FILMSTK(I).MOVIECODE , RC = <<RC"
  COMPUTE I=I+1;
ENDREPEAT
END
```
The output is:

RC SHOULD BE 1 WHERE MOVIECODE EXISTS IN BOTH FILES
FOR MOVIECODE = 001MCA, RC = 1

FOR MOVIECODE = 387PLA, RC = 0

FOR MOVIECODE = 963CBS, RC = 1
TRANSACTIONS: COMMITS = 1 ROLLBACKS = 0
SEGMENTS: INCLUDED = 0 UPDATED = 0 DELETED = 0

Example: Verifying the Existence of a Value in the Same Data Source (Maintain)

In the following example, FIND determines if a data value is found in the same data source.

```
MAINTAIN FILE CAR
COMPUTE RETAIL_COST=31500;
COMPUTE CHECK/I1;
COMPUTE CHECK= FIND (RETAIL_COST);
  IF CHECK = 1 THEN GOTO FOUND1
  ELSE GOTO NOT1;
CASE FOUND1
  TYPE "THERE IS A CAR WITH A RETAIL_COST OF <<RETAIL_COST"
-* ....
ENDCASE
CASE NOT1
  TYPE "THERE IS NO CAR WITH A RETAIL_COST OF <<RETAIL_COST"
-*....
ENDCASE
-*....
END
```

The output is:

THERE IS A CAR WITH A RETAIL_COST OF 31,500
TRANSACTIONS: COMMITS = 1 ROLLBACKS = 0
SEGMENTS: INCLUDED = 0 UPDATED = 0 DELETED = 0
**Example:** Verifying the Existence of a Value in an Indexed Field (MODIFY)

FIND determines if a supplied value in the EMP_ID field is in the EDUCFILE data source. The procedure then displays a message indicating the result of the search.

```plaintext
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
COMPUTE
   EDTEST = FIND(EMP_ID IN EDUCFILE);
   MSG/A40 = IF EDTEST NE 0 THEN
      'STUDENT LISTED IN EDUCATION FILE' ELSE
      'STUDENT NOT LISTED IN EDUCATION FILE';
MATCH EMP_ID
   ON NOMATCH TYPE "<MSG"
   ON MATCH TYPE "<MSG"
DATA

A sample execution is:

```
> EMPLOYEE ON 12/04/2001 AT 12.09.03
DATA FOR TRANSACTION 1
   EMP_ID               = 112847612
   STUDENT LISTED IN EDUCATION FILE
DATA FOR TRANSACTION 2
   EMP_ID               = 219984371
   STUDENT NOT LISTED IN EDUCATION FILE
DATA FOR TRANSACTION 3
```

The procedure processes as follows:

1. The procedure prompts you for an employee ID. You enter 112847612.
2. The procedure searches the EDUCFILE data source for the employee ID 112847612. It finds the ID so it prints STUDENT LISTED IN EDUCATION FILE.
3. The procedure prompts you for an employee ID. You enter 219984371.
4. The procedure searches the EDUCFILE data source for the employee ID 219984371. It does not find the ID so it prints STUDENT NOT LISTED IN EDUCATION FILE.
**Example: Rejecting a Transaction When a Value Is Not Found (MODIFY)**

The following updates the number of hours an employee spent in class. The VALIDATE command rejects a transaction for an employee whose ID is not found in the EDUCFILE data source, which records class attendance.

```plaintext
MODIFY FILE EMPLOYEE
PROMPT EMP_ID ED_HRS
VALIDATE
  EDTEST = FIND(EMP_ID IN EDUCFILE);
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE ED_HRS
DATA
```

A sample execution is:

```
> EMPLOYEE ON 12/04/2001 AT 12/26/08
DATA FOR TRANSACTION 1
EMP_ID       = 112847612
ED_HRS      = 7
DATA FOR TRANSACTION 2
EMP_ID       = 219984371
ED_HRS      = 0
(FOC421) TRANS 2 REJECTED INVALID EDTEST
219984371, 0, $
DATA FOR TRANSACTION 3
```

The procedure processes as follows:

1. The procedure prompts you for an employee ID and the number of hours the employee spent in class. You enter the following data:

   ```plaintext
   EMP_ID:  112847612
   ED_HRS:  7
   ```

2. The procedure updates the number of hours for the ID 112847612.

3. The procedure prompts you for an employee ID and the number of hours the employee spent in class. You enter the following data:

   ```plaintext
   EMP_ID:  219984371
   ED_HRS:  0
   ```
4. The procedure rejects the record for the ID 219984371 because it does not exist in the EDUCFILE data source, and an error message is returned.

**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)
```

**IMPUTE: Replacing Missing Values With Aggregated Values**

IMPUTE calculates a value to replace missing numeric data on report output, within a partition.

In place of eliminating data records with missing values from analysis, IMPUTE enables you to substitute a variety of estimates for the missing values, including the mean, the median, the mode, or a numeric constant, all calculated within the data partition specified by the reset key.

This function is designed to be used with detail level reports (PRINT or LIST commands), and with calculated values (fields created with the COMPUTE command).

**Syntax:** How to Replace Missing Values With Aggregated Values

```
IMPUTE (field, reset_key, replacement)
```

where:

- **field**
  - Is the name of the numeric input field that is defined with MISSING ON.

- **reset_key**
  - Defines the partition for the calculation. Valid values are:
    - A sort field name.
    - PRESET, which uses the break defined by the SET PARTITION_ON command.
    - TABLE, which performs the calculation on the entire table.

- **replacement**
  - Is a numeric constant or one of the following:
    - MEAN
    - MEDIAN
    - MODE
Example: Replacing Missing Values With Aggregated Values

To run this example, the FOCUS data source SALEMISS must be created. SALEMISS is the
SALES data source with some missing values added in the RETURNS and DAMAGED fields.
The following is the SALEMISS Master File, which should be added to the IBISAMP application.

FILENAME=KSALES, SUFFIX=FOC, REMARKS='Legacy Metadata Sample: sales',$

SEGNAME=STOR_SEG, SEGTYPE=S1,
    FIELDNAME=STORE_CODE, ALIAS=SNO, FORMAT=A3, $
    FIELDNAME=CITY, ALIAS=CTY, FORMAT=A15, $
    FIELDNAME=AREA, ALIAS=LOC, FORMAT=A1, $

SEGNAME=DATE_SEG, PARENT=STOR_SEG, SEGTYPE=SH1,
    FIELDNAME=DATE, ALIAS=DTE, FORMAT=A4MD, $

SEGNAME=PRODUCT, PARENT=DATE_SEG, SEGTYPE=S1,
    FIELDNAME=PROD_CODE, ALIAS=PCODE, FORMAT=A3, FIELDTYPE=I, $
    FIELDNAME=UNIT_SOLD, ALIAS=SOLD, FORMAT=I5, $
    FIELDNAME=RETAIL_PRICE, ALIAS=RP, FORMAT=D5.2M, $
    FIELDNAME=DELIVER_AMT, ALIAS=SHIP, FORMAT=I5, $
    FIELDNAME=OPENING_AMT, ALIAS=INV, FORMAT=I5, $
    FIELDNAME=RETURNS, ALIAS=RTN, FORMAT=I3, MISSING=ON, $
    FIELDNAME=DAMAGED, ALIAS=BAD, FORMAT=I3, MISSING=ON, $

IMPUTE: Replacing Missing Values With Aggregated Values
The following procedure creates the SALEMISS data source and then adds the missing values to the RETURNS and DAMAGED fields:

```
CREATE FILE ibisamp/SALEMISS
MODIFY FILE ibisamp/SALEMISS
FIXFORM STORE_CODE/3 CITY/15 AREA/1 DATE/4 PROD_CODE/3
FIXFORM UNIT_SOLD/5 RETAIL_PRICE/5 DELIVER_AMT/5
FIXFORM OPENING_AMT/5 RETURNS/3 DAMAGED/3
MATCH STORE_CODE
ON NOMATCH INCLUDE
ON MATCH CONTINUE
MATCH DATE
ON NOMATCH INCLUDE
ON MATCH CONTINUE
MATCH PROD_CODE
ON NOMATCH INCLUDE
ON MATCH REJECT
DATA
14BSTAMFORD S1212B10 60 .95 80 65 10 6
14BSTAMFORD S1212B12 40 1.29 20 50 3 3
14BSTAMFORD S1212B17 29 1.89 30 30 2 1
14BSTAMFORD S1212C13 25 1.99 30 40 3 0
14BSTAMFORD S1212C7 45 2.39 50 49 5 4
14BSTAMFORD S1212D12 27 2.19 40 35 0 0
14BSTAMFORD S1212E2 80 .99 100 100 9 4
14BSTAMFORD S1212E3 70 1.09 80 90 8 9
14NEW YORK U1017B10 30 .85 30 10 2 3
14NEW YORK U1017B17 20 1.89 40 25 2 1
14NEW YORK U1017B20 15 1.99 30 5 0 1
14NEW YORK U1017C17 12 2.09 10 15 0 0
14NEW YORK U1017D12 20 2.09 30 10 3 2
14NEW YORK U1017E1 30 .89 25 45 4 7
14NEW YORK U1017E3 35 1.09 25 45 4 2
77FUNIONDALE R1018B20 25 2.09 40 25 1 1
77FUNIONDALE R1018C7 40 2.49 40 40 0 0
K1 NEWARK U1019B12 29 1.49 30 30 1 0
K1 NEWARK U1018B10 13 .99 30 15 1 1
END
-RUN
```
MODIFY FILE ibisamp/SALEMISS
FIXFORM STORE_CODE/3 DATE/5 PROD_CODE/4
FIXFORM UNIT/3 RETAIL/5 DELIVER/3
FIXFORM OPEN/3 RETURNS/C3 DAMAGED/C3
MATCH STORE_CODE
ON NOMATCH INCLUDE
ON MATCH CONTINUE
MATCH DATE
ON NOMATCH INCLUDE
ON MATCH CONTINUE
MATCH PROD_CODE
ON NOMATCH INCLUDE
ON MATCH REJECT
DATA
14Z1017 C13 15 1.99 35 30 6
1421017 C14 18 2.05 30 25 4
1421017 E2 33 0.99 45 40
END
-RUN

The following request against the SALEMISS data source generates replacement values for the missing values in the RETURNS field, using only the values within the same store.

SET PARTITION_ON=FIRST
TABLE FILE SALEMISS
PRINT RETURNS
COMPUTE MEDIAN1 = IMPUTE(RETURNS, PRESET, MEDIAN);
COMPUTE MEAN1 = IMPUTE(RETURNS, PRESET, MEAN);
COMPUTE MODE1 = IMPUTE(RETURNS, PRESET, MODE);
BY STORE_CODE
ON TABLE SET PAGE NOPAGE
ON TABLE SET STYLE *
TYPE=REPORT, GRID=OFF,$
ENDSTYLE
END
The output is shown in the following image. The missing values occur in store 14Z, and the replacement values are calculated using only the RETURNS values from that store because PARTITION_ON is set to FIRST.

<table>
<thead>
<tr>
<th>STORE_CODE</th>
<th>RETURNS</th>
<th>MEAN1</th>
<th>MEAN1</th>
<th>MODE1</th>
</tr>
</thead>
<tbody>
<tr>
<td>14B</td>
<td>10</td>
<td>10.00</td>
<td>10.00</td>
<td>10.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3.00</td>
<td>3.00</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3.00</td>
<td>3.00</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5.00</td>
<td>5.00</td>
<td>5.00</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>.00</td>
<td>.00</td>
<td>.00</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>9.00</td>
<td>9.00</td>
<td>9.00</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8.00</td>
<td>8.00</td>
<td>8.00</td>
</tr>
<tr>
<td>14Z</td>
<td>2</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>.00</td>
<td>.00</td>
<td>.00</td>
</tr>
<tr>
<td>.</td>
<td>2.00</td>
<td>2.00</td>
<td>4.00</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4.00</td>
<td>4.00</td>
<td>4.00</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>.00</td>
<td>.00</td>
<td>.00</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3.00</td>
<td>3.00</td>
<td>3.00</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4.00</td>
<td>4.00</td>
<td>4.00</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>2.00</td>
<td>2.00</td>
<td>4.00</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4.00</td>
<td>4.00</td>
<td>4.00</td>
<td></td>
</tr>
<tr>
<td>77F</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>0</td>
<td>.00</td>
<td>.00</td>
<td>.00</td>
<td></td>
</tr>
<tr>
<td>K1</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td></td>
</tr>
</tbody>
</table>
Changing the PARTITION_ON setting to TABLE produces the following output, in which the replacement values are calculated using all of the rows in the table.

<table>
<thead>
<tr>
<th>STORE_CODE</th>
<th>RETURNS</th>
<th>MEDIAN1</th>
<th>MEAN1</th>
<th>MODE1</th>
</tr>
</thead>
<tbody>
<tr>
<td>14B</td>
<td>10</td>
<td>10.00</td>
<td>10.00</td>
<td>10.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3.00</td>
<td>3.00</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3.00</td>
<td>3.00</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5.00</td>
<td>5.00</td>
<td>5.00</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>.00</td>
<td>.00</td>
<td>.00</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>9.00</td>
<td>9.00</td>
<td>9.00</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8.00</td>
<td>8.00</td>
<td>8.00</td>
</tr>
<tr>
<td>14Z</td>
<td>2</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>.00</td>
<td>.00</td>
<td>.00</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>2.00</td>
<td>3.00</td>
<td>.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4.00</td>
<td>4.00</td>
<td>4.00</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>.00</td>
<td>.00</td>
<td>.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3.00</td>
<td>3.00</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4.00</td>
<td>4.00</td>
<td>4.00</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>2.00</td>
<td>3.00</td>
<td>.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4.00</td>
<td>4.00</td>
<td>4.00</td>
</tr>
<tr>
<td>77F</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>.00</td>
<td>.00</td>
<td>.00</td>
</tr>
<tr>
<td>K1</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

LAST: Retrieving the Preceding Value

Available Languages: reporting

The LAST function retrieves the preceding value for a field.

The effect of LAST depends on whether it appears in a DEFINE or COMPUTE command:

- In a DEFINE command, the LAST value applies to the previous record retrieved from the data source before sorting takes place.
In a COMPUTE command, the LAST value applies to the record in the previous line of the internal matrix.

Do not use LAST with the -SET command in Dialogue Manager.

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 argument.

**Example:** Retrieving the Preceding Value

LAST retrieves the previous value of the DEPARTMENT field to determine whether to restart the running total of salaries by department. If the previous value equals the current value, CURR_SAL is added to RUN_TOT to generate a running total of salaries within each department.

TABLE FILE EMPLOYEE
PRINT LAST_NAME CURR_SAL AND COMPUTE
RUN_TOT/D12.2M = IF DEPARTMENT EQ LAST DEPARTMENT THEN
  (RUN_TOT + CURR_SAL) ELSE CURR_SAL ;
AS 'RUNNING,TOTAL,SALARY'
BY DEPARTMENT SKIP-LINE
END
The output is:

<table>
<thead>
<tr>
<th>DEPARTMENT</th>
<th>LAST_NAME</th>
<th>CURR_SAL</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIS</td>
<td>SMITH</td>
<td>$13,200.00</td>
<td>$13,200.00</td>
</tr>
<tr>
<td></td>
<td>JONES</td>
<td>$18,480.00</td>
<td>$31,680.00</td>
</tr>
<tr>
<td></td>
<td>MCCOY</td>
<td>$18,480.00</td>
<td>$50,160.00</td>
</tr>
<tr>
<td></td>
<td>BLACKWOOD</td>
<td>$21,780.00</td>
<td>$71,940.00</td>
</tr>
<tr>
<td></td>
<td>GREENSPAN</td>
<td>$9,000.00</td>
<td>$80,940.00</td>
</tr>
<tr>
<td></td>
<td>CROSS</td>
<td>$27,062.00</td>
<td>$108,002.00</td>
</tr>
<tr>
<td>PRODUCTION</td>
<td>STEVENS</td>
<td>$11,000.00</td>
<td>$11,000.00</td>
</tr>
<tr>
<td></td>
<td>SMITH</td>
<td>$9,500.00</td>
<td>$20,500.00</td>
</tr>
<tr>
<td></td>
<td>BANNING</td>
<td>$29,700.00</td>
<td>$50,200.00</td>
</tr>
<tr>
<td></td>
<td>IRVING</td>
<td>$26,862.00</td>
<td>$77,062.00</td>
</tr>
<tr>
<td></td>
<td>ROMANS</td>
<td>$21,120.00</td>
<td>$98,182.00</td>
</tr>
<tr>
<td></td>
<td>MCKNIGHT</td>
<td>$16,100.00</td>
<td>$114,282.00</td>
</tr>
</tbody>
</table>

LAST retrieves the previous value of DEPARTMENT:

LAST DEPARTMENT

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

Available Languages: MODIFY

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 Master File 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.

To modify more than one data source in one request, use the COMBINE command or the Maintain Data facility.

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

\[
\text{RTN} = \text{LOOKUP}(\text{SEG.DATE_ATTEND});
\]

- 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**

\[
\text{LOOKUP}(\text{field});
\]

where:

\[
\text{field}
\]

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 argument.

**Example:**

**Using the LOOKUP Function**

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

\[
\text{LOOKUP}(\text{DATE_ENROLL})
\]
Example: Reading a Value From a Cross-referenced Data Source

You may need to determine if employees were hired before or after a specific date, for example, January 1, 1982. The employee IDs (EMP_ID) and hire date (HIRE_DATE) are located in the host segment. The following diagram shows the file structure:

![Diagram of file structure](image)

The request is:

```sql
MODIFY FILE EMPLOYEE
PROMPT EMP_ID ED_HRS
COMPUTE EDTEST = LOOKUP(HIRE_DATE);
    COMPUTE ED_HRS = IF DATE_ENROLL GE 820101 THEN ED_HRS * 1.1
    ELSE ED_HRS;
MATCH EMP_ID
    ON MATCH UPDATE ED_HRS
    ON NOMATCH REJECT
DATA
```

A sample execution is:

1. The request prompts you for the employee ID and number of class hours. Enter the ID 117593129 and 10 class hours.
2. LOOKUP locates the first instance in the cross-referenced segment containing the employee ID 117593129. Since the instance exists, the function returns a 1 to the EDTEST variable. This instance lists the enroll date as 821028 (October 28, 1982).

3. LOOKUP retrieves the value 821028 for the DATE_ENROLL field.

4. The COMPUTE command tests the value of DATE_ENROLL. Since October 28, 1982 is after January 1, 1982, the ED_HRS are increased from 10 to 11.

5. The request updates the classroom hours for employee 117593129 with the new value.

Example: Using a Value in a Host Segment to Search a Data Source

You can use a field value in a host segment instance to search a cross-referenced segment. Do the following:

- In the MATCH command that selects the host segment instance, activate the host field with the ACTIVATE command.
- In the same MATCH command, code LOOKUP after the ACTIVATE command.

This request displays the employee ID, date of salary increase, employee name, and the employee position after the raise was granted:

- The employee ID and name (EMP_ID) are in the root segment.
- The date of increase (DAT_INC) is in the descendant host segment.
- The job position is in the cross-referenced segment.
- The shared field is JOBCODE. You never enter a job code; the values are stored in the data source.

The request is:

```mod
MODIFY FILE EMPLOYEE
PROMPT EMP_ID DAT_INC
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE
MATCH DAT_INC
  ON NOMATCH REJECT
  ON MATCH ACTIVATE JOBCODE
  ON MATCH COMPUTE
    RTN = LOOKUP (JOB_DESC) ;
  ON MATCH TYPE
    "EMPLOYEE ID:        <EMP_ID"
    "DATE INCREASE:      <DAT_INC"
    "NAME:               <D.FIRST_NAME <D.LAST_NAME"
    "POSITION:           <JOB_DESC"
DATA
```

7. Data Source and Decoding Functions
A sample execution is:

1. The request prompts you for the employee ID and date of pay increase. Enter the employee ID 071382660 and the date 820101 (January 1, 1982).

2. The request locates the instance containing the ID 071382660, then locates the child instance containing the date of increase 820101.

3. This child instance contains the job code A07. The ACTIVATE command makes this value available to LOOKUP.

4. LOOKUP locates the job code A07 in the cross-referenced segment. It returns a 1 the RTN variable and retrieves the corresponding job description SECRETARY.

5. The TYPE command displays the values:

```
EMPLOYEE ID: 071382660
DATE INCREASE: 82/01/01
NAME: ALFRED STEVENS
POSITION: SECRETARY
```

Fields retrieved by LOOKUP do not require the D. prefix. FOCUS treats the field values as transaction values.

You may also need to activate the host field if you are using LOOKUP within a NEXT command. This request displays the latest position held by an employee:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE
NEXT DAT_INC
  ON NONEXT REJECT
  ON NEXT ACTIVATE JOBCODE
  ON NEXT COMPUTE
    RTN = LOOKUP (JOB_DESC) ;
  ON MATCH TYPE
    "EMPLOYEE ID: <EMP_ID"
    "DATE OF POSITION: <DAT_INC"
    "NAME: <D.FIRST_NAME <D.LAST_NAME"
    "POSITION: <JOB_DESC"
DATA
```

**Example:** Using the LOOKUP Function With a VALIDATE Command

When you use LOOKUP, reject transactions containing values for which there is no corresponding instance in the cross-reference segment. To do this, place the function in a VALIDATE command. If the function cannot locate the instance in the cross-referenced segment, it sets the value of the return variable to 0, causing the request to reject the transaction.
The following request updates an employee’s classroom hours (ED_HRS). If the employee enrolled in classes on or after January 1, 1982, the request increases the number of classroom hours by 10%. The enrollment dates are stored in a cross-referenced segment (field DATE_ATTEND). The shared field is the employee ID.

The request is as follows:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID ED_HRS
VALIDATE
  TEST_DATE = LOOKUP(DATE_ENROLL);
COMPUTE
  ED_HRS = IF DATE_ENROLL GE 820101 THEN ED_HRS * 1.1
           ELSE ED_HRS;
MATCH EMP_ID
  ON MATCH UPDATE ED_HRS
  ON NOMATCH REJECT
DATA
```

If an employee record is not found in the cross-referenced segment, that employee never enrolled in a class. The transaction is rejected as an error.

### Using the Extended LOOKUP Function

If the LOOKUP function cannot locate a value of the host field in the cross-referenced segment, use extended syntax to locate the next highest or lowest cross-referenced field value in the cross-referenced segment.

To use this feature, create the index with the INDEX parameter set to NEW (the binary tree scheme). To determine the type of index used by a data source, enter the FDT command.

**Syntax:** How to Use the Extended LOOKUP Function

```
COMPUTE
  LOOKUP(field action);
```

where:

- **field**

  Is the name of the field in the cross-referenced data source, used in a MODIFY computation. If the field name also exists in the host data source, you must qualify it here.
**action**

Specifies the action the request takes. Valid values are:

**EQ** causes LOOKUP to take no further action if an exact match is not found. If a match is found, the value of rcode is set to 1; otherwise, it is set to 0. This is the default.

**GE** causes LOOKUP to locate the instance with the next highest value of the cross-referenced field. The value of rcode is set to 2.

**LE** causes LOOKUP to locate the instance with the next lowest value of the cross-referenced field. The value of rcode is set to -2.

Do not include a space between LOOKUP and the left parenthesis.

The following table shows the value of rcode, depending on which instance LOOKUP locates:

<table>
<thead>
<tr>
<th>Value</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Exact cross-referenced value located.</td>
</tr>
<tr>
<td>2</td>
<td>Next highest cross-referenced value located.</td>
</tr>
<tr>
<td>-2</td>
<td>Next lowest cross-referenced value located.</td>
</tr>
<tr>
<td>0</td>
<td>Cross-referenced value not located.</td>
</tr>
</tbody>
</table>

**NULLIF: Returning a Null Value When Parameters Are Equal**

NULLIF returns a null (missing) value when its parameters are equal. If they are not equal, it returns the first value. The field to which the value is returned should have MISSING ON.

**Syntax:** How to Return a Null Value for Equal Parameters

```
NULLIF(arg1, arg2)
```

where:

**arg1, arg2**

Any type of field, constant, or expression.

Are the input parameters that are tested for equality. They must either both be numeric or both be alphanumeric.

The output data type is the same as the input data types.
Example:  Testing for Equal Parameters

The following request uses NULLIF to test the DAMAGED and RETURNS field values for equality.

```
DEFINE FILE SALES
  NULL1/I4 MISSING ON = NULLIF(DAMAGED, RETURNS);
END
TABLE FILE SALES
PRINT DAMAGED RETURNS NULL1
   BY STORE_CODE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
END
```

The output is shown in the following image.

<table>
<thead>
<tr>
<th>STORE_CODE</th>
<th>DAMAGED</th>
<th>RETURNS</th>
<th>NULL1</th>
</tr>
</thead>
<tbody>
<tr>
<td>14B</td>
<td>6</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>14Z</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>77F</td>
<td>1</td>
<td>1</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>.</td>
</tr>
<tr>
<td>K1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>.</td>
</tr>
</tbody>
</table>
NULLIF tests the DAMAGED and RETURNS field values for equality.

\[
\text{NULLIF(DAMAGED, RETURNS)}
\]

For DAMAGED=3 and RETURNS = 3, the result is MISSING (.).
For DAMAGED=2 and RETURNS = 3, the result is 2.
Simplified date and date-time functions have streamlined parameter lists, similar to those used by SQL functions. In some cases, these simplified functions provide slightly different functionality than previous versions of similar functions.

The simplified functions do not have an output argument. Each function returns a value that has a specific data type.

When used in a request against a relational data source, these functions are optimized (passed to the RDBMS for processing).

Standard date and date-time formats refer to YYMD and HYYMD syntax (dates that are not stored in alphanumeric or numeric fields). Dates not in these formats must be converted before they can be used in the simplified functions. Input date and date-time parameters must provide full component dates. Literal date-time values can be used with the DT function.

All arguments can be either literals, field names, or amper variables.

**Note:** The simplified date and date-time functions are not supported in Maintain Data.

**In this chapter:**

- **DT_CURRENT_DATE:** Returning the Current Date
- **DT_CURRENT_DATETIME:** Returning the Current Date and Time
- **DT_CURRENT_TIME:** Returning the Current Time
- **DTADD:** Incrementing a Date or Date-Time Component
- **DTDIFF:** Returning the Number of Component Boundaries Between Date or Date-Time Values
- **DTIME:** Extracting Time Components From a Date-Time Value
- **DTPART:** Returning a Date or Date-Time Component in Integer Format
- **DTRUNC:** Returning the Start of a Date Period for a Given Date
DT_CURRENT_DATE: Returning the Current Date

The DT_CURRENT_DATE function returns the current date-time provided by the running operating environment in date-time format. The time portion of the date-time is set to zero.

Syntax: How to Return the Current Date

DT_CURRENT_DATE()

Example: Returning the Current Date

The following request returns the current date.

DEFINE FILE WF_RETAIL_LITE
CURRDATE/YYMD WITH COUNTRY_NAME = DT_CURRENT_DATE();
END
TABLE FILE WF_RETAIL_LITE
SUM CURRDATE
ON TABLE SET PAGE NOPAGE
END

The output is shown in the following image.

![CURRDATE 2016/09/08]

DT_CURRENT_DATE returns the current date.

DT_CURRENT_DATE()

For September 8, 2016 (returning to a YYMD field), the result is 2016/09/08.

DT_CURRENT_DATETIME: Returning the Current Date and Time

DT_CURRENT_DATETIME returns the current date and time provided by the running operating environment in date-time format, with a specified time precision.

Syntax: How to Return the Current Date and Time

DT_CURRENT_DATETIME(component)
where:

**component**

Is one of the following time precisions.

- **SECOND.**
- **MILLISECOND.**
- **MICROSECOND.**

**Note:** The field to which the value is returned must have a format that supports the time precision requested.

**Example:** Returning the Current Date and Time

The following request returns the current date and time, with the time specified in microseconds.

```sql
DEFINE FILE WF RETAIL_LITE
CURRDATE/HYYMDm WITH COUNTRY_NAME = DT_CURRENT_DATETIME(MICROSECOND);
END
TABLE FILE WF RETAIL_LITE
SUM CURRDATE
ON TABLE SET PAGE NOPAGE
END
```

The output is shown in the following image.

```
CURRDATE
2016/09/08 17:10:31.605718
```

**DT_CURRENT_DATETIME** returns the current date and time to microsecond precision.

**DT_CURRENT_DATETIME (MICROSECOND)**

For September 8, 2106 at 5:10:31.605718 p.m. (returned to a field with format HYYMDm), the result is 2016/09/08 17:10:31.605718.

**DT_CURRENT_TIME:** Returning the Current Time

The **DT_CURRENT_TIME** function returns the current time provided by the running operating environment in date-time format, with a specified time precision. The date portion of the returned date-time value is set to zero.
Syntax: How to Return the Current Time

\[ \text{DT_CURRENT_TIME}(\text{component}) \]

where:

\( \text{component} \) Is one of the following time precisions.

- SECOND.
- MILLISECOND.
- MICROSECOND.

Note: The field to which the value is returned must have a format that supports the time precision requested.

Example: Returning the Current Time

The following request returns the current time, with the time precision set to milliseconds.

```
DEFINE FILE WF_RETAIL_LITE
CURRTIME/HHISs WITH COUNTRY_NAME = DT_CURRENT_TIME(MILLISECOND);
END
TABLE FILE WF_RETAIL_LITE
SUM CURRTIME
ON TABLE SET PAGE NOPAGE
END
```

The output is shown in the following image.

```
| CURRTIME | 17:23:13.098 |
```

DT_CURRENT_TIME returns the current time in milliseconds.

```
DT_CURRENT_TIME(MILLISECOND)
```

For 5:23:13.098 p.m. (returned to a field with format HHISs), the result is 17:23:13.098.

DTADD: Incrementing a Date or Date-Time Component

Given a date in standard date or date-time format, DTADD returns a new date after adding the specified number of a supported component. The returned date format is the same as the input date format.
**Syntax:** How to Increment a Date or Date-Time Component

\[ \text{DTADD}(\text{date}, \text{component}, \text{increment}) \]

where:

- **date**  
  Date or date-time  
  Is the date or date-time value to be incremented, which must provide a full component date.

- **component**  
  Keyword  
  Is the component to be incremented. Valid components (and acceptable values) are:

  - YEAR (1-9999).
  - QUARTER (1-4).
  - MONTH (1-12).
  - WEEK (1-53). This is affected by the WEEKFIRST setting.
  - DAY (of the Month, 1-31).
  - HOUR (0-23).
  - MINUTE (0-59).
  - SECOND (0-59).

- **increment**  
  Integer  
  Is the value (positive or negative) to add to the component.
Example: **Incrementing the DAY Component of a Date**

The following request against the WF_RETAIL data source adds three days to the employee date of birth:

```plaintext
DEFINE FILE WF_RETAILLITE
NEWDATE/YYMD = DTADD(DATE_OF_BIRTH, DAY, 3);
MGR/A3 = DIGITS(ID_MANAGER, 3);
END
TABLE FILE WF_RETAILLITE
SUM MGR NOPRINT DATE_OF_BIRTH NEWDATE
BY MGR
ON TABLE SET PAGE NOPAGE
END
```

The output is:

```
<table>
<thead>
<tr>
<th>MGR</th>
<th>Date of Birth</th>
<th>NEWDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>1985/01/29</td>
<td>1985/02/01</td>
</tr>
<tr>
<td>101</td>
<td>1982/04/01</td>
<td>1982/04/04</td>
</tr>
<tr>
<td>201</td>
<td>1976/11/14</td>
<td>1976/11/17</td>
</tr>
<tr>
<td>301</td>
<td>1980/05/15</td>
<td>1980/05/18</td>
</tr>
<tr>
<td>401</td>
<td>1975/10/19</td>
<td>1975/10/22</td>
</tr>
<tr>
<td>501</td>
<td>1985/04/11</td>
<td>1985/04/14</td>
</tr>
<tr>
<td>601</td>
<td>1967/02/03</td>
<td>1967/02/06</td>
</tr>
<tr>
<td>701</td>
<td>1977/10/16</td>
<td>1977/10/19</td>
</tr>
<tr>
<td>801</td>
<td>1970/04/18</td>
<td>1970/04/21</td>
</tr>
<tr>
<td>901</td>
<td>1972/03/29</td>
<td>1972/04/01</td>
</tr>
<tr>
<td>999</td>
<td>1976/10/21</td>
<td>1976/10/24</td>
</tr>
</tbody>
</table>
```

DTADD adds three days to the employee date of birth:

```plaintext
DTADD(DATE_OF_BIRTH, DAY, 3)
```

For 1976/10/21, the result is 1976/10/24.
Reference: Usage Notes for DTADD

- Each element must be manipulated separately. Therefore, if you want to add 1 year and 1 day to a date, you need to call the function twice, once for YEAR (you need to take care of leap years) and once for DAY. The simplified functions can be nested in a single expression, or created and applied in separate DEFINE or COMPUTE expressions.

- With respect to parameter validation, DTADD will not allow anything but a standard date or a date-time value to be used in the first parameter.

- The increment is not checked, and the user should be aware that decimal numbers are not supported and will be truncated. Any combination of values that increases the YEAR beyond 9999 returns the input date as the value, with no message. If the user receives the input date when expecting something else, it is possible there was an error.

**DTDIFF: Returning the Number of Component Boundaries Between Date or Date-Time Values**

Given two dates in standard date or date-time formats, DTDIFF returns the number of given component boundaries between the two dates. The returned value has integer format for calendar components or double precision floating point format for time components.

**Syntax:** How to Return the Number of Component Boundaries

\[
\text{DTDIFF} (\text{end\_date}, \text{start\_date}, \text{component})
\]

where:

**end\_date**

- Date or date-time

Is the ending full-component date in either standard date or date-time format. If this date is given in standard date format, all time components are assumed to be zero.

**start\_date**

- Date or date-time

Is the starting full-component date in either standard date or date-time format. If this date is given in standard date format, all time components are assumed to be zero.
**component**

Keyword

Is the component on which the number of boundaries is to be calculated. For example, QUARTER finds the difference in quarters between two dates. Valid components (and acceptable values) are:

- **YEAR** (1-9999).
- **QUARTER** (1-4).
- **MONTH** (1-12).
- **WEEK** (1-53). This is affected by the WEEKFIRST setting.
- **DAY** (of the Month, 1-31).
- **HOUR** (0-23).
- **MINUTE** (0-59).
- **SECOND** (0-59).

**Example:** Returning the Number of Years Between Two Dates

The following request against the WF_RETAIL data source calculates employee age when hired:

```plaintext
DEFINE FILE WF_RETAILLITE
YEARS/I9 = DTDIFF(START_DATE, DATE_OF_BIRTH, YEAR);
END
TABLE FILE WF_RETAILLITE
PRINT START_DATE DATE_OF_BIRTH YEARS AS 'Hire,Age'
BY  EMPLOYEE_NUMBER
WHERE EMPLOYEE_NUMBER CONTAINS 'AA'
ON TABLE SET PAGE NOPAGE
END
```
The output is:

![Employee Data Table]

DTDIFF calculates employee age when hired:

```
DTDIFF(START_DATE, DATE_OF_BIRTH, YEAR)
```

For the date of birth 1991/06/04 and the start date 2008/11/14, the result is 17.

**DTIME: Extracting Time Components From a Date-Time Value**

Given a date-time value and time component keyword as input, DTIME returns the value of all of the time components up to and including the requested component. The remaining time components in the value are set to zero. The field to which the time component is returned must have a time format that supports the component being returned.

**Syntax:**

```
DTIME(datetime, component)
```

where:

- `datetime`
  - Date-time

  Is the date-time value from which to extract the time component. It can be a field name or a date-time literal. It must provide a full component date.
component

Keyword

Valid values are:

- **TIME.** The complete time portion is returned. Its smallest component depends on the input date-time format. Nanoseconds are not supported or returned.
- **HOUR.** The time component up to and including the hour component is extracted.
- **MINUTE.** The time component up to and including the minute component is extracted.
- **SECOND.** The time component up to and including the second component is extracted.
- **MILLISECOND.** The time component up to and including the millisecond component is extracted.
- **MICROSECOND.** The time component up to and including the microsecond component is extracted.

**Example: Extracting Time Components**

The following request defines two date-time fields:

- **TRANSTIME** contains the extracted time components from **TRANSDATE** down to the minute.
- **TRANSTIME2** extracts all of the time components from the literal date-time value 2018/01/17 05:45:22.777888.

```plaintext
DEFINE FILE VIDEOTR2
TRANSTIME/HHISsm = DTIME(TRANSDATE, MINUTE);
TRANSTIME2/HHISsm = DTIME(DT(2018/01/17 05:45:22.777888), TIME);
END
TABLE FILE VIDEOTR2
SUM  TRANSTIME TRANSTIME2
   BY MOVIECODE
   BY  TRANSDATE
WHERE MOVIECODE CONTAINS 'MGM'
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF, $
ENDSTYLE
END
```
DTIME extracts the TIME component from the data-time value 2018/01/17 05:45:22.777888.

\[
\text{DTIME(DT(2018/01/17 05:45:22.777888), \text{TIME})}
\]

The result is 05:45:22.777888.

**DTPART: Returning a Date or Date-Time Component in Integer Format**

Given a date in standard date or date-time format and a component, DTPART returns the component value in integer format.

**Syntax:**

\[
\text{DTPART(\text{date, component})}
\]

where:

\[
\text{date}
\]

Date or date-time

Is the full-component date in standard date or date-time format.
component
Keyword

Is the component to extract in integer format. Valid components (and values) are:

- YEAR (1-9999).
- QUARTER (1-4).
- MONTH (1-12).
- WEEK (of the year, 1-53). This is affected by the WEEKFIRST setting.
- DAY (of the Month, 1-31).
- DAY_OF_YEAR (1-366).
- WEEKDAY (day of the week, 1-7). This is affected by the WEEKFIRST setting.
- HOUR (0-23).
- MINUTE (0-59).
- SECOND (0-59).
- MILLISECOND (0-999).
- MICROSECOND (0-999999).

**Example:** Extracting the Quarter Component as an Integer

The following request against the WF_RETAIL data source extracts the QUARTER component from the employee start date:

```sql
DEFINE FILE WF_RETAILLITE
  QTR/I2 = DTPART(START_DATE, QUARTER);
END
TABLE FILE WF_RETAILLITE
  PRINT START_DATE QTR AS Quarter
  BY EMPLOYEE_NUMBER
  WHERE EMPLOYEE_NUMBER CONTAINS 'AH'
ON TABLE SET PAGE NOPAGE
END
```
The output is:

<table>
<thead>
<tr>
<th>Employee Number</th>
<th>Start Date</th>
<th>Quarter</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH118</td>
<td>2013/01/15</td>
<td>1</td>
</tr>
<tr>
<td>AH288</td>
<td>2013/11/11</td>
<td>4</td>
</tr>
<tr>
<td>AH42</td>
<td>2008/11/13</td>
<td>4</td>
</tr>
<tr>
<td>AH928</td>
<td>2009/04/11</td>
<td>2</td>
</tr>
</tbody>
</table>

DTPART extracts the quarter from the employee start date:

\[
\text{DTPART(START\_DATE, \text{QUARTER})}
\]

For 2009/04/11, the result is 2.

**DTRUNC: Returning the Start of a Date Period for a Given Date**

Given a date or timestamp and a component, DTRUNC returns the first date within the period specified by that component.

**Syntax:**

\[
\text{DTRUNC(\text{date\_or\_timestamp}, \text{date\_period})}
\]

where:

- **date\_or\_timestamp**
  
  Date or date-time
  
  Is the date or timestamp of interest, which must provide a full component date.

- **date\_period**
  
  Is the period whose starting or ending date you want to find. Can be one of the following:

  - DAY, returns the date that represents the input date (truncates the time portion, if there is one).
  - YEAR, returns the date of the first day of the year.
  - MONTH, returns the date of the first day of the month.
  - QUARTER, returns the date of the first day in the quarter.
WEEK, returns the date that represents the first date of the given week.

By default, the first day of the week will be Sunday, but this can be changed using the WEEKFIRST parameter.

YEAR_END, returns the last date of the year.

QUARTER_END, returns the last date of the quarter.

MONTH_END, returns the last date of the month.

WEEK_END, returns the last date of the week.

**Example:** Returning the First Date in a Date Period

In the following request against the WF_RETAIL data source, DTRUNC returns the first date of the quarter given the start date of the employee:

```plaintext
DEFINE FILE WF_RETAILLITE
QTRSTART/YYMD = DTRUNC(START_DATE, QUARTER);
END
TABLE FILE WF_RETAILLITE
PRINT START_DATE QTRSTART AS 'Start,of Quarter'
BY EMPLOYEE_NUMBER
WHERE EMPLOYEE_NUMBER CONTAINS 'AH'
ON TABLE SET PAGE NOPAGE
END
```

The output is:

<table>
<thead>
<tr>
<th>Employee Number</th>
<th>Start Date</th>
<th>Start of Quarter</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH118</td>
<td>2013/01/15</td>
<td>2013/01/01</td>
</tr>
<tr>
<td>AH288</td>
<td>2013/11/11</td>
<td>2013/10/01</td>
</tr>
<tr>
<td>AH42</td>
<td>2008/11/13</td>
<td>2008/10/01</td>
</tr>
<tr>
<td>AH928</td>
<td>2009/04/11</td>
<td>2009/04/01</td>
</tr>
</tbody>
</table>

DTRUNC returns the first date of the quarter given the date of birth:

```plaintext
DTRUNC(DATE_OF_BIRTH, QUARTER)
```

For 1993/03/27, the result is 1993/03/01.
**Example:** Using the Start of Week Parameter for DTRUNC

The following request returns the date that is the start of the week for the start date of certain employees:

```
DEFINE FILE WF_RETAILLITE
DAY1/WT = DTRUNC(START_DATE, DAY);
WKSTART/YYMD = DTRUNC(START_DATE, WEEK);
DAY2/WT = DTRUNC(WKSTART, DAY);
END
TABLE FILE WF_RETAILLITE
PRINT START_DATE
DAY1 AS 'DOW 1'
WKSTART AS 'Start, of Week'
DAY2 AS 'DOW 2'
BY EMPLOYEE_NUMBER
WHERE START_DATE GT '20130101'
WHERE EMPLOYEE_NUMBER CONTAINS 'AH'
ON TABLE SET PAGE NOPAGE
END
```

The output is:

<table>
<thead>
<tr>
<th>Employee Number</th>
<th>Start Date</th>
<th>Start DOW 1</th>
<th>Start of Week</th>
<th>Start DOW 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH118</td>
<td>2013/01/15</td>
<td>TUE</td>
<td>2013/01/13</td>
<td>SUN</td>
</tr>
<tr>
<td>AH2272</td>
<td>2013/01/17</td>
<td>THU</td>
<td>2013/01/13</td>
<td>SUN</td>
</tr>
<tr>
<td>AH288</td>
<td>2013/11/11</td>
<td>MON</td>
<td>2013/11/10</td>
<td>SUN</td>
</tr>
<tr>
<td>AH3520</td>
<td>2013/09/23</td>
<td>MON</td>
<td>2013/09/22</td>
<td>SUN</td>
</tr>
<tr>
<td>AH3591</td>
<td>2013/09/22</td>
<td>SUN</td>
<td>2013/09/22</td>
<td>SUN</td>
</tr>
<tr>
<td>AH5177</td>
<td>2013/07/21</td>
<td>SUN</td>
<td>2013/07/21</td>
<td>SUN</td>
</tr>
</tbody>
</table>

DTRUNC returns the date that represents the start of the week.

**DTRUNC(START_DATE, WEEK)**

For 2013/01/15, the result is 2013/01/13
Example:  Returning the Date of the First and Last Days of a Week

The following request returns the dates that correspond to the first day of the week and the last day of the week for the given date.

```
DEFINE FILE WF_RETAILLITE
WEEKSTART/YYMD = DTRUNC(START_DATE, WEEK);
WEEKEND/YYMD = DTRUNC(START_DATE, WEEK_END);
END
TABLE FILE WF_RETAILLITE
PRINT START_DATE WEEKSTART AS 'Start, of Week'
WEEKEND AS 'End, of Week'
BY EMPLOYEE_NUMBER
WHERE EMPLOYEE_NUMBER CONTAINS 'AH1'
ON TABLE SET PAGE NOPAGE
END
```

The output is shown in the following image.

<table>
<thead>
<tr>
<th>Employee Number</th>
<th>Start Date</th>
<th>Start of Week</th>
<th>End of Week</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH118</td>
<td>2013/01/15</td>
<td>2013/01/13</td>
<td>2013/01/19</td>
</tr>
<tr>
<td>AH1994</td>
<td>2006/01/01</td>
<td>2006/01/01</td>
<td>2006/01/07</td>
</tr>
</tbody>
</table>

DTRUNC calculates the date of the end of the week.

```
WEEKEND/YYMD = DTRUNC(START_DATE, WEEK_END)
```

For 2013/01/15, the result is 2013/01/19.
Date Functions

Date functions manipulate date values. There are two types of date functions:

- Standard date functions for use with non-legacy dates.
- Legacy date functions for use with legacy dates.

If a date is in an alphanumeric or numeric field that contains date display options (for example, I6YMD), you must use the legacy date functions.

In this chapter:

- Overview of Date Functions
- Using Standard Date Functions
- 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
- DATETRAN: Formatting Dates in International Formats
- FIYR: Obtaining the Financial Year
- FIQTR: Obtaining the Financial Quarter
- FIYYQ: Converting a Calendar Date to a Financial Date
- TODAY: Returning the Current Date
- Using Legacy Date Functions
- AYMD: Adding or Subtracting Days
- CHGDAT: Changing How a Date String Displays
- 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
Overview of Date Functions

The following explains the difference between the types of date functions:

- **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, for example, 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.

- **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, 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.

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.

In addition to the functions discussed in this topic, there are date and time functions that are available only in the Maintain language. For information on these functions, see Maintain-specific Date and Time Functions.
For many functions, the output argument can be supplied either as a field name or as a format enclosed in single quotation marks. However, if a function is called from a Dialogue Manager command, this argument must always be supplied as a format, and if a function is called from a Maintain procedure, this argument must always be supplied as a field name. For detailed information about calling a function and supplying arguments, see Accessing and Calling a Function.

Using Standard Date Functions

When using standard date functions, you need to understand the settings that alter the behavior of these functions, as well as the acceptable formats and how to supply values in these formats.

You can affect the behavior of date functions in the following ways:

- Defining which days of the week are work days and which are not. Then, when you use a date function involving work days, dates that are not work days are ignored. For details, see Specifying Work Days on page 280.

- Determining whether to display leading zeros when a date function in Dialogue Manager returns a date. For details, see Enabling Leading Zeros For Date and Time Functions in Dialogue Manager on page 286.

For detailed information on each standard date function, see:

- **DATEADD**: Adding or Subtracting a Date Unit to or From a Date on page 287
- **DATECVT**: Converting the Format of a Date on page 291
- **DATEDIF**: Finding the Difference Between Two Dates on page 293
- **DATEMOV**: Moving a Date to a Significant Point on page 297
- **DATETRAN**: Formatting Dates in International Formats on page 304
- **DPART**: Extracting a Component From a Date Field
- **FIYR**: Obtaining the Financial Year on page 320
- **FIQTR**: Obtaining the Financial Quarter on page 322
- **FIYYQ**: Converting a Calendar Date to a Financial Date on page 325
- **TODAY**: Returning the Current Date on page 328
Specifying Work Days

You can determine which days are work days and which are not. Work days affect the DATEADD, DATEDIF, and DATEMOV functions. You identify work days as business days or holidays.

Specifying Business Days

Business days are traditionally Monday through Friday, but not every business has this schedule. For example, if your company does business on Sunday, Tuesday, Wednesday, Friday, and Saturday, you can tailor business day units to reflect that schedule.

Syntax: How to Set Business Days

```
SET BUSDAYS = smtwtfs
```

where:

`smtwtfs`

Is the seven character list of days that represents your business week. The list has a position for each day from Sunday to Saturday:

- To identify a day of the week as a business day, enter the first letter of that day in that day's position.
- To identify a non-business day, enter an underscore (_) in that day's position.

If a letter is not in its correct position, or if you replace a letter with a character other than an underscore, you receive an error message.

Example: Setting Business Days to Reflect Your Work Week

The following designates work days as Sunday, Tuesday, Wednesday, Friday, and Saturday:

```
SET BUSDAYS = S_TW_FS
```

Syntax: How to View the Current Setting of Business Days

```
? SET BUSDAYS
```

Specifying Holidays

You can specify a list of dates that are designated as holidays in your company. These dates are excluded when using functions that perform calculations based on working days. For example, if Thursday in a given week is designated as a holiday, the next working day after Wednesday is Friday.
To define a list of holidays, you must:

1. Create a holiday file using a standard text editor.
2. Select the holiday file by issuing the SET command with the HDAY parameter.

**Reference:** Rules for Creating a Holiday File

- Dates must be in YYMD format.
- Dates must be in ascending order.
- Each date must be on its own line.
- Each year for which data exists must be included or the holiday file is considered invalid. Calling a date function with a date value outside the range of the holiday file returns a zero for business day requests.

If you are subtracting two dates in 2005, and the latest date in the holiday file is 20041231, the subtraction will not be performed. One way to avoid invalidating the holiday file is to put a date very far in the future in any holiday file you create (for example, 29991231), and then it will always be considered valid.

- You may include an optional description of the holiday, separated from the date by a space.

By default, the holiday file has a file name of the form HDAYxxxx.err and is on your path, or on z/OS under PDS deployment, is a member named HDAYxxxx of a PDS allocated to DDNAME ERRORS. In your procedure or request, you must issue the SET HDAY=xxxx command to identify the file or member name. Alternatively, you can define the file to have any name and be stored anywhere or, on z/OS under PDS deployment, allocate the holiday file as a sequential file of any name or as member HDAYxxxx of any PDS. For information about using non-default holiday file names, see How to FILEDEF or DYNAM the Holiday File on page 283.

**Procedure:** How to Create a Holiday File

1. In a text editor, create a list of dates designated as holidays using the Rules for Creating a Holiday File on page 281.
2. Save the file.

If you are not using the default naming convention, see How to FILEDEF or DYNAM the Holiday File on page 283. If you are using the default naming convention, use the following instructions:

**In Windows and UNIX:** The file must be HDAYxxxx.ERF

**In z/OS:** The file must be a member of ERRORS named HDAYxxxx.

where:

*xxxx*

Is a string of text four characters long.

**Syntax:** How to Select a Holiday File

```
SET HDAY = xxxx
```

where:

*xxxx*

Is the part of the name of the holiday file after HDAY. This string must be four characters long.

**Example:** Creating and Selecting a Holiday File

The following is the HDAYTEST file, which establishes holidays:

```
19910325 TEST HOLIDAY
19911225 CHRISTMAS
```

The following sets HDAYTEST as the holiday file:

```
SET BUSDAYS = SMTWTF
SET HDAY = TEST
```

This request uses HDAYTEST in its calculations:

```
TABLE FILE MOVIES
PRINT TITLE RELDATE
COMPUTE NEXTDATE/YMD = DATEADD(RELDATE, 'BD', 1);
WHERE RELDATE GE '19910101';
END
```

The output is:
Syntax: How to FILEDEF or DYNAM the Holiday File

In all environments except z/OS under PDS deployment, use the following syntax.

```
FILEDEF HDAYxxxx DISK {app/|path}/filename.ext
```

where:

**HDAYxxxx**

Is the logical name (DDNAME) for the holiday file, where xxxx is any four characters. You establish this logical name by issuing the `SET HDAY=xxxx` command in your procedure or request.

**app**

Is the name of the application in which the holiday file resides.

**path**

Is the path to the holiday file.

**filename.ext**

Is the name of the holiday file.

On z/OS under PDS deployment, use the following to allocate a sequential holiday file.

```
DYNAM ALLOC {DD|FILE} HDAYxxxx DA qualif.filename.suffix SHR REU
```

On z/OS under PDS deployment, use the following to allocate a holiday file that is a member of a PDS.

```
DYNAM ALLOC {DD|FILE} HDAYxxxx DA qualif.filename.suffix(HDAYxxx) SHR REU
```

where:

**HDAYxxxx**

Is the DDNAME for the holiday file. Your FOCEXEC or request must set the HDAY parameter to xxxx, where xxxx is any four characters you choose. If your holiday file is a member of a PDS, HDAYxxxx must also be the member name.
qualif.filename.suffix

Is the fully-qualified name of the sequential file that contains the list of holidays or the PDS with member HDAYxxxx that contains the list of holidays.

**Example:** Defining a Holiday File

The following holiday file, named holiday.data in the c:\temp directory on Windows, defines November 3, 2011 and December 24, 2011 as holidays:

```
20111103
20111224
```

The following request against the MOVIES data source uses the FILEDEF command to define this file as the holiday file. The logical name in the FILEDEF command is HDAYMMMM, and the procedure issues the SET HDAY=MMMM command. It then defines the date November 2, 2011 and calculates the next business day:

```
FILEDEF HDAYMMMM DISK c:\ibi\holiday.data
SET HDAY = MMMM
SET BUSDAYS = _MTWTF_
DEFINE FILE MOVIES
NEWDATE/YYMD = '20111102';
NEXTDATE/YYMD = DATEADD(NEWDATE, 'BD', 1);
END
TABLE FILE MOVIES
SUM COPIES NEWDATE NEXTDATE
ON TABLE SET PAGE NOPAGE
END
```

The output shows that the next business day after November 2 is November 4 because November 3 is a holiday:

```
<table>
<thead>
<tr>
<th>COPIES</th>
<th>NEWDATE</th>
<th>NEXTDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>117</td>
<td>2011/11/02</td>
<td>2011/11/04</td>
</tr>
</tbody>
</table>
```

The following defines and sets the holiday file. Then DATEADD finds the next business day taking the holiday file into account:

```
FILEDEF HDAYMMMM DISK c:\ibi\holiday.data
SET HDAY = MMMM
SET BUSDAYS = _MTWTF_
DATEADD(NEWDATE, 'BD', 1);
```

For 2011/11/02, DATEADD returns 2011/11/04 because November 3 is a holiday.
Example: Allocating the Holiday File to a Sequential File on z/OS Under PDS Deployment

The following sequential file, named USER1.HOLIDAY.DATA, defines November 3, 2011 and December 24, 2011 as holidays:

```
20111103
20111224
```

The following request against the MOVIES data source uses the DYNAM command to allocate this file as the holiday file. The DDNAME in the DYNAM command is HDAYMMMM, and the procedure issues the SET HDAY=MMMM command. It then defines the date November 2, 2011 and calculates the next business day:

```
DYNAM ALLOC DD HDAYMMMM DA USER1.HOLIDAY.DATA SHR REU
SET HDAY = MMMM
SET BUSDAYS = _MTWTF_
DEFINE FILE MOVIES
NEWDATE/YYMD = '20111102';
NEXTDATE/YYMD = DATEADD(NEWDATE, 'BD', 1);
END
TABLE FILE MOVIES
SUM COPIES NEWDATE NEXTDATE
ON TABLE SET PAGE NOPAGE
END
```

The output shows that the next business day after November 2 is November 4 because November 3 is a holiday:

```
COPIES  NEWDATE     NEXTDATE
------  -------     --------
117  2011/11/02  2011/11/04
```

The following defines and sets the holiday file. Then DATEADD finds the next business day taking the holiday file into account:

```
DYNAM ALLOC DD HDAYMMMM DA USER1.HOLIDAY.DATA SHR REU
SET HDAY = MMMM
DATEADD(NEWDATE, 'BD', 1);
```

For 2011/11/02, DATEADD returns 2011/11/04 because November 3 is a holiday.

Example: Allocating the Holiday File to a PDS Member on z/OS Under PDS Deployment

The following holiday file, member HDAYMMMM in a PDS named USER1.HOLIDAY.DATA, defines November 3, 2011 and December 24, 2011 as holidays:

```
20111103
20111224
```
The following request against the MOVIES data source uses the DYNAM command to allocate this file as the holiday file. The DDNAME in the DYNAM command is HDAYMMMM, the member name is also HDAYMMMM, and the procedure issues the SET HDAY=MMMM command. It then defines the date November 2, 2011 and calculates the next business day:

```
DYNAM ALLOC DD HDAYMMMM DA USER1.HOLIDAY.DATA(HDAYMMMM) SHR REU
SET HDAY = MMMM
SET BUSDAYS = _MTWTFS_
DEFINE FILE MOVIES
NEWDATE/YYMD = '20111102';
NEXTDATE/YYMD = DATEADD(NEWDATE, 'BD', 1);
END
TABLE FILE MOVIES
SUM COPIES NEWDATE NEXTDATE
ON TABLE SET PAGE NOPAGE
END
```

The output shows that the next business day after November 2 is November 4 because November 3 is a holiday:

<table>
<thead>
<tr>
<th>COPIES</th>
<th>NEWDATE</th>
<th>NEXTDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>117</td>
<td>2011/11/02</td>
<td>2011/11/04</td>
</tr>
</tbody>
</table>

The following defines and sets the holiday file. Then DATEADD finds the next business day taking the holiday file into account:

```
DYNAM ALLOC DD HDAYMMMM DA USER1.HOLIDAY.DATA(HDAYMMMM) SHR REU
SET HDAY = MMMM
SET BUSDAYS = _MTWTFS_
DATEADD(NEWDATE, 'BD', 1);
```

For 2011/11/02, DATEADD returns 2011/11/04 because November 3 is a holiday.

### Enabling Leading Zeros For Date and Time Functions in Dialogue Manager

If you use a date and time function in Dialogue Manager that returns a numeric integer format, Dialogue Manager truncates any leading zeros. For example, if a function returns the value 000101 (indicating January 1, 2000), Dialogue Manager truncates the leading zeros, producing 101, an incorrect date. To avoid this problem, use the LEADZERO parameter.

LEADZERO only supports an expression that makes a direct call to a function. An expression that has nesting or another mathematical function always truncates leading zeros. For example,

```
-SET &OUT = AYM(&IN, 1, 'I4')/100;
```

truncates leading zeros regardless of the LEADZERO parameter setting.
**Syntax:**  How to Set the Display of Leading Zeros

SET LEADZERO = {ON | OFF}

where:

**ON**

Displays leading zeros if present.

**OFF**

Truncates leading zeros. OFF is the default value.

**Example:**  Displaying Leading Zeros

The AYM function adds one month to the input date of December 1999:

```plaintext
-SET &IN = '9912';
-RUN
-SET &OUT = AYM(&IN, 1, 'I4');
-TYPE &OUT
```

Using the default LEADZERO setting, this yields:

1

This represents the date January 2000 incorrectly. Setting the LEADZERO parameter in the request as follows:

```plaintext
SET LEADZERO = ON
-SET &IN = '9912';
-SET &OUT = AYM(&IN, 1, 'I4');
-TYPE &OUT
```

results in the following:

0001

This correctly indicates January 2000.

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

Available Languages: reporting, Maintain

The DATEADD function adds a unit to or subtracts a unit from 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, first DATEADD moves to the next weekday, Monday, then it adds a day. The result is Tuesday.

Business day. When using the business day unit, DATEADD uses the BUSDAYS parameter setting and holiday file to determine which days are working days and disregards the rest. If Monday is not a working day, then one business day past Sunday is Tuesday. See Specifying Holidays on page 280 for more information.

Note that when the DATEADD function calculates the next or previous business day or work day, it always starts from a business day or work day. So if the actual day is Saturday or Sunday, and the request wants to calculate the next business day, the function will use Monday as the starting day, not Saturday or Sunday, and will return Tuesday as the next business day. Similarly, when calculating the previous business day, it will use the starting day Friday, and will return Thursday as the previous business day. You can use the DATEMOV function to move the date to the correct type of day before using DATEADD. For more information, see DATEMOV: Moving a Date to a Significant Point on page 297.

DATEADD requires a date to be in date format. Since Dialogue Manager interprets a date as alphanumeric or numeric, and DATEADD requires a standard date stored as an offset from the base date, do not use DATEADD with Dialogue Manager unless you first convert the variable used as the input date to an offset from the base date.

For more information, see Calling a Function From a Dialogue Manager Command.

You add or subtract non day-based dates (for example, YM or YQ) directly without using DATEADD.

DATEADD works only with full component dates.

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

DATEADD(date, 'component', increment)

where:

date
  Date
  Is a full component date.
component

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.

increment

Integer

Is the number of date units added to or subtracted from 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 argument. It uses the format of the date argument 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, and so on), 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.

**Example:** Truncation With DATEADD

The number of units passed to DATEADD is always a whole unit. For example

```sql
DATEADD(DATE, 'M', 1.999)
```

adds one month because the number of units is less than two.

**Example:** Using the Weekday Unit

If you use the weekday unit and a Saturday or Sunday is the input date, DATEADD changes the input date to Monday. The function

```sql
DATEADD('910623', 'WD', 1)
```

in which DATE is either Saturday or Sunday yields Tuesday; Saturday and Sunday are not weekdays, so DATEADD begins with Monday and adds one.

Note that the single quotes around the number in the first argument, `910623`, causes it to be treated as a natural date literal.

**Example:** Adding Weekdays to a Date (Reporting)

DATEADD adds three weekdays to NEW_DATE. In some cases, it adds more than three days because HIRE_DATE_PLUS_THREE would otherwise be on a weekend.

```sql
TABLE FILE EMPLOYEE
PRINT FIRST_NAME AND HIRE_DATE AND COMPUTE
NEW_DATE/YYMD = HIRE_DATE;
HIRE_DATE_PLUS_THREE/YYMD = DATEADD(NEW_DATE, 'WD', 3);
BY LAST_NAME
WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>HIRE_DATE</th>
<th>NEW_DATE</th>
<th>HIRE_DATE_PLUS_THREE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>82/04/01</td>
<td>1982/04/01</td>
<td>1982/04/06</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>81/11/02</td>
<td>1981/11/02</td>
<td>1981/11/05</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>82/04/01</td>
<td>1982/04/01</td>
<td>1982/04/06</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>82/05/01</td>
<td>1982/05/01</td>
<td>1982/05/06</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>81/07/01</td>
<td>1981/07/01</td>
<td>1981/07/06</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>81/07/01</td>
<td>1981/07/01</td>
<td>1981/07/06</td>
</tr>
</tbody>
</table>
Example: Determining If a Date Is a Work Day (Reporting)

DATEADD determines which values in the TRANSDATE field do not represent work days by adding zero days to TRANSDATE using the business day unit. If TRANSDATE does not represent a business day, DATEADD returns the next business day to DATEX. TRANSDATE is then compared to DATEX, and the day of the week is printed for all dates that do not match between the two fields, resulting in a list of all non-work days.

```
DEFINE FILE VIDEOTRK
DATEX/YMD = DATEADD(TRANSDATE, 'BD', 0);
DATEINT/I8YYMD = DATECVT(TRANSDATE, 'YMD','I8YYMD');
END
TABLE FILE VIDEOTRK
SUM TRANSDATE NOPRINT
COMPUTE DAYNAME/A8 = DOWKL(DATEINT, DAYNAME); AS 'Day of Week'
BY TRANSDATE AS 'Date'
WHERE TRANSDATE NE DATEX
END
```

The output is:

<table>
<thead>
<tr>
<th>Date</th>
<th>Day of Week</th>
</tr>
</thead>
<tbody>
<tr>
<td>91/06/22</td>
<td>SATURDAY</td>
</tr>
<tr>
<td>91/06/23</td>
<td>SUNDAY</td>
</tr>
<tr>
<td>91/06/30</td>
<td>SUNDAY</td>
</tr>
</tbody>
</table>

Example: Adding Months to a Date (Maintain)

DATEADD adds months to the DATE1 field:

```
MAINTAIN
compute DATE1/yymd = '20000101'
compute DATE2/yymd= dateadd(date1, 'M', 2, date2);
type "DATE1 = <<DATE1 + 2 MONTHS =  DATE2 = <<DATE2"
END
```

The result is:

DATE1 = 2000/01/01+ 2 MONTHS =  DATE2 = 2000/03/01

DATECVT: Converting the Format of a Date

Available Languages: reporting, Maintain

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

DATECVT turns off optimization and compilation.
Note: You can use simple assignment instead of calling this function.

Syntax: How to Convert a Date Format

\[
\text{DATECVT}(\text{date}, \ 'in\_format', \ output)\\
\]

where:

\text{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.

\text{in\_format}

Alphanumeric

Is the format of the date enclosed in single quotation marks. It is one of the following:

- A 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). A non-date format in \text{in\_format} functions as an offset from the base date of a YYMD field (12/31/1900).

\text{output}

Alphanumeric

Is the output format enclosed in single quotation marks or a field containing the format. It is one of the following:

- A 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 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.

\[
\text{CHARDATE/}A13 = \text{DATECVT} (\text{NUMDATE}, \ 'I8YYMD', \ 'A8YYMD');
\]
Note: DATECVT does not use an output format; it uses the format of the argument output_format for the result.

Example: Converting a YYMD Date to DMY

DATECVT converts 19991231 to 311299 and stores the result in CONV_FIELD:

CONV_FIELD/DMY = DATECVT(19991231, 'I8YYMD', 'DMY');

or

CONV_FIELD/DMY = DATECVT('19991231', 'A8YYMD', 'DMY');

Example: Converting a Legacy Date to Date Format (Reporting)

DATECVT converts HIRE_DATE from I6YMD legacy date format to YYMD date format:

TABLE FILE EMPLOYEE
PRINT FIRST_NAME AND HIRE_DATE AND COMPUTE
NEW_HIRE_DATE/YYMD = DATECVT(HIRE_DATE, 'I6YMD', 'YYMD');
BY LAST_NAME
WHERE DEPARTMENT EQ 'MIS';
END

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>HIRE_DATE</th>
<th>NEW_HIRE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>82/04/01</td>
<td>1982/04/01</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>81/11/02</td>
<td>1981/11/02</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>82/04/01</td>
<td>1982/04/01</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>82/05/01</td>
<td>1982/05/01</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>81/07/01</td>
<td>1981/07/01</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>81/07/01</td>
<td>1981/07/01</td>
</tr>
</tbody>
</table>

DATEDIF: Finding the Difference Between Two Dates

Available Languages: reporting, Maintain

The DATEDIF function returns the difference between two full component standard dates in units of a specified component. 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 component 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 setting 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. See Rules for Creating a Holiday File on page 281 for more information.

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.

You can find the difference between non-day based dates (for example YM or YQ) directly without using DATEDIF.

Since Dialogue Manager interprets a date as alphanumeric or numeric, and DATEDIF requires a standard date stored as an offset from the base date, do not use DATEDIF with Dialogue Manager unless you first convert the variable used as the input date to an offset from the base date.

For more information, see Calling a Function From a Dialogue Manager Command.

DATEDIF works only with full component dates.
**Syntax:**  How to Find the Difference Between Two Dates

DATEDIF(from_date, to_date, 'component')

DATEDIF('from_date', 'to_date', 'component')

where:

*from_date*
  Date

  Is the start date from which to calculate the difference. Is a full component date.

*to_date*
  Date

  Is the end date from which to calculate the difference.

*component*
  Alphanumeric

  Is one of the following enclosed in single quotation marks:

  - *Y* indicates a year unit.
  - *M* indicates a month unit.
  - *D* indicates a day unit.
  - *WD* indicates a weekday unit.
  - *BD* indicates a business day unit.

**Note:** DATEDIF does not use an *output* argument 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:

\[ \text{MYDIFF/18} = \frac{\text{NEW}_\text{YYM}_\text{DT}/\text{YYM}}{\text{OLD}_\text{YYM}_\text{DT}}; \]

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

**Example:** **Truncation With DATEDIF**

DATEDIF calculates the difference between March 2, 1996, and March 1, 1997, and returns a zero because the difference is less than a year:

\[ \text{DATEDIF('19960302', '19970301', 'Y')} \]

**Example:** **Using Month Calculations**

The following expressions return a result of minus one month:

\[ \text{DATEDIF('19990228', '19990128', 'M')} \]
\[ \text{DATEDIF('19990228', '19990129', 'M')} \]
\[ \text{DATEDIF('19990228', '19990130', 'M')} \]
\[ \text{DATEDIF('19990228', '19990131', 'M')} \]

Additional examples:

\[ \text{DATEDIF( 'March 31 2001', 'May 31 2001', 'M')} \text{ yields 2.} \]
\[ \text{DATEDIF( 'March 31 2001', 'May 30 2001', 'M')} \text{ yields 1 (because May 30 is not the end of the month).} \]
\[ \text{DATEDIF( 'March 31 2001', 'April 30 2001', 'M')} \text{ yields 1.} \]
**Example:** Finding the Number of Weekdays Between Two Dates (Reporting)

DATECVT converts the legacy dates in HIRE_DATE and DAT_INC to the date format YYMD. DATEDIF then uses those date formats to determine the number of weekdays between NEW_HIRE_DATE and NEW_DAT_INC:

```
TABLE FILE EMPLOYEE
PRINT FIRST_NAME AND
COMPUTE NEW_HIRE_DATE/YYMD = DATECVT(HIRE_DATE, 'I6YMD', 'YYMD'); AND
COMPUTE NEW_DAT_INC/YYMD = DATECVT(DAT_INC, 'I6YMD', 'YYMD'); AND
COMPUTE WDAYS_HIRED/I8 = DATEDIF(NEW_HIRE_DATE, NEW_DAT_INC, 'WD');
BY LAST_NAME
IF WDAYS_HIRED NE 0
WHERE DEPARTMENT EQ 'PRODUCTION';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>NEW_HIRE_DATE</th>
<th>NEW_DAT_INC</th>
<th>WDAYS_HIRED</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>1982/01/04</td>
<td>1982/05/14</td>
<td>94</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>1982/02/02</td>
<td>1982/05/14</td>
<td>73</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>1982/01/04</td>
<td>1982/05/14</td>
<td>94</td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>1980/06/02</td>
<td>1982/01/01</td>
<td>414</td>
</tr>
<tr>
<td>ALFRED</td>
<td>1980/06/02</td>
<td>1981/01/01</td>
<td>153</td>
<td></td>
</tr>
</tbody>
</table>

**Example:** Finding the Number of Years Between Two Dates (Maintain)

DATEDIF determines the number of years between DATE2 and DATE1:

```
MAINTAIN
Case Top
compute DATE1/ymd = '20020717';
compute DATE2/ymd = '19880705';
compute DIFF/I3= DATEDIF(DATE2, DATE1, 'Y', DIFF);
type "<<DATE1  -  <<DATE2  = <DIFF  YEARS"
ENDCASE
END
```

The result is:

```
2002/07/17 - 1988/07/05 = 14 YEARS
```

**DATEMOV: Moving a Date to a Significant Point**

Available Languages: reporting, Maintain

The DATEMOV function moves a date to a significant point on the calendar.
Note: Using the beginning of week point (BOW) will always return Monday, and using the end of week point (EOW) will always return Friday. Also, if the date used with the DATEMOV function falls on Saturday or Sunday, the actual date used by the function will be the moved forward to the next Monday. If you do not want to do the calculation by moving the date from Saturday or Sunday to Monday, or if you want the BOW to be Sunday and the EOW to be Saturday, you can use the DTRUNC function.

Since Dialogue Manager interprets a date as alphanumeric or numeric, and DATEMOV requires a standard date stored as an offset from the base date, do not use DATEMOV with Dialogue Manager unless you first convert the variable used as the input date to an offset from the base date. For example, the following converts the integer legacy date 20050131 to a smart date, adds one month, and converts the result to an alphanumeric legacy date:

-SET &STRT=DATECVT(20050131,'I8YYMD', 'YYMD');
-SET &NMT=DATEADD(&STRT,'M',1);
-SET &NMTA=DATECVT(&NMT,'YYMD','A8MTDYY');
-TYPE A MONTH FROM 20050131 IS &NMTA

The output shows that the DATEADD function added the actual number of days in the month of February to get to the end of the month from the end of January:

A MONTH FROM 20050131 IS 02282005

For more information, see Calling a Function From a Dialogue Manager Command.

DATEMOV works only with full component dates.

**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 enclosed in single quotation marks ('). An invalid point results in a return code of zero. Valid values are:

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

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

Note that when the DATEADD function calculates the next or previous business day or work day, it always starts from a business day or work day. So if the actual day is Saturday or Sunday, and the request wants to calculate the next business day, the function will use Monday as the starting day, not Saturday or Sunday, and will return Tuesday as the next business day. Similarly, when calculating the previous business day, it will use the starting day Friday, and will return Thursday as the previous business day.

To avoid skipping a business day or work day, use DATEMOV. To return the next business or work day, use BD- or WD- to first move to the previous business or work day (if it is already a business day or work day, it will not be moved). Then use DATEADD to move to the next business or work day. If you want to return the previous business or work day, first use BD+ or WD+ to move to the next business or work day (if it is already the correct type of day, it will not be moved). Then use DATEADD to return the previous business or work day.
**Note:** DATEMOV does not use an *output* argument. It uses the format of the *date* argument 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.

**Example:** Returning the Next Business Day

This example shows why you may need to use DATEMOV to get the correct result.

The following request against the GGSALES data source uses the BD (Business Day) move point against the DATE field. First DATE is converted to a smart date, then DATEADD is called with the BD move-point:

```
DEFINE FILE GGSALES
DT1/WMDYY=DATE;
DT2/WMDYY = DATEADD(DT1 , 'BD', 1);
DAY/Dt = DT1;
END

TABLE FILE GGSALES
SUM  DT1
DT2
BY DT1 NOPRINT
WHERE RECORDLIMIT EQ 10
END
```

When the date is on a Saturday or Sunday on the output, the next business day is returned as a Tuesday. This is because before doing the calculation, the original date was moved to a business day:

<table>
<thead>
<tr>
<th>DT1</th>
<th>DT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUN, 09/01/1996</td>
<td>TUE, 09/03/1996</td>
</tr>
<tr>
<td>FRI, 11/01/1996</td>
<td>MON, 11/04/1996</td>
</tr>
<tr>
<td>SUN, 12/01/1996</td>
<td>TUE, 12/03/1996</td>
</tr>
<tr>
<td>SAT, 03/01/1997</td>
<td>TUE, 03/04/1997</td>
</tr>
<tr>
<td>TUE, 04/01/1997</td>
<td>WED, 04/02/1997</td>
</tr>
<tr>
<td>THU, 05/01/1997</td>
<td>FRI, 05/02/1997</td>
</tr>
<tr>
<td>SUN, 06/01/1997</td>
<td>TUE, 06/03/1997</td>
</tr>
<tr>
<td>MON, 09/01/1997</td>
<td>TUE, 09/02/1997</td>
</tr>
<tr>
<td>WED, 10/01/1997</td>
<td>THU, 10/02/1997</td>
</tr>
</tbody>
</table>
In the following version of the request, DATEMOV is called to make sure the starting day is a business day. The move point specified in the first call is BD- which only moves the date to the prior business day if it is not already a business day. The call to DATEADD then uses the BD move point to return the next business day:

```assembler
DEFINE FILE GGSALES
DT1/WMDYY=DATE;
DT1A/WMDYY=DATEMOV(DT1, 'BD-');
DT2/WMDYY = DATEADD(DT1A,'BD',1);
DAY/Dt = DT1;
END
```

TABLE FILE GGSALES
SUM  DT1 DT1A DT2
BY DT1 NOPRINT
WHERE RECORDLIMIT EQ 10
END

On the output, the next business day after a Saturday or Sunday is now returned as Monday:

<table>
<thead>
<tr>
<th>DT1</th>
<th>DT1A</th>
<th>DT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUN, 09/01/96</td>
<td>FRI, 08/30/96</td>
<td>MON, 09/02/96</td>
</tr>
<tr>
<td>FRI, 11/01/96</td>
<td>FRI, 11/01/96</td>
<td>MON, 11/04/96</td>
</tr>
<tr>
<td>SUN, 12/01/96</td>
<td>FRI, 11/29/96</td>
<td>MON, 12/02/96</td>
</tr>
<tr>
<td>SAT, 03/01/97</td>
<td>FRI, 02/28/97</td>
<td>MON, 03/03/97</td>
</tr>
<tr>
<td>TUE, 04/01/97</td>
<td>TUE, 04/01/97</td>
<td>WED, 04/02/97</td>
</tr>
<tr>
<td>THU, 05/01/97</td>
<td>THU, 05/01/97</td>
<td>FRI, 05/02/97</td>
</tr>
<tr>
<td>SUN, 06/01/97</td>
<td>FRI, 05/30/97</td>
<td>MON, 06/02/97</td>
</tr>
<tr>
<td>MON, 09/01/97</td>
<td>MON, 09/01/97</td>
<td>TUE, 09/02/97</td>
</tr>
<tr>
<td>WED, 10/01/97</td>
<td>WED, 10/01/97</td>
<td>THU, 10/02/97</td>
</tr>
</tbody>
</table>

**Example:** Using a DEFINE FUNCTION to Move a Date to the Beginning of the Week

The following DEFINE FUNCTION named BOWK takes a date and the name of the day you want to consider the beginning of the week and returns a date that corresponds to the beginning of the week:

```assembler
DEFINE FUNCTION BOWK(THEDATE/MDYY,WEEKSTART/A10)
DAYOFWK=W=THEDATE;
DAYNO/I1=IF DAYOFWK EQ 7 THEN 0 ELSE DAYOFWK;
FIRSTOFWK/I1=DECODE WEEKSTART('SUNDAY' 0 'MONDAY' 1 'TUESDAY' 2 'WEDNESDAY' 3 'THURSDAY' 4 'FRIDAY' 5 'SATURDAY' 6 'SUN' 0 'MON' 1 'TUE' 2 'WED' 3 'THU' 4 'FRI' 5 'SAT' 6);
BOWK/MDYY=IF DAYNO GE FIRSTOFWK THEN THEDATE-DAYNO+FIRSTOFWK ELSE THEDATE-7-DAYNO+FIRSTOFWK;
END
```
The following request uses the BOWK function to use return a date (DT2) that corresponds to the beginning of the week for each value of the DT1 field:

```
DEFINE FILE GGSALES
DT1/WMDYY=DATE;
DT2/WMDYY = BOWK(DT1 ,'SUN');
END

TABLE FILE GGSALES
SUM  DT1
DT2
BY DT1 NOPRINT
WHERE RECORDLIMIT EQ 10
ON TABLE SET PAGE NOLEAD
END
```

The output is shown in the following image:

<table>
<thead>
<tr>
<th>DT1</th>
<th>DT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUN, 09/01/1996</td>
<td>SUN, 09/01/1996</td>
</tr>
<tr>
<td>FRI, 11/01/1996</td>
<td>SUN, 10/27/1996</td>
</tr>
<tr>
<td>SUN, 12/01/1996</td>
<td>SUN, 12/01/1996</td>
</tr>
<tr>
<td>SAT, 03/01/1997</td>
<td>SUN, 02/23/1997</td>
</tr>
<tr>
<td>TUE, 04/01/1997</td>
<td>SUN, 03/30/1997</td>
</tr>
<tr>
<td>THU, 05/01/1997</td>
<td>SUN, 04/27/1997</td>
</tr>
<tr>
<td>SUN, 06/01/1997</td>
<td>SUN, 06/01/1997</td>
</tr>
<tr>
<td>MON, 09/01/1997</td>
<td>SUN, 08/31/1997</td>
</tr>
<tr>
<td>WED, 10/01/1997</td>
<td>SUN, 09/28/1997</td>
</tr>
</tbody>
</table>
Example: Determining Significant Points for a Date (Reporting)

The BUSDAYS parameter sets the business days to Monday, Tuesday, Wednesday, and Thursday. DATECVT converts the legacy date HIRE_DATE to the date format YYMD and provides date display options. DATEMOV then determines significant points for HIRE_DATE.

```
SET BUSDAY = _MTWT_
TABLE FILE EMPLOYEE
PRINT
COMPUTE NEW_DATE/YYMD = DATECVT(HIRE_DATE, 'I6YMD', 'YYMD'); AND
COMPUTE NWD/WT = DATEMOV(NEW_DATE, 'NWD'); AND
COMPUTE PWD/WT = DATEMOV(NEW_DATE, 'PWD'); AND
COMPUTE WDP/WT = DATEMOV(NEW_DATE, 'WD+'); AS 'WD+' AND
COMPUTE WDM/WT = DATEMOV(NEW_DATE, 'WD-'); AS 'WD-' AND
COMPUTE NBD/WT = DATEMOV(NEW_DATE, 'NBD'); AND
COMPUTE PBD/WT = DATEMOV(NEW_DATE, 'PBD'); AND
COMPUTE WBP/WT = DATEMOV(NEW_DATE, 'BD+'); AS 'BD+' AND
COMPUTE WBM/WT = DATEMOV(NEW_DATE, 'BD-'); AS 'BD-' BY LAST_NAME NOPRINT
HEADING
"Examples of DATEMOV"
"Business days are Monday, Tuesday, Wednesday, + Thursday ",
" "
"START DATE.. | MOVE POINTS..........................."
WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

Examples of DATEMOV
Business days are Monday, Tuesday, Wednesday, + Thursday
START DATE.. | MOVE POINTS...........................
NEW_DATE    DOW  NWD  PWD  WD+  WD-  NBD  PBD  BD+  BD-
--------    ---  ---  ---  ---  ---  ---  ---  ---  ---
1982/04/01  THU  FRI  WED  THU  THU  MON  WED  THU  THU
1981/11/02  MON  TUE  FRI  MON  MON  TUE  THU  MON  MON
1982/04/01  THU  FRI  WED  THU  THU  MON  WED  THU  THU
1982/05/01  SAT  TUE  THU  MON  FRI  TUE  WED  MON  THU
1981/07/01  WED  THU  TUE  WED  WED  THU  TUE  WED  WED
1981/07/01  WED  THU  TUE  WED  WED  THU  TUE  WED  WED

Example: Determining the End of the Week (Reporting)

DATEMOV determines the end of the week for each date in NEW_DATE and stores the result in EOW:

```
TABLE FILE EMPLOYEE
PRINT FIRST_NAME AND
COMPUTE NEW_DATE/YYMDWT = DATECVT(HIRE_DATE, 'I6YMD', 'YYMDWT'); AND
COMPUTE EOW/YYMDWT = DATEMOV(NEW_DATE, 'EOW');
BY LAST_NAME WHERE DEPARTMENT EQ 'PRODUCTION';
END
```
The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>NEW_DATE</th>
<th>EOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>1982 AUG 1</td>
<td>SUN 1982 AUG 6, FRI</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>1982 JAN 4</td>
<td>MON 1982 JAN 8, FRI</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>1982 FEB 2</td>
<td>TUE 1982 FEB 5, FRI</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>1982 JUL 1</td>
<td>THU 1982 JUL 2, FRI</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>1982 JAN 4</td>
<td>MON 1982 JAN 8, FRI</td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>1980 JUN 2</td>
<td>MON 1980 JUN 6, FRI</td>
</tr>
</tbody>
</table>

**Example:**  
Determining the End of the Week (Maintain)

DATEMOV determines the end of the week for each date:

MAINTAIN
COMPUTE X/YYMDWT='20020717';
COMPUTE Y/YYMDWT=DATEMOV(X, 'EOW', Y);
TYPE "<<X <<Y END OF WEEK "
END

The result is:

2002/07/17, WED  2002/07/19, FRI END OF WEEK

**DATETRAN: Formatting Dates in International Formats**

Available Languages: reporting, Maintain

The DATETRAN function formats dates in international formats.

**Syntax:**  
How to Format Dates in International Formats

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

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.

The following table shows the single component input types:

<table>
<thead>
<tr>
<th>Single Component Input Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>' (W) '</td>
<td>Day of week component only (original format must have only W component).</td>
</tr>
<tr>
<td>' (M) '</td>
<td>Month component only (original format must have only M component).</td>
</tr>
</tbody>
</table>

The following table shows the two-component input types:

<table>
<thead>
<tr>
<th>Two-Component Input Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>' (YYM) '</td>
<td>Four-digit year followed by month.</td>
</tr>
<tr>
<td>' (YM) '</td>
<td>Two-digit year followed by month.</td>
</tr>
<tr>
<td>' (MYY) '</td>
<td>Month component followed by four-digit year.</td>
</tr>
<tr>
<td>' (MY) '</td>
<td>Month component followed by two-digit year.</td>
</tr>
</tbody>
</table>

The following table shows the three-component input types:

<table>
<thead>
<tr>
<th>Three-Component Input Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>' (YYMD) '</td>
<td>Four-digit year followed by month followed by day.</td>
</tr>
<tr>
<td>' (YMD) '</td>
<td>Two-digit year followed by month followed by day.</td>
</tr>
<tr>
<td>' (DMYY) '</td>
<td>Day component followed by month followed by four-digit year.</td>
</tr>
</tbody>
</table>
Three-Component Input Type | Description
--- | ---
'(DMY)' | Day component followed by month followed by two-digit year.
'(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.
  
  **Note:** Zero suppression replaces initial zeros with blanks spaces.

- 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 listed in the following table. Note that the initial zero is replaced by a blank space:

<table>
<thead>
<tr>
<th>Format Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>Zero-suppresses months (displays numeric months before October as 1 through 9 rather than 01 through 09).</td>
</tr>
<tr>
<td>Format Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>d</td>
<td>Displays days before the tenth of the month as 1 through 9 rather than 01 through 09.</td>
</tr>
<tr>
<td>dp</td>
<td>Displays days before the tenth of the month as 1 through 9 rather than 01 through 09 with a period after the number.</td>
</tr>
<tr>
<td>do</td>
<td>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.</td>
</tr>
</tbody>
</table>

The following table shows valid month and day name translation options:

<table>
<thead>
<tr>
<th>Format Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Displays month as an abbreviated name, with no punctuation, all uppercase.</td>
</tr>
<tr>
<td>TR</td>
<td>Displays month as a full name, all uppercase.</td>
</tr>
<tr>
<td>Tp</td>
<td>Displays month as an abbreviated name, followed by a period, all uppercase.</td>
</tr>
<tr>
<td>t</td>
<td>Displays month as an abbreviated name with no punctuation. The name is all lowercase or initial uppercase, depending on language code.</td>
</tr>
<tr>
<td>tr</td>
<td>Displays month as a full name. The name is all lowercase or initial uppercase, depending on language code.</td>
</tr>
<tr>
<td>tp</td>
<td>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).</td>
</tr>
<tr>
<td>Format Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>W</td>
<td>Includes an abbreviated day-of-the-week name at the start of the displayed date, all uppercase with no punctuation.</td>
</tr>
<tr>
<td>WR</td>
<td>Includes a full day-of-the-week name at the start of the displayed date, all uppercase.</td>
</tr>
<tr>
<td>WP</td>
<td>Includes an abbreviated day-of-the-week name at the start of the displayed date, all uppercase, followed by a period.</td>
</tr>
<tr>
<td>w</td>
<td>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).</td>
</tr>
<tr>
<td>wr</td>
<td>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).</td>
</tr>
<tr>
<td>wp</td>
<td>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).</td>
</tr>
<tr>
<td>X</td>
<td>Includes an abbreviated day-of-the-week name at the end of the displayed date, all uppercase with no punctuation.</td>
</tr>
<tr>
<td>XR</td>
<td>Includes a full day-of-the-week name at the end of the displayed date, all uppercase.</td>
</tr>
</tbody>
</table>
The following table shows valid date delimiter options:

<table>
<thead>
<tr>
<th>Format Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>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.</td>
</tr>
<tr>
<td>.</td>
<td>Uses a period (.) as the component delimiter.</td>
</tr>
<tr>
<td>–</td>
<td>Uses a minus sign (-) as the component delimiter. This is the default when the conditions for a blank default delimiter are not satisfied.</td>
</tr>
<tr>
<td>Format Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>/</td>
<td>Uses a slash (/) as the component delimiter.</td>
</tr>
<tr>
<td></td>
<td>Omits component delimiters.</td>
</tr>
<tr>
<td>K</td>
<td>Uses appropriate Asian characters as component delimiters.</td>
</tr>
<tr>
<td>c</td>
<td>Places a comma (,) after the month name (following T, Tp, TR, t, tp, or tr).</td>
</tr>
<tr>
<td></td>
<td>Places a comma and blank after the day name (following W, Wp, WR, w, wp, or wr).</td>
</tr>
<tr>
<td></td>
<td>Places a comma and blank before the day name (following X, XR, x, or xr).</td>
</tr>
<tr>
<td>e</td>
<td>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.</td>
</tr>
<tr>
<td>D</td>
<td>Inserts a comma (,) after the day number and before the general delimiter character specified.</td>
</tr>
<tr>
<td>Y</td>
<td>Inserts a comma (,) after the year and before the general delimiter character specified.</td>
</tr>
</tbody>
</table>

**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
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

Alphanumeric

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

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 selected. 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 Server and Client must be configured with the correct code pages.

- If you use a terminal emulator program, it must be set to use a code page that can display the accent marks and characters in the translated dates. You may not be able to display dates translated into European and Asian characters at the same time. Similarly, if you want to print the translated dates, your printer must be capable of printing the required characters.

- The DATETRAN function is not supported in Dialogue Manager.
**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 ;
DATEEYMD/YYMDW=TRANS1 ;
DATEEYMD2/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
TABLE HOLD FORMAT HTML
ON TABLE SET PAGE-NUM OFF
END
```
The output is:

```
<table>
<thead>
<tr>
<th>FORMAT Btdp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blank-delimited (B)</td>
</tr>
<tr>
<td>Jan 4th 2005</td>
</tr>
</tbody>
</table>
```

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
TABLE HOLD FORMAT HTML
ON TABLE SET PAGE-NUM OFF
END
```
The output is:

<table>
<thead>
<tr>
<th>FORMAT Btdp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blank-delimited (B)</td>
</tr>
<tr>
<td>Abbreviated month name, default case (t)</td>
</tr>
<tr>
<td>Zero-suppress day number, end with period (dp)</td>
</tr>
<tr>
<td>German</td>
</tr>
</tbody>
</table>


The following request prints a blank delimited date in French, with a full day name at the beginning and a full month name, in lowercase (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
TABLE HOLD FORMAT HTML
ON TABLE SET PAGE-NUM OFF
END
The output is:

<table>
<thead>
<tr>
<th>FORMAT Bwrctrde</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blank-delimited (B)</td>
</tr>
<tr>
<td>Full day of week name at beginning of date, default case (wr)</td>
</tr>
<tr>
<td>Comma after day name (c)</td>
</tr>
<tr>
<td>Full month name, default case (tr)</td>
</tr>
<tr>
<td>Zero-suppress day number (d)</td>
</tr>
<tr>
<td>de between day and month and between month and year (e)</td>
</tr>
<tr>
<td>Spanish</td>
</tr>
</tbody>
</table>

martes, 4 de enero de 2005   miércoles, 2 de marzo de 2005

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 NOPRINTON
TABLE HOLD FORMAT HTML
ON TABLE SET PAGE-NUM OFF
END
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
TABLE HOLD FORMAT HTML
ON TABLE SET PAGE-NUM OFF
END

The output is:

<table>
<thead>
<tr>
<th>Date</th>
<th>Date in Greek</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005年1月4日</td>
<td>2005年3月2日</td>
</tr>
</tbody>
</table>

The output is:
The output is:

```
FORMAT Bwrcr
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
```

| Τρίτη, 04 Ιανουάριος 2005 | Τετάρτη, 02 Μάρτιος 2005 |

**FIYR: Obtaining the Financial Year**

The FIYR function returns the financial year, also known as the fiscal year, corresponding to a given calendar date based on the financial year starting date and the financial year numbering convention.

Since Dialogue Manager interprets a date as alphanumeric or numeric, and FIYR requires a standard date stored as an offset from the base date, do not use FIYR with Dialogue Manager unless you first convert the variable used as the input date to an offset from the base date.

For more information, see *Calling a Function From a Dialogue Manager Command*.

**Syntax:** **How to Obtain the Financial Year**

```
FIYR(inputdate, lowcomponent, startmonth, startday, yrnumbering, output)
```

where:

- **inputdate**
  
  Date
  
  Is the date for which the financial year is returned. The date must be a standard date stored as an offset from the base date.
  
  If the financial year does not begin on the first day of a month, the date must have Y(Y), M, and D components, or Y(Y) and JUL components (note that JUL is equivalent to YJUL). Otherwise, the date only needs Y(Y) and M components or Y(Y) and Q components.

- **lowcomponent**
  
  Alphanumeric
Is one of the following:

- **D** if the date contains a D or JUL component.
- **M** if the date contains an M component, but no D component.
- **Q** if the date contains a Q component.

**startmonth**

Numeric

1 through 12 are used to represent the starting month of the financial year, where 1 represents January and 12 represents December. If the low component is Q, the start month must be 1, 4, 7, or 10.

**startday**

Numeric

Is the starting day of the starting month, usually 1. If the low component is M or Q, 1 is required.

**yrnumbering**

Alphanumeric

Valid values are:

- **FYE** to specify the *Financial Year Ending* convention. The financial year number is the calendar year of the ending date of the financial year. For example, when the financial year starts on October 1, 2008, the date 2008 November 1 is in FY 2009 Q1 because that date is in the financial year that ends on 2009 September 30.

- **FYS** to specify the *Financial Year Starting* convention. The financial year number is the calendar year of the starting date of the financial year. For example, when the financial year starts on April 6, 2008, the date 2008 July 6 is in FY 2008 Q2 because that date is in the financial year that starts on 2008 April 6.

**output**

I, Y, or YY

The result will be in integer format, or Y or YY. This function returns a year value. In case of an error, zero is returned.

**Note:** February 29 cannot be used as a start day for a financial year.
Example: Obtaining the Financial Year

The following request against the CENTSTM data source obtains the financial year corresponding to an account period (field PERIOD, format YYM) and returns the values in each of the supported formats: Y, YY, and I4.

```
DEFINE FILE CENTSTM
FISCALYY/YY=FIYR(PERIOD,'M', 4,1,'FYE',FISCALYY);
FISCALY/Y=FIYR(PERIOD,'M', 4,1,'FYE',FISCALY);
FISCALI/I4=FIYR(PERIOD,'M', 4,1,'FYE',FISCALI);
END

TABLE FILE CENTSTM
PRINT PERIOD FISCALYY FISCALY FISCALI
BY GL_ACCOUNT
WHERE GL_ACCOUNT LT '2100'
END
```

On the output, note that the period April 2002 (2002/04) is in fiscal year 2003 because the starting month is April (4), and the FYE numbering convention is used:

<table>
<thead>
<tr>
<th>Ledger</th>
<th>Account</th>
<th>PERIOD</th>
<th>FISCALYY</th>
<th>FISCALY</th>
<th>FISCALI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1000</td>
<td>2002/01</td>
<td>2002</td>
<td>02</td>
<td>2002</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2002/02</td>
<td>2002</td>
<td>02</td>
<td>2002</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2002/03</td>
<td>2002</td>
<td>02</td>
<td>2002</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2002/04</td>
<td>2003</td>
<td>03</td>
<td>2003</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2002/05</td>
<td>2003</td>
<td>03</td>
<td>2003</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2002/06</td>
<td>2003</td>
<td>03</td>
<td>2003</td>
</tr>
<tr>
<td></td>
<td>2000</td>
<td>2002/01</td>
<td>2002</td>
<td>02</td>
<td>2002</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2002/02</td>
<td>2002</td>
<td>02</td>
<td>2002</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2002/03</td>
<td>2002</td>
<td>02</td>
<td>2002</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2002/04</td>
<td>2003</td>
<td>03</td>
<td>2003</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2002/05</td>
<td>2003</td>
<td>03</td>
<td>2003</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2002/06</td>
<td>2003</td>
<td>03</td>
<td>2003</td>
</tr>
</tbody>
</table>

FIYR obtains the financial year for PERIOD, which has format YYM:

```
FIYR(PERIOD, 'M', 4,1,'FYE','YY');
```

For PERIOD 2002/03, the result is 2002

For PERIOD 2002/04, the result is 2003.

FIQTR: Obtaining the Financial Quarter

The FIQTR function returns the financial quarter corresponding to a given calendar date based on the financial year starting date and the financial year numbering convention.
Since Dialogue Manager interprets a date as alphanumeric or numeric, and FIQTR requires a standard date stored as an offset from the base date, do not use FIQTR with Dialogue Manager unless you first convert the variable used as the input date to an offset from the base date.

For more information, see *Calling a Function From a Dialogue Manager Command*.

**Syntax:**

**How to Obtain the Financial Quarter**

\[ \text{FIQTR}(\text{inputdate}, \text{lowcomponent}, \text{startmonth}, \text{startday}, \text{yrnumbering}, \text{output}) \]

where:

- **inputdate**
  - Date
  - Is the date for which the financial year is returned. The date must be a standard date stored as an offset from the base date.
  - If the financial year does not begin on the first day of a month, the date must have Y(Y), M, and D components, or Y(Y) and JUL components (note that JUL is equivalent to YJUL). Otherwise, the date only needs Y(Y) and M components or Y(Y) and Q components.

- **lowcomponent**
  - Alphanumeric
  - Is one of the following:
    - D if the date contains a D or JUL component.
    - M if the date contains an M component, but no D component.
    - Q if the date contains a Q component.

- **startmonth**
  - Numeric
  - 1 through 12 are used to represent the starting month of the financial year, where 1 represents January and 12 represents December. If the low component is Q, the start month must be 1, 4, 7, or 10.

- **startday**
  - Numeric
  - Is the starting day of the starting month, usually 1. If the low component is M or Q, 1 is required.
yrnumbering

Alphanumeric

Valid values are:

FYE to specify the Financial Year Ending convention. The financial year number is the calendar year of the ending date of the financial year. For example, when the financial year starts on October 1, 2008, the date 2008 November 1 is in FY 2009 Q1 because that date is in the financial year that ends on 2009 September 30.

FYS to specify the Financial Year Starting convention. The financial year number is the calendar year of the starting date of the financial year. For example, when the financial year starts on April 6, 2008, the date 2008 July 6 is in FY 2008 Q2 because that date is in the financial year that starts on 2008 April 6.

output

I or Q

The result will be in integer format, or Q. This function will return a value of 1 through 4. In case of an error, zero is returned.

Note: February 29 cannot be used as a start day for a financial year.

Example: Obtaining the Financial Quarter

The following request against the CENTHR data source obtains the financial quarter corresponding to an employee starting date (field START_DATE, format YYMD) and returns the values in each of the supported formats: Q and I1.

```
DEFINE FILE CENTHR
FISCALQ/Q=FIQTR(START_DATE,'D',10,1,'FYE',FISCALQ);
FISCALI/I1=FIQTR(START_DATE,'D',10,1,'FYE',FISCALI);
END

TABLE FILE CENTHR
PRINT START_DATE FISCALQ FISCALI
BY LNAME BY FNAME
WHERE LNAME LIKE 'C%'
END
```
On the output, note that the date November 12, 1998 (1998/11/12) is in fiscal quarter Q1 because the starting month is October (10):

<table>
<thead>
<tr>
<th>Last Name</th>
<th>First Name</th>
<th>Starting Date</th>
<th>FISCALQ</th>
<th>FISCALI</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARNEY</td>
<td>ROSS</td>
<td>1998/09/12</td>
<td>Q4</td>
<td></td>
</tr>
<tr>
<td>CHIEN</td>
<td>CHRISTINE</td>
<td>1997/10/01</td>
<td>Q1</td>
<td></td>
</tr>
<tr>
<td>CLEVELAND</td>
<td>PHILIP</td>
<td>1996/07/30</td>
<td>Q4</td>
<td></td>
</tr>
<tr>
<td>CLINE</td>
<td>STEPHEN</td>
<td>1998/11/12</td>
<td>Q1</td>
<td></td>
</tr>
<tr>
<td>COHEN</td>
<td>DANIEL</td>
<td>1997/10/05</td>
<td>Q1</td>
<td></td>
</tr>
<tr>
<td>CORRIVEAU</td>
<td>RAYMOND</td>
<td>1997/12/05</td>
<td>Q1</td>
<td></td>
</tr>
<tr>
<td>COSSMAN</td>
<td>MARK</td>
<td>1996/12/19</td>
<td>Q1</td>
<td></td>
</tr>
<tr>
<td>CROWDEN</td>
<td>CHRIS</td>
<td>1996/12/03</td>
<td>Q1</td>
<td></td>
</tr>
<tr>
<td>CULLEN</td>
<td>DENNIS</td>
<td>1995/09/05</td>
<td>Q4</td>
<td></td>
</tr>
<tr>
<td>CUMMINGS</td>
<td>JAMES</td>
<td>1993/07/11</td>
<td>Q4</td>
<td></td>
</tr>
<tr>
<td>CUTLIP</td>
<td>GREGG</td>
<td>1997/03/26</td>
<td>Q2</td>
<td></td>
</tr>
</tbody>
</table>

FIQTR obtains the financial quarter for START_DATE (format YYMD) and returns a column with format Q:

\[\text{FIQTR(START\_DATE,'D',10,1,'FYE','Q');}\]

For 1997/10/01, the result is Q1.

For 1996/07/30, the result is Q4.

**FIYYQ: Converting a Calendar Date to a Financial Date**

The FIYYQ function returns a financial date containing both the financial year and quarter that corresponds to a given calendar date. The returned financial date is based on the financial year starting date and the financial year numbering convention.

Since Dialogue Manager interprets a date as alphanumeric or numeric, and FIYYQ requires a standard date stored as an offset from the base date, do not use FIYYQ with Dialogue Manager unless you first convert the variable used as the input date to an offset from the base date.

For more information, see [Calling a Function From a Dialogue Manager Command](#).

**Syntax:** How to Convert a Calendar Date to a Financial Date

\[\text{FIYYQ(inputdate, lowcomponent, startmonth, startday, yrnumbering, output)}\]

where:

- **inputdate**
  - Date
Is the date for which the financial year is returned. The date must be a standard date stored as an offset from the base date.

If the financial year does not begin on the first day of a month, the date must have Y(Y), M, and D components, or Y(Y) and JUL components (note that JUL is equivalent to YJUL). Otherwise, the date only needs Y(Y) and M components or Y(Y) and Q components.

**lowcomponent**

Alphanumeric

Is one of the following:

- D if the date contains a D or JUL component.
- M if the date contains an M component, but no D component.
- Q if the date contains a Q component.

**startmonth**

Numeric

1 through 12 are used to represent the starting month of the financial year, where 1 represents January and 12 represents December. If the low component is Q, the start month must be 1, 4, 7, or 10.

**startday**

Numeric

Is the starting day of the starting month, usually 1. If the low component is M or Q, 1 is required.

**yrnumbering**

Alphanumeric

Valid values are:

- FYE to specify the Financial Year Ending convention. The financial year number is the calendar year of the ending date of the financial year. For example, when the financial year starts on October 1, 2008, the date 2008 November 1 is in FY 2009 Q1 because that date is in the financial year that ends on 2009 September 30.

- FYS to specify the Financial Year Starting convention. The financial year number is the calendar year of the starting date of the financial year. For example, when the financial year starts on April 6, 2008, the date 2008 July 6 is in FY 2008 Q2 because that date is in the financial year that starts on 2008 April 6.
Y[Y]Q or QY[Y]

In case of an error, zero is returned.

Note: February 29 cannot be used as a start day for a financial year.

Example: Converting a Calendar Date to a Financial Date

The following request against the CENTHR data source converts each employee starting date (field START_DATE, format YYMD) to a financial date containing year and quarter components in all the supported formats: YQ, YYQ, QY, and QYY.

```
DEFINE FILE CENTHR
FISYQ/YQ=FIYYQ(START_DATE,'D',10,1,'FYE',FISYQ);
FISYYQ/YYQ=FIYYQ(START_DATE,'D',10,1,'FYE',FISYYQ);
FISQY/QY=FIYYQ(START_DATE,'D',10,1,'FYE',FISQY);
FISQYY/QYY=FIYYQ(START_DATE,'D',10,1,'FYE',FISQYY);
END
TABLE FILE CENTHR
PRINT START_DATE FISYQ FISYYQ FISQY FISQYY
BY LNAME BY FNAME
WHERE LNAME LIKE 'C%'
END
```

On the output, note that the date November 12, 1998 (1998/11/12) is converted to Q1 1999 because the starting month is October (10), and the FYE numbering convention is used:

<table>
<thead>
<tr>
<th>Last Name</th>
<th>First Name</th>
<th>Starting Date</th>
<th>FISYQ</th>
<th>FISYYQ</th>
<th>FISQY</th>
<th>FISQYY</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHIEN</td>
<td>CHRISTINE</td>
<td>1997/10/01</td>
<td>98 Q1</td>
<td>1998 Q1</td>
<td>Q1 98</td>
<td>Q1 1998</td>
</tr>
<tr>
<td>CLEVELAND</td>
<td>PHILIP</td>
<td>1996/07/30</td>
<td>96 Q4</td>
<td>1996 Q4</td>
<td>Q4 96</td>
<td>Q4 1996</td>
</tr>
<tr>
<td>CLINE</td>
<td>STEPHEN</td>
<td>1998/11/12</td>
<td>99 Q1</td>
<td>1999 Q1</td>
<td>Q1 99</td>
<td>Q1 1999</td>
</tr>
<tr>
<td>COHEN</td>
<td>DANIEL</td>
<td>1997/10/05</td>
<td>98 Q1</td>
<td>1998 Q1</td>
<td>Q1 98</td>
<td>Q1 1998</td>
</tr>
<tr>
<td>CORRIVEAU</td>
<td>RAYMOND</td>
<td>1997/12/05</td>
<td>98 Q1</td>
<td>1998 Q1</td>
<td>Q1 98</td>
<td>Q1 1998</td>
</tr>
<tr>
<td>COSSMAN</td>
<td>MARK</td>
<td>1996/12/19</td>
<td>97 Q1</td>
<td>1997 Q1</td>
<td>Q1 97</td>
<td>Q1 1997</td>
</tr>
<tr>
<td>CRONIN</td>
<td>CHRIS</td>
<td>1996/12/03</td>
<td>97 Q1</td>
<td>1997 Q1</td>
<td>Q1 97</td>
<td>Q1 1997</td>
</tr>
<tr>
<td>CROWDER</td>
<td>WESLEY</td>
<td>1996/09/17</td>
<td>96 Q4</td>
<td>1996 Q4</td>
<td>Q4 96</td>
<td>Q4 1996</td>
</tr>
<tr>
<td>CULLEN</td>
<td>DENNIS</td>
<td>1995/09/05</td>
<td>95 Q4</td>
<td>1995 Q4</td>
<td>Q4 95</td>
<td>Q4 1995</td>
</tr>
<tr>
<td>CUMMINGS</td>
<td>JAMES</td>
<td>1993/07/11</td>
<td>93 Q4</td>
<td>1993 Q4</td>
<td>Q4 93</td>
<td>Q4 1993</td>
</tr>
<tr>
<td>CUTLIP</td>
<td>GREGG</td>
<td>1997/03/26</td>
<td>97 Q2</td>
<td>1997 Q2</td>
<td>Q2 97</td>
<td>Q2 1997</td>
</tr>
</tbody>
</table>

FIYYQ returns the financial date in format YQ that corresponds to START_DATE (format YYMD);

```
FIYYQ(START_DATE,'D',10,1,'FYE','YQ');
```

For 1997/10/01, the result is 98 Q1.

For 1996/07/30, the result is 96 Q4.
TODAY: Returning the Current Date

Available Languages: reporting

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

You can also retrieve the date in the same format (separated by slashes) 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.

A compiled MODIFY procedure must use TODAY to obtain the date. It cannot use the system variables.

Syntax: How to Retrieve the Current Date

TODAY(output)

where:

output

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 apply:

- If the format is A8 or A9, TODAY returns the 2-digit year.
- If the format is A10 or greater, TODAY returns the 4-digit year.

Example: Retrieving the Current Date

TODAY retrieves the current date and stores it in the DATE field. The request then displays the date in the page heading.

```
DEFINE FILE EMPLOYEE
DATE/A10 WITH EMP_ID = TODAY(DATE);
END
```
The output is:

SALARY REPORT RUN ON 12/13/2006
DEPARTMENT       CURR_SAL
--------------     --------
MIS                $108,002.00
PRODUCTION        $114,282.00

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

TODAY('A10')

Using Legacy Date Functions

The legacy date functions were created for use with dates in integer, packed decimal, or alphanumeric format.

For detailed information on each legacy date function, see:

AYM: Adding or Subtracting Months
AYMD: Adding or Subtracting Days on page 331
CHGDAT: Changing How a Date String Displays on page 333
DA Functions: Converting a Legacy Date to an Integer on page 336
DMY, MDY, YMD: Calculating the Difference Between Two Dates on page 338
DOWK and DOWKL: Finding the Day of the Week on page 339
DT Functions: Converting an Integer to a Date on page 341
GREGDT: Converting From Julian to Gregorian Format on page 342
JULDAT: Converting From Gregorian to Julian Format on page 344
YM: Calculating Elapsed Months on page 346

Using Old Versions of 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.
All legacy date functions support dates for the year 2000 and later.

**Using Dates With Two- and Four-Digit Years**

Legacy date functions accept dates with two- or four-digit years. Four-digit years that display the century, such as 2000 or 1900, can be used if their formats are specified as I8YYMD, P8YYMD, D8YYMD, F8YYMD, or A8YYMD. Two-digit years can use the DEFCENT and YRTHRESH parameters to assign century values if the field has a length of six (for example, I6YMD). For information on these parameters, see *Customizing Your Environment in Developing Reporting Applications.*

**Example: Using Four-Digit Years**

The EDIT function creates dates with four-digit years. The functions JULDAT and GREGDT then convert these dates to Julian and Gregorian formats.

```plaintext
DEFINE FILE EMPLOYEE
DATE/I8YYMD = EDIT('19'|EDIT(HIRE_DATE));
JDATE/I7 = JULDAT(DATE, 'I7');
GDATE/I8 = GREGDT(JDATE, 'I8');
END

TABLE FILE EMPLOYEE
PRINT DATE JDATE GDATE
END
```

The output is:

<table>
<thead>
<tr>
<th>DATE</th>
<th>JDATE</th>
<th>GDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980/06/02</td>
<td>1980154</td>
<td>19800602</td>
</tr>
<tr>
<td>1981/07/01</td>
<td>1981182</td>
<td>19810701</td>
</tr>
<tr>
<td>1982/05/01</td>
<td>1982121</td>
<td>19820501</td>
</tr>
<tr>
<td>1982/01/04</td>
<td>1982004</td>
<td>19820104</td>
</tr>
<tr>
<td>1982/08/01</td>
<td>1982213</td>
<td>19820801</td>
</tr>
<tr>
<td>1982/01/04</td>
<td>1982004</td>
<td>19820104</td>
</tr>
<tr>
<td>1982/07/01</td>
<td>1982182</td>
<td>19820701</td>
</tr>
<tr>
<td>1981/07/01</td>
<td>1981182</td>
<td>19810701</td>
</tr>
<tr>
<td>1982/04/01</td>
<td>1982091</td>
<td>19820401</td>
</tr>
<tr>
<td>1982/02/02</td>
<td>1982033</td>
<td>19820202</td>
</tr>
<tr>
<td>1982/04/01</td>
<td>1982091</td>
<td>19820401</td>
</tr>
<tr>
<td>1981/11/02</td>
<td>1981306</td>
<td>19811102</td>
</tr>
<tr>
<td>1982/04/01</td>
<td>1982091</td>
<td>19820401</td>
</tr>
<tr>
<td>1982/05/15</td>
<td>1982135</td>
<td>19820515</td>
</tr>
</tbody>
</table>
**Example: Using Two-Digit Years**

The AYMD function returns an eight-digit date when the input argument has a six-digit legacy date format. Since DEFCENT is 19 and YRTHRESH is 83, year values from 83 through 99 are interpreted as 1983 through 1999, and year values from 00 through 82 are interpreted as 2000 through 2082.

```plaintext
SET DEFCENT=19, YRTHRESH=83
DEFINE FILE EMPLOYEE
NEW_DATE/I8YYMD = AYMD(EFFECT_DATE, 30, 'I8');
END

TABLE FILE EMPLOYEE
PRINT EFFECT_DATE NEW_DATE BY EMP_ID
END
```

The output is:

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>EFFECT_DATE</th>
<th>NEW_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>071382660</td>
<td>82/11/01</td>
<td>2082/12/01</td>
</tr>
<tr>
<td>112847612</td>
<td>83/01/01</td>
<td>1983/01/31</td>
</tr>
<tr>
<td>117593129</td>
<td>83/03/01</td>
<td>1983/03/31</td>
</tr>
<tr>
<td>119265415</td>
<td>83/05/01</td>
<td>1983/05/31</td>
</tr>
<tr>
<td>119329144</td>
<td>83/07/01</td>
<td>1983/07/31</td>
</tr>
<tr>
<td>123764317</td>
<td>83/09/01</td>
<td>1983/09/31</td>
</tr>
<tr>
<td>126724188</td>
<td>83/11/01</td>
<td>1983/11/31</td>
</tr>
<tr>
<td>219984371</td>
<td>83/13/01</td>
<td>1983/13/31</td>
</tr>
<tr>
<td>326179357</td>
<td>83/15/01</td>
<td>1983/15/31</td>
</tr>
<tr>
<td>451123478</td>
<td>83/17/01</td>
<td>1983/17/31</td>
</tr>
<tr>
<td>543729165</td>
<td>83/19/01</td>
<td>1983/19/31</td>
</tr>
<tr>
<td>818692173</td>
<td>83/21/01</td>
<td>1983/21/31</td>
</tr>
</tbody>
</table>

**AYMD: Adding or Subtracting Days**

Available Languages: reporting, Maintain

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

```plaintext
AYMD(indate, days, output)
```

where:

- `indate`
  
  I6, I6YMD, I8, I8YYMD

  Is the legacy date in year-month-day format. If the date is not valid, the function returns the value 0 (zero).
**AYMD: Adding or Subtracting Days**

*days*

Integer

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

*output*

I6, I6YMD, I8, or I8YMD

Is the same format as `indate`.

Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. If `indate` is a field, `output` must have the same format.

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 AFTER35DAYS:

```plaintext
TABLE FILE EMPLOYEE
PRINT HIRE_DATE AND COMPUTE
AFTER35DAYS/I6YMD = AYMD(HIRE_DATE, 35, AFTER35DAYS);
BY LAST_NAME BY FIRST_NAME
WHERE DEPARTMENT EQ 'PRODUCTION';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>HIRE_DATE</th>
<th>AFTER35DAYS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>82/08/01</td>
<td>82/09/05</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>82/01/04</td>
<td>82/02/08</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>82/02/02</td>
<td>82/03/09</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>82/07/01</td>
<td>82/08/05</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>82/01/04</td>
<td>82/02/08</td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>80/06/02</td>
<td>80/07/07</td>
</tr>
</tbody>
</table>

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

```plaintext
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 How a Date String Displays

Available Languages: reporting, Maintain

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 of the date components, such as year, month, or day.

The input and output date strings are described by display options that specify both the order of date 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:

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Day of the month (01 through 31).</td>
</tr>
<tr>
<td>M</td>
<td>Month of the year (01 through 12).</td>
</tr>
<tr>
<td>Y[Y]</td>
<td>Year. Y indicates a two-digit year (such as 94); YY indicates a four-digit year (such as 1994).</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Displays the month as a three-letter abbreviation.</td>
</tr>
<tr>
<td>X</td>
<td>Displays the full name of the month.</td>
</tr>
</tbody>
</table>

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, then two digit month, then 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:

<table>
<thead>
<tr>
<th>Portion of Date Missing</th>
<th>Portion Supplied by Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Day (for example, from YM to YMD)</td>
<td>Last day of the month.</td>
</tr>
<tr>
<td>Month (for example, from Y to YM)</td>
<td>Last month of the year (December).</td>
</tr>
<tr>
<td>Year (for example, from MD to YMD)</td>
<td>The year 99.</td>
</tr>
<tr>
<td>Converting year from two-digit to four-digit</td>
<td>The century will be determined by the 100-year window defined by DEFCENT and YRTHRESH.</td>
</tr>
<tr>
<td>(for example, from YMD to YYMD)</td>
<td>See Customizing Your Environment in Developing Reporting Applications or Working With Cross-Century Dates in the iBase archive for details on DEFCENT and YRTHRESH.</td>
</tr>
<tr>
<td></td>
<td>See Working With Cross-Century Dates in Developing Applications for details on DEFCENT and YRTHRESH.</td>
</tr>
</tbody>
</table>

**Syntax:** How to Change the Date Display String

```plaintext
CHGDAT('in_display_options','out_display_options',date_string,output)
```

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

Axx, 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

The EDIT function changes HIRE_DATE from numeric to alphanumeric format. CHGDAT then converts each value in ALPHA_HIRE from displaying the components as YMD to MDYYX and stores the result in HIRE_MDY, which has the format A17. The option X in the output value displays the full name of the month.

TABLE FILE EMPLOYEE
PRINT HIRE_DATE AND COMPUTE
ALPHA_HIRE/A17 = EDIT(HIRE_DATE); NOPRINT AND COMPUTE
HIRE_MDY/A17 = CHGDAT('YMD', 'MDYYX', ALPHA_HIRE, 'A17');
BY LAST_NAME BY FIRST_NAME
WHERE DEPARTMENT EQ 'PRODUCTION';
END
The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>HIRE_DATE</th>
<th>HIRE_MDY</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>82/08/01</td>
<td>AUGUST 01 1982</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>82/01/04</td>
<td>JANUARY 04 1982</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>82/02/02</td>
<td>FEBRUARY 02 1982</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>82/07/01</td>
<td>JULY 01 1982</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>82/01/04</td>
<td>JANUARY 04 1982</td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>80/06/02</td>
<td>JUNE 02 1980</td>
</tr>
</tbody>
</table>

ALPHA_HIRE is HIRE_DATE converted from numeric to alphanumerical 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

Available Languages: reporting, Maintain

The DA functions convert a legacy date to the number of days between it and a base date (December 31, 1899). By converting a date to the number of days, you can add and 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 back to a date using the DT functions discussed in DT Functions: Converting an Integer to a Date on page 341.

There are six DA functions; each one accepts a date in a different format.
**Syntax:** How to Convert a Date to an Integer

function(indate, output)

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 or P6xxx, where xxx corresponds to the function DAxxx you are using.

Is the legacy date to be converted, or the name of a field that contains the date. The date is truncated to an integer before conversion. 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

Integer

Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. The format of the date returned depends on the function.

**Example:** Converting Dates and Calculating the Difference Between Them

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

```
TABLE FILE EMPLOYEE
PRINT DAT_INC AS 'RAISE DATE' AND COMPUTE
DAYS_HIRED/I8 = DAYMD(DAT_INC, 'I8') - DAYMD(HIRE_DATE, 'I8');
BY LAST_NAME BY FIRST_NAME
IF DAYS_HIRED NE 0
WHERE DEPARTMENT EQ 'PRODUCTION';
```
The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>RAISE_DATE</th>
<th>DAYS_HIRED</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>82/05/14</td>
<td>130</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>82/05/14</td>
<td>101</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>82/05/14</td>
<td>130</td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>82/01/01</td>
<td>578</td>
</tr>
<tr>
<td></td>
<td></td>
<td>81/01/01</td>
<td>213</td>
</tr>
</tbody>
</table>

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:

\[
\text{DAYMD(DAT\_INC, 'I8') - DAYMD(HIRE\_DATE, 'I8')}
\]

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

Available Languages: reporting, Maintain

The DMY, MDY, and YMD functions calculate the difference between two legacy dates in integer, alphanumeric, or packed format.

**Syntax:** How to Calculate the Difference Between Two Dates

\[
\text{function(from_date, to_date)}
\]

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**
  - I, P, or A format with date display options.
  - Is the beginning legacy date, or the name of a field that contains the date.

- **to_date**
  - I, P, or A format with date display options.I6xxx or I8xxx where xxx corresponds to the specified function (DMY, YMD, or MDY).
  - Is the end date, or the name of a field that contains the 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:

```sql
TABLE FILE EMPLOYEE
SUM HIRE_DATE FST.DAT_INC AS 'FIRST PAY, INCREASE' AND COMPUTE
DIFF/14 = YMD(HIRE_DATE, FST.DAT_INC); AS 'DAYS, BETWEEN'
BY LAST_NAME BY FIRST_NAME
WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>HIRE_DATE</th>
<th>FIRST PAY</th>
<th>INCREASE</th>
<th>DAYS BETWEEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>82/04/01</td>
<td>82/04/01</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>81/11/02</td>
<td>82/04/09</td>
<td>158</td>
<td></td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>82/04/01</td>
<td>82/06/11</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>82/05/01</td>
<td>82/06/01</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>81/07/01</td>
<td>82/01/01</td>
<td>184</td>
<td></td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>81/07/01</td>
<td>82/01/01</td>
<td>184</td>
<td></td>
</tr>
</tbody>
</table>

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**

Available Languages: reporting, Maintain

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\}(\textit{indate}, \textit{output})

where:

\textit{indate}

I6YMD or I8YYMD

Is the legacy date in year-month-day format. If the date is not valid, the function returns spaces. If the date specifies a two digit year and DEFCENT and YRTHRESH values have not been set, the function assumes the 20th century.

\textit{output}

DOWK: A4. DOWKL: A12

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

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 DATED:

\begin{verbatim}
TABLE FILE EMPLOYEE
PRINT EMP_ID AND HIRE_DATE AND COMPUTE
DATED/A4 = DOWK(HIRE_DATE, DATED);
WHERE DEPARTMENT EQ 'PRODUCTION';
END
\end{verbatim}

The output is:

\begin{verbatim}
EMP_ID    HIRE_DATE    DATED
-------    ----------    -----
071382660  80/06/02    MON
119265415  82/01/04    MON
119329144  82/08/01    SUN
123764317  82/01/04    MON
126724188  82/07/01    THU
451123478  82/02/02    TUE
\end{verbatim}

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.

\texttt{DOWK(HIRE_DATE, 'A4')}\n
For 80/06/02, the result is MON.

For 82/08/01, the result is SUN.
DT Functions: Converting an Integer to a Date

Available Languages: reporting, Maintain

The DT functions convert an integer representing the number of days elapsed since December 31, 1899 to the corresponding date. They are useful when you are performing arithmetic on a date converted to the number of days (for more information, see DA Functions: Converting a Legacy Date to an Integer on page 336). The DT functions convert the result back to a date.

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

**Note:** When USERFNS is set to LOCAL, DT functions only display a six-digit date.

**Syntax:**

How to Convert an Integer to a Date

\[ function(number, output) \]

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 December 31, 1899. The number is truncated to an integer.
  - Is the number of days since the base date, possibly received from the functions DAxxx.

- **output**
  - I8xxx, where xxx corresponds to the function DTxxx in the above list.
  - Is the name of the field containing the result or the format of the output value enclosed in single quotation marks. The output format depends on the function being used.
Example: Converting an Integer to a Date

DTMDY converts the NEWF field (which was converted to the number of days by DAYMD) to the corresponding date and stores the result in NEW_HIRE_DATE:

```plaintext
-* THIS PROCEDURE CONVERTS HIRE_DATE, WHICH IS IN I6YMD FORMAT,
-* TO A DATE IN I8MDYY FORMAT.
-* FIRST IT USES THE DAYMD FUNCTION TO CONVERT HIRE_DATE
-* TO A NUMBER OF DAYS.
-* THEN IT USES THE DTMDY FUNCTION TO CONVERT THIS NUMBER OF
-* DAYS TO I8MDYY FORMAT
- *
DEFINE FILE EMPLOYEE
NEWF/I8 WITH EMP_ID = DAYMD(HIRE_DATE, NEWF);
NEW_HIRE_DATE/I8MDYY WITH EMP_ID = DTMDY(NEWF, NEW_HIRE_DATE);
END FILE

TABLE FILE EMPLOYEE
PRINT HIRE_DATE NEW_HIRE_DATE
BY FN BY LN
WHERE DEPARTMENT EQ 'MIS'
END FILE
```

The output is:

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>HIRE_DATE</th>
<th>NEW_HIRE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BARBARA</td>
<td>CROSS</td>
<td>81/11/02</td>
<td>11/02/1981</td>
</tr>
<tr>
<td>DIANE</td>
<td>JONES</td>
<td>82/05/01</td>
<td>05/01/1982</td>
</tr>
<tr>
<td>JOHN</td>
<td>MCCOY</td>
<td>81/07/01</td>
<td>07/01/1981</td>
</tr>
<tr>
<td>MARY</td>
<td>GREENSPAN</td>
<td>82/04/01</td>
<td>04/01/1982</td>
</tr>
<tr>
<td>ROSEMARIE</td>
<td>BLACKWOOD</td>
<td>82/04/01</td>
<td>04/01/1982</td>
</tr>
</tbody>
</table>

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.

```plaintext
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

Available Languages: reporting, Maintain

The GREGDT function converts a date in Julian format (year-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; June 21, 2004 in Julian format is 2004173.

**Reference:** Format Options for GREGDT

GREGDT converts a Julian date to either YMD or YYMD format using the DEFCENT and YRTHRESH parameter settings to determine the century, if required. GREGDT returns a date as follows:

- If the format is I6 or I7, GREGDT returns the date in YMD format.
- If the format is I8 or greater, GREGDT returns the date in YYMD format.

**Syntax:** How to Convert From Julian to Gregorian Format

```
GREGDT(indate, output)
```

where:

**indate**

I5 or I7

Is the Julian date, which is truncated to an integer before conversion. Each value must be a five- or seven-digit number after truncation. If the date is invalid, the function returns a 0 (zero).

**output**

I6, I8, I6YMD, or I8YYMD

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

**Example:** Converting From Julian to Gregorian Format

GREGDT converts the JULIAN field to YYMD (Gregorian) format. It determines the century using the default DEFCENT and YRTHRESH parameter settings.

```
TABLE FILE EMPLOYEE
PRINT HIRE_DATE
AND
COMPUTE JULIAN/I5 = JULDAT(HIRE_DATE, JULIAN); AND
COMPUTE GREG_DATE/I8 = GREGDT(JULIAN, 'I8');
BY LAST_NAME BY FIRST_NAME
WHERE DEPARTMENT EQ 'PRODUCTION';
END
```
The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>HIRE_DATE</th>
<th>JULIAN</th>
<th>GREG_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>82/08/01</td>
<td>82213</td>
<td>19820801</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>82/01/04</td>
<td>82004</td>
<td>19820104</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>82/02/02</td>
<td>82033</td>
<td>19820202</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>82/07/01</td>
<td>82182</td>
<td>19820701</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>82/01/04</td>
<td>82004</td>
<td>19820104</td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>80/06/02</td>
<td>80154</td>
<td>19800602</td>
</tr>
</tbody>
</table>

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.

\[
\text{DTMDY(NEWF, 'I8MDYY')}
\]

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

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

**JULDAT: Converting From Gregorian to Julian Format**

Available Languages: reporting, Maintain

The JULDAT function converts a date from Gregorian format (year-month-day) to Julian format (year-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.

**Reference:** Format Settings for JULDAT

JULDAT converts a Gregorian date to either YYNNN or YYYYNNN format, using the DEFCENT and YRTHRESH parameter settings to determine if the century is required.

JULDAT returns dates as follows:

- If the format is I6, JULDAT returns the date in YYNNN format.
- If the format is I7 or greater, JULDAT returns the date in YYYYNNN format.
Syntax: How to Convert From Gregorian to Julian Format

\[ \text{JULDAT}(\text{indate, output}) \]

where:

\text{indate}

I6, I8, I6YMD, I8YYMD

Is the legacy date to convert or the name of the field that contains the date in year-month-day format (YMD or YYMD).

\text{output}

I5 or I7

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

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.

\begin{verbatim}
TABLE FILE EMPLOYEE
PRINT HIRE_DATE AND COMPUTE
JULIAN/I7 = JULDAT(HIRE_DATE, JULIAN);
BY LAST_NAME BY FIRST_NAME
WHERE DEPARTMENT EQ 'PRODUCTION';
END
\end{verbatim}

The output is:

\begin{tabular}{llll}
\hline
LAST_NAME & FIRST_NAME & HIRE_DATE & JULIAN \\
\hline
BANNING   & JOHN       & 82/08/01   & 1982213  \\
IRVING    & JOAN       & 82/01/04   & 1982004  \\
MCKNIGHT  & ROGER      & 82/02/02   & 1982033  \\
ROMANS    & ANTHONY    & 82/07/01   & 1982182  \\
SMITH     & RICHARD    & 82/01/04   & 1982004  \\
STEVENS   & ALFRED     & 80/06/02   & 1980154  \\
\hline
\end{tabular}

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.

\[ \text{GREGDT}(\text{JULIAN}, 'I8') \]

For 82213, the result is 19820801.

For 82004, the result is 19820104.
YM: Calculating Elapsed Months

Available Languages: reporting, Maintain

The YM function calculates the number of months 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(fromdate, todate, output)
```

where:

- **fromdate**
  I4YM or I6YYM
  Is the start date in year-month format (for example, I4YM). If the date is not valid, the function returns the value 0 (zero).

- **todate**
  I4YM or I6YYM
  Is the end date in year-month format. If the date is not valid, the function returns the value 0 (zero).

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

*Tip:* If `fromdate` 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, which is now after the decimal point.
Example: Calculating Elapsed Months

The COMPUTE commands convert the dates from year-month-day to year-month format; then YM calculates the difference between the values in the HIRE_DATE/100 and DAT_INC/100 fields:

TABLE FILE EMPLOYEE
PRINT DAT_INC AS 'RAISE DATE' AND COMPUTE
HIRE_MONTH/I4YM = HIRE_DATE/100; NOPRINT AND COMPUTE
MONTH_INC/I4YM = DAT_INC/100; NOPRINT AND COMPUTE
MONTHS_HIRED/I3 = YM(HIRE_MONTH, MONTH_INC, 'I3');
BY LAST_NAME BY FIRST_NAME BY HIRE_DATE
IF MONTHS_HIRED NE 0
WHERE DEPARTMENT EQ 'MIS';
END

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>HIRE_DATE</th>
<th>RAISE_DATE</th>
<th>MONTHS_HIRED</th>
</tr>
</thead>
<tbody>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>81/11/02</td>
<td>82/04/09</td>
<td>5</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>82/04/01</td>
<td>82/06/11</td>
<td>2</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>82/05/01</td>
<td>82/06/01</td>
<td>1</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>81/07/01</td>
<td>82/01/01</td>
<td>6</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>81/07/01</td>
<td>82/01/01</td>
<td>6</td>
</tr>
</tbody>
</table>

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')
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 nanosecond.

**In this chapter:**

- Using Date-Time Functions
- HADD: Incrementing a Date-Time Value
- HCNVRT: Converting a Date-Time Value to Alphanumeric Format
- HDATE: Converting the Date Portion of a Date-Time Value to a Date Format
- HDIFF: Finding the Number of Units Between Two Date-Time Values
- HDTTM: Converting a Date Value to a Date-Time Value
- HGETC: Storing the Current Local Date and Time in a Date-Time Field
- HGETZ: Storing the Current Coordinated Universal Time in a Date-Time Field
- HHMMSS: Retrieving the Current Time
- HHMS: Converting a Date-Time Value to a Time Value
- HINPUT: Converting an Alphanumeric String to a Date-Time Value
- HMIDNT: Setting the Time Portion of a Date-Time Value to Midnight
- HNAME: Retrieving a Date-Time Component in Alphanumeric Format
- HPART: Retrieving a Date-Time Component as a Numeric Value
- HSETPT: Inserting a Component Into a Date-Time Value
- HTIME: Converting the Time Portion of a Date-Time Value to a Number
- HTMTOTS or TIMETOTS: Converting a Time to a Timestamp
- HYYWD: Returning the Year and Week Number From a Date-Time Value
Using 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
```

Date-Time Parameters

The DATEFORMAT parameter specifies the order of the date components for certain types of date-time values. The WEEKFIRST parameter specifies the first day of the week. The DTSTRICT parameter determines the extent to which date-time values are checked for validity.

Specifying the Order of Date Components

The DATEFORMAT parameter specifies the order of the date components (month/day/year) when date-time values are entered in the formatted string and translated string formats described in Using Date-Time Formats on page 355. It makes the input format of a value independent of the format of the variable to which it is being assigned.
Syntax: How to Specify the Order of Date Components in a Date-Time Field

SET DATEFORMAT = option

where:

option

Can be one of the following: MDY, DMY, YMD, or MYD. MDY is the default value for the U.S. English format.

Example: Using the DATEFORMAT Parameter

The following request uses a natural date literal with ambiguous numeric day and month components (APR 04 05) as input to the HINPUT function:

SET DATEFORMAT = MYD
DEFINE FILE EMPLOYEE
DTFLDYYMD/HYMDI = HINPUT(9,'APR 04 05', 8, DTFLDYYMD);
END
TABLE FILE EMPLOYEE
SUM CURR_SAL NOPRINT DTFLDYYMD
END

With DATEFORMAT set to MYD, the value is interpreted as April 5, 1904:

DTFLDYYMD
--------
1904-04-05 00:00

Specifying the First Day of the Week for Use in Date-Time Functions

The WEEKFIRST parameter specifies a day of the week as the start of the week. This is used in week computations by the HADD, HDIFF, HNAME, HPART, and HYYWD functions. It is also used by the DTADD, DTDIFF, DTRUNC, and DTPART functions. The default values are different for these functions, as described in How to Set a Day as the Start of the Week on page 352. The WEEKFIRST parameter does not change the day of the month that corresponds to each day of the week, but only specifies which day is considered the start of the week.

The HPART, DTPART, HYYWD, and HNAME subroutines can extract a week number from a date-time value. To determine a week number, they can use different definitions. For example, ISO 8601 standard week numbering defines the first week of the year as the first week in January with four or more days. Any preceding days in January belong to week 52 or 53 of the preceding year. The ISO standard also establishes Monday as the first day of the week.

You specify which type of week numbering to use by setting the WEEKFIRST parameter, as described in How to Set a Day as the Start of the Week on page 352.
Since the week number returned by HNAME, DTPART, and HPART functions can be in the current year or the year preceding or following, the week number by itself may not be useful. The function HYYWD returns both the year and the week for a given date-time value.

**Syntax:** How to Set a Day as the Start of the Week

```
SET WEEKFIRST = value
```

where:

- **value**
  
  Can be:

- **1 through 7**, representing Sunday through Saturday with non-standard week numbering.

  Week numbering using these values establishes the first week in January with seven days as week number 1. Preceding days in January belong to the last week of the previous year. All weeks have seven days.

- **ISO1 through ISO7**, representing Sunday through Saturday with ISO standard week numbering.

  **Note:** ISO is a synonym for ISO2.

  Week numbering using these values establishes the first week in January with at least four days as week number 1. Preceding days in January belong to the last week of the previous year. All weeks have seven days.

- **STD1 through STD7**, in which the digit 1 (Sunday) through 7 (Saturday) indicates the starting day of the week.

  **Note:** STD without a digit is equivalent to STD1.

  Week numbering using these values is as follows. Week number 1 begins on January 1 and ends on the day preceding the first day of the week. For example, for STD1, the first week ends on the first Saturday of the year. The first and last week may have fewer than seven days.

- **SIMPLE**, which establishes January 1 as the start of week 1, January 8 is the start of week 2, and so on. The first day of the week is, thus, the same as the first day of the year. The last week (week 53) is either one or two days long.
0 (zero), is the value of the WEEKFIRST setting before the user issues an explicit WEEKFIRST setting. The date-time functions HPART, HNAME, HYYWD, HADD, and HDIFF use Saturday as the start of the week, when the WEEKFIRST setting is 0. The simplified functions DTADD, DTDIFF, DTRUNC, and DTPART, as well as printing of dates truncated to weeks, and recognition of date constant strings that contain week numbers, use Sunday as the default value, when the WEEKFIRST setting is 0. If the user explicitly sets WEEKFIRST to another value, that value is used by all of the functions.

Example: Setting Sunday as the Start of the Week

The following designates Sunday as the start of the week, using non-standard week numbering:

```
SET WEEKFIRST = 1
```

Syntax: How to View the Current Setting of WEEKFIRST

```
? SET WEEKFIRST
```

This returns the value that indicates the week numbering algorithm and the first day of the week. For example, the integer 1 represents Sunday with non-standard week numbering.

Controlling Processing of Date-Time Values

Strict processing checks date-time values when they are input by an end user, read from a transaction file, displayed, or returned by a subroutine to ensure that they represent a valid date and time. For example, a numeric month must be between 1 and 12, and the day must be within the number of days for the specified month.
Syntax: How to Enable Strict Processing of Date-Time Values

SET DTSTRICT = {ON|OFF}

where:

ON
Invokes strict processing. ON is the default value.

Strict processing checks date-time values when they are input by an end user, read from a transaction file, displayed, or returned by a subroutine to ensure that they represent a valid date and time. For example, a numeric month must be between 1 and 12, and the day must be within the number of days for the specified month.

If DTSTRICT is ON and the result would be an invalid date-time value, the function returns the value zero (0).

OFF
Does not invoke strict processing. Date-time components can have any value within the constraint of the number of decimal digits allowed in the field. For example, if the field is a two-digit month, the value can be 12 or 99, but not 115.

Supplying Arguments for Date-Time Functions

Date-time functions may operate on a component of a date-time value. This topic lists the valid component names and abbreviations for use with these functions.

Reference: Arguments for Use With Date and Time Functions

The following component names, valid abbreviations, and values are supported as arguments for the date-time functions that require them:

<table>
<thead>
<tr>
<th>Component Name</th>
<th>Abbreviation</th>
<th>Valid Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>yy</td>
<td>0001-9999</td>
</tr>
<tr>
<td>quarter</td>
<td>qq</td>
<td>1-4</td>
</tr>
<tr>
<td>month</td>
<td>mm</td>
<td>1-12 or a month name, depending on the function.</td>
</tr>
<tr>
<td>day-of-year</td>
<td>dy</td>
<td>1-366</td>
</tr>
<tr>
<td>Component Name</td>
<td>Abbreviation</td>
<td>Valid Values</td>
</tr>
<tr>
<td>----------------------------</td>
<td>--------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>day or day-of-month</td>
<td>dd</td>
<td>1-31 (The two component names are equivalent.)</td>
</tr>
<tr>
<td>week</td>
<td>wk</td>
<td>1-53</td>
</tr>
<tr>
<td>weekday</td>
<td>dw</td>
<td>1-7 (Sunday-Saturday)</td>
</tr>
<tr>
<td>hour</td>
<td>hh</td>
<td>0-23</td>
</tr>
<tr>
<td>minute</td>
<td>mi</td>
<td>0-59</td>
</tr>
<tr>
<td>second</td>
<td>ss</td>
<td>0-59</td>
</tr>
<tr>
<td>millisecond</td>
<td>ms</td>
<td>0-999</td>
</tr>
<tr>
<td>microsecond</td>
<td>mc</td>
<td>0-999999</td>
</tr>
<tr>
<td>nanosecond</td>
<td>ns</td>
<td>0-999999999999</td>
</tr>
</tbody>
</table>

**Note:**
- For an argument that specifies a length of eight, ten, or 12 characters, use eight to include milliseconds, ten to include microseconds, and 12 to include nanoseconds 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 date format (for example, YYMD).

**Using Date-Time Formats**

There are three types of date formats that are valid in date-time values: numeric string format, formatted-string format, and translated-string format. In each format, two-digit years are interpreted using the DEFCENT and YRTHRESH parameters.

Time components are separated by colons and may be followed by A.M., P.M., a.m., or p.m.
The DATEFORMAT parameter specifies the order of the date components (month/day/year) when date-time values are entered in the formatted string and translated string formats. It makes a value’s input format independent of the format of the variable to which it is being assigned.

**Numeric String Format**

The numeric string format is exactly two, four, six, or eight digits. Four-digit strings are considered to be a year (century must be specified), and the month and day are set to January 1. Six and eight-digit strings contain two or four digits for the year, followed by two for the month, and two for the day. Because the component order is fixed with this format, the DATEFORMAT setting is ignored.

If a numeric-string format longer than eight digits is encountered, it is treated as a combined date-time string in the Hnn format.

**Example: Using Numeric String Format**

The following are examples of numeric string date constants:

<table>
<thead>
<tr>
<th>String</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>January 1, 1999</td>
</tr>
<tr>
<td>1999</td>
<td>January 1, 1999</td>
</tr>
<tr>
<td>19990201</td>
<td>February 1, 1999</td>
</tr>
</tbody>
</table>

**Formatted-string Format**

The formatted-string format contains a one or two-digit day, a one or two-digit month, and a two or four-digit year, each component separated by a space, slash, hyphen, or period. All three components must be present and follow the DATEFORMAT setting. If any of the three fields is four digits, it is interpreted as the year, and the other two fields must follow the order given by the DATEFORMAT setting.
**Example:** **Using Formatted-string Format**

The following are examples of formatted-string date constants and specify May 20, 1999:

- `1999/05/20`
- `5 20 1999`
- `99.05.20`
- `1999–05–20`

**Translated-string Format**

The translated-string format contains the full or abbreviated month name. The year must also be present in four-digit or two-digit form. If the day is missing, day 1 of the month is assumed; if present, it can have one or two digits. If the string contains both a two-digit year and a two-digit day, they must be in the order given by the DATEFORMAT setting.

**Example:** **Using Translated-string Format**

The following date is in translated-string format:

- `January 6 2000`

**Time Format**

Time components are separated by colons and may be followed by A.M., P.M., a.m., or p.m.

Seconds can be expressed with a decimal point or be followed by a colon. If there is a colon after seconds, the value following it represents milliseconds. There is no way to express microseconds or nanoseconds using this notation.

A decimal point in the seconds value indicates the decimal fraction of a second. Microseconds can be represented using six decimal digits. Nanoseconds can be represented using nine decimal digits.

**Example:** **Using Time Formats**

The following are examples of acceptable time formats:

- `14:30:20:99` (99 milliseconds)
- `14:30`
- `14:30:20.99` (99/100 seconds)
- `14:30:20.999999` (999999 microseconds)
- `02:30:20:500pm`
Example: Using Universal Date-Time Input Values

With DTSTANDARD settings of STANDARD and STANDARDU, the following date-time values can be read as input:

<table>
<thead>
<tr>
<th>Input Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14:30[:20,99]</td>
<td>Comma separates time components instead of period</td>
</tr>
<tr>
<td>14:30[:20.99]Z</td>
<td>Universal time</td>
</tr>
<tr>
<td>15:30[:20,99]+0100</td>
<td>Each of these is the same as above in Central European Time</td>
</tr>
<tr>
<td>09:30[:20.99]-05</td>
<td>Same as above in Eastern Standard Time</td>
</tr>
</tbody>
</table>

Note that these values are stored identically internally with the STANDARDU setting. With the STANDARD setting, everything following the Z, +, or - is ignored.

Assigning Date-Time Values

A date-time value is a constant in character format assigned by one of the following:

- A sequential data source.
- An expression that defines WHERE or IF criteria or creates a temporary field using the DEFINE or COMPUTE command.

A date-time constant can have blanks at the beginning or end or immediately preceding an am/pm indicator.

Syntax: How to Assign Date-Time Values

In a character file

```
date_string [time_string]
```

or

```
time_string [date_string]
```
In a COMPUTE, DEFINE, or WHERE expression

\[ DT(date\_string \ [time\_string]) \]

or

\[ DT(time\_string \ [date\_string]) \]

In an IF expression

\['date\_string \ [time\_string]'\]

or

\['time\_string \ [date\_string]'\]

where:

- \[time\_string\]
  Is a time string in acceptable format. A time string can have a blank immediately preceding an am/pm indicator.

- \[date\_string\]
  Is a date string in numeric string, formatted-string, or translated-string format.

In an IF criteria, if the value does not contain blanks or special characters, the single quotation marks are not necessary.

**Note:** The date and time strings must be separated by at least one blank space. Blank spaces are also permitted at the beginning and end of the date-time string.

**Example:** Assigning Date-Time Literals

The DT prefix can be used in a COMPUTE, DEFINE, or WHERE expression to assign a date-time literal to a date-time field. For example:

\[
\begin{align*}
\text{DT2/HYYMDS} & = \text{DT}(20051226 \ 05:45); \\
\text{DT3/HYYMDS} & = \text{DT}(2005 \ \text{DEC} \ 26 \ 05:45); \\
\text{DT4/HYYMDS} & = \text{DT}(December \ 26 \ 2005 \ 05:45);
\end{align*}
\]
**Example: Assigning a Date-Time Value in a COMPUTE Command**

The following uses the DT function in a COMPUTE command to create a new field containing an assigned date-time value.

```plaintext
TABLE FILE EMPLOYEE
PRINT LAST_NAME FIRST_NAME AND COMPUTE
NEWSL/D12.2M = CURR_SAL + (0.1 * CURR_SAL);
RAISETIME/HYYMDIA = DT(20000101 09:00AM);
WHERE CURR_JOBCODE LIKE 'B%'
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>NEWSAL</th>
<th>RAISETIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>$14,520.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>$20,328.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>$23,232.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>$20,328.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>$23,958.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>$17,710.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
</tbody>
</table>

**Example: Assigning a Date-Time Value in WHERE Criteria**

The following uses the DT function to create a new field containing an assigned date-time value. This value is then used as a WHERE criteria.

```plaintext
DEFINE FILE EMPLOYEE
NEWSAL/D12.2M = CURR_SAL + (0.1 * CURR_SAL);
RAISETIME/HYYMDIA = DT(20000101 09:00AM);
END
```

```plaintext
TABLE FILE EMPLOYEE
PRINT LAST_NAME FIRST_NAME NEWSAL RAISETIME
WHERE RAISETIME EQ DT(20000101 09:00AM)
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>NEWSAL</th>
<th>RAISETIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>$12,100.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>$14,520.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>$20,328.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>$10,450.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>$32,670.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>$29,548.20</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>$23,232.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>$20,328.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>$23,958.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>$17,710.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>$9,900.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>$29,768.20</td>
<td>2000/01/01 9:00AM</td>
</tr>
</tbody>
</table>
**Example:**Assigning a Date-Time Value in IF Criteria

The following uses the DT function to create a new field containing an assigned date-time value. This value is then used in the IF phrase.

```
DEFINE FILE EMPLOYEE
NEWSAL/D12.2M = CURR_SAL + (0.1 * CURR_SAL);
RAISETIME/HYYMDIA = DT(20000101 09:00AM);
END

TABLE FILE EMPLOYEE
PRINT LAST_NAME FIRST_NAME NEWSAL RAISETIME
IF RAISETIME EQ '20000101 09:00AM'
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>NEWSAL</th>
<th>RAISETIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>$12,100.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>$14,520.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>$20,328.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>$10,450.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>$32,670.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>$29,548.20</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>$23,232.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>$20,328.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>$23,958.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>$17,710.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>$9,900.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>$29,768.20</td>
<td>2000/01/01 9:00AM</td>
</tr>
</tbody>
</table>

**HADD: Incrementing a Date-Time Value**

Available Languages: reporting, Maintain

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

**Syntax:**How to Increment a Date-Time Value

```
HADD(datetime, 'component', increment, length, output)
```

where:

```
datetime
```

Date-time

Is the date-time value to be incremented, the name of a date-time field that contains the value, or an expression that returns the value.
**HADD: Incrementing a Date-Time Value**

*component*

Alphanumeric

Is the name of the component to be incremented enclosed in single quotation marks. For a list of valid components, see *Arguments for Use With Date and Time Functions* on page 354.

**Note:** WEEKDAY is not a valid component for HADD.

*increment*

Integer

Is the number of units (positive or negative) by which to increment the component, the name of a numeric field that contains the value, or an expression that returns the value.

*length*

Integer

Is the number of characters returned. Valid values are:

- **8** indicates a date-time value that includes one to three decimal digits (milliseconds).
- **10** indicates a date-time value that includes four to six decimal digits (microseconds).
- **12** indicates a date-time value that includes seven to nine decimal digits (nanoseconds).

*output*

Date-time

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

**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.
Example: Incrementing the Month Component of a Date-Time Field (Reporting)

HADD adds two months to each value in TRANSDATE and stores the result in ADD_MONTH. If necessary, the day is adjusted so that it is valid for the resulting month.

```
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
ADD_MONTH/HYYMDS = HADD(TRANSDATE, 'MONTH', 2, 8, 'HYYMDS');
WHERE DATE EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>ADD_MONTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>2000/04/05 03:30:00</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>2000/08/26 05:45:00</td>
</tr>
</tbody>
</table>

Example: Incrementing the Month Component of a Date-Time Field (Maintain)

HADD adds two months to the DT1 field:

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID DT1 INTO DTSTK
COMPUTE
NEW_DATE/HYYMDS = HADD(DTSTK.DT1, 'MONTH', 2,10, NEW_DATE);
TYPE "DT1 IS: <DTSTK(1).DT1"
TYPE "NEW_DATE IS: <NEW_DATE"
```

The result is:

- DT1 IS: 2000/1/1 02:57:25
- NEW_DATE IS: 2000/3/1 02:57:25

Transactions: Commits = 1 Rollbacks = 0
Segments: Included = 0 Updated = 0 Deleted = 0

Example: Converting Unix (Epoch) Time to a Date-Time Value

Unix time (also known as Epoch time) defines an instant in time as the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970, not counting leap seconds.

The following DEFINE FUNCTION takes a number representing epoch time and converts it to a date-time value by using the HADD function to add the number of seconds represented by the input value in epoch time to the epoch base date:

```
DEFINE FUNCTION UNIX2GMT(INPUT/I9)
UNIX2GMT/HYYMDS = HADD(DT(1970 JAN 1), 'SECONDS',INPUT,8,'HYYMDS');
END
```
The following request uses this DEFINE FUNCTION to convert the epoch time 1449068652 to a date-time value:

```
DEFINE FILE GGSALES
INPUT/I9=1449068652;
OUTDATE/HMTDYYSB = UNIX2GMT(INPUT);
END
TABLE FILE GGSALES
PRINT DATE NOPRINT INPUT OUTDATE
WHERE RECORDLIMIT EQ 1
ON TABLE SET PAGE NOLEAD
END
```

The output is shown in the following image:

```
<table>
<thead>
<tr>
<th>INPUT</th>
<th>OUTDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1449068652</td>
<td>December 02 2015 3:04:12 pm</td>
</tr>
</tbody>
</table>
```

**HCNVRT: Converting a Date-Time Value to Alphanumeric Format**

Available Languages: reporting, Maintain

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(datetime, '(format)', length, output)
```

where:

`datetime`

Date-time

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

`format`

Alphanumeric

Is the format of the date-time field enclosed in parentheses and single quotation marks. It must be a date-time format (data type H, up to H23).
**length**

Integer

Is the number of characters in the alphanumeric field that is returned. You can supply the actual value, the name of a numeric field that contains the value, or an expression that returns the value. If *length* is smaller than the number of characters needed to display the alphanumeric field, the function returns a blank.

**output**

Alphanumeric

Is the field that contains the result, or the format of the output value enclosed in single quotation marks. This field must be in alphanumeric format and must be long enough to contain all of the characters returned.

**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.

**Example:** Converting a Date-Time Field to Alphanumeric Format (Reporting)

HCNVRT converts the TRANSDATE field to alphanumeric format. The first function does not include date-time display options for the field; the second function does for readability. It also specifies the display of seconds in the input field.

```
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
ALPHA_DATE_TIME1/A20 = HCNVRT(TRANSDATE, '(H17)', 17, 'A20');
ALPHA_DATE_TIME2/A20 = HCNVRT(TRANSDATE, '(HYYMDS)', 20, 'A20');
WHERE DATE EQ 2000
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>ALPHA_DATE_TIME1</th>
<th>ALPHA_DATE_TIME2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>200000205030000000</td>
<td>2000/02/05 03:30:00</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>200006260545000000</td>
<td>2000/06/26 05:45:00</td>
</tr>
</tbody>
</table>
Example: Converting a Date-Time Field to Alphanumeric Format (Maintain)

HCNVRT converts the DT1 field to alphanumeric format:

```
MAINTAIN FILE DATETIME
FOR ALL NEXT ID INTO STK;
COMPUTE
RESULT_HCNVRT/A20 = HCNVRT(STK.DT1, '(HYYMDH)', 20, RESULT_HCNVRT);
TYPE "$STK(1).DT1 = "$STK(1).DT1;
TYPE "$RESULT_HCNVRT = "$RESULT_HCNVRT;
END
```

HDATE: Converting the Date Portion of a Date-Time Value to a Date Format

Available Languages: reporting, Maintain

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 Date-Time Value to a Date Format

```
HDATE(datetime, output)
```

where:

**datetime**

Date-time

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

**output**

Date

Is the format in single quotation marks or the field that contains the result.

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.
Example: Converting the Date Portion of a Date-Time Field to a Date Format (Reporting)

HDATE converts the date portion of the TRANSDATE field to the date format YYMD:

```
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
TRANSDATE_DATE/YYMD = HDATE(TRANSDATE, 'YYMD');
WHERE DATE EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>TRANSDATE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>2000/02/05</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>2000/06/26</td>
</tr>
</tbody>
</table>

Example: Converting the Date Portion of a Date-Time Field to a Date Format (Maintain)

HDATE converts the date portion of DT1 to date format YYMD:

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
DT1_DATE/YYMD = HDATE(STK.DT1, DT1_DATE);
TYPE "STK(1).DT1 = <STK(1).DT1";
TYPE "DT1_DATE = <DT1_DATE";
END
```

The output is:

```
STK(1).DT1 = 2000/1/1 02:57:25
DT1_DATE = 2000/01/01
```

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

Available Languages: reporting, Maintain

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

Reference: Usage Notes for HDIFF

HDIFF does its subtraction differently from DATEDIF, which subtracts date components stored in date fields. The DATEDIF calculation looks for full years or full months. Therefore, subtracting the following two dates and requesting the number of months or years, results in 0:

```
DATE1 12/25/2014, DATE2 1/5/2015
```
Performing the same calculation using HDIFF on date-time fields results in a value of 1 month or 1 year as, in this case, the month or year is first extracted from each date-time value, and then the subtraction occurs.

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

```
HDIFF(end_dt, start_dt, 'component', output)
```

where:

**end_dt**

Date-time

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

**start_dt**

Date-time

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

**component**

Alphanumeric

Is the name of the component to be used in the calculation, enclosed in single quotation marks. If the component is a week, the WEEKFIRST parameter setting is used in the calculation.

**output**

Floating-point double-precision

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

\[
\text{DIFSEC/I9} = \text{HDIF(\text{DTCUR}, \text{DT(2004 February 16)}, 'SECOND', 'D9.0')}
\]

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.

**Example:** Finding the Number of Days Between Two Date-Time Fields (Reporting)

HDIFF calculates the number of days between the TRANSDATE and ADD_MONTH fields and stores the result in DIFF_PAYS, which has the format D12.2:

```
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
ADD_MONTH/HYYMDS = HADD(TRANSDATE, 'MONTH', 2, 8, 'HYYMDS');
DIFF_DAYS/D12.2 = HDIFF(ADD_MONTH, TRANSDATE, 'DAY', 'D12.2');
WHERE DATE EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>ADD_MONTH</th>
<th>DIFF_DAYS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>2000/04/05 03:30:00</td>
<td>60.00</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>2000/08/26 05:45:00</td>
<td>61.00</td>
</tr>
</tbody>
</table>

**Example:** Finding the Number of Days Between Two Date-Time Fields (Maintain)

HDIFF calculates the number of days between ADD_MONTH and DT1:

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
NEW_DATE/HYYMDS = HADD(STK.DT1, 'MONTH', 2, 10, NEW_DATE);
DIFF_DAYS/D12.2 = HDIFF(NEW_DATE, STK.DT1, 'DAY', DIFF_DAYS);
TYPE "STK(1).DT1 = "STK(1).DT1;
TYPE "NEW_DATE = "NEW_DATE;
TYPE "DIFF_DAYS = "DIFF_DAYS
END
```

**HDTTM: Converting a Date Value to a Date-Time Value**

Available Languages: reporting, Maintain

The HDTTM function converts a date value to a date-time value. The time portion is set to midnight.
How to Convert a Date Value to a Date-Time Value

HDTTM(date, length, output)

where:

date

Date

Is the date to be converted, the name of a date field that contains the value, or an expression that returns the value. It must be a full component format date. For example, it can be MDYY or YYJUL.

length

Integer

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 that includes microseconds.
- 12 indicates a time value that includes nanoseconds.

output

Date-time

Is the generated date-time value. It can be a field or the format of the output value enclosed in single quotation marks. The value must have a date-time format (data type H).

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.
Example: Converting a Date Field to a Date-Time Field (Reporting)

HDTTM converts the date field TRANSDATE_DATE to a date-time field:

```
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
TRANSDATE_DATE/YYMD = HDATE(TRANSDATE, 'YYMD');
DT2/HYMDIA = HDTTM(TRANSDATE_DATE, 8, 'HYMDIA');
WHERE DATE EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>TRANSDATE_DATE</th>
<th>DT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>2000/02/05</td>
<td>2000/02/05 12:00AM</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>2000/06/26</td>
<td>2000/06/26 12:00AM</td>
</tr>
</tbody>
</table>

Example: Converting a Date Field to a Date-Time Field (Maintain)

HDTTM converts the date field DT1_DATE to a date-time field:

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
DT1_DATE/YYMD = HDATE(DT1, DT1_DATE);
DT2/HYMDIA = HDTTM(DT1_DATE, 8, DT2);
TYPE "STK(1).DT1 = <STK(1).DT1";
TYPE "DT1_DATE = <DT1_DATE";
TYPE "DT2 = <DT2";
END
```

HGETC: Storing the Current Local Date and Time in a Date-Time Field

Available Languages: reporting, Maintain

The HGETC function returns the current local date and time 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 Local Date and Time in a Date-Time Field

HGETC(length, output)

where:

length

Integer

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 that includes microseconds.
- **12** indicates a time value that includes nanoseconds.

output

Date-time

Is the returned date-time value. Can be a field that contains the result, or the format of the output value enclosed in single quotation marks. The format must be in date-time format (data type H).

Example: Storing the Current Date and Time as a Timestamp

This example,

HGETC(8, 'HYYMDS')

creates a timestamp representing the current date and time.

Example: Storing the Current Date and Time in a Date-Time Field (Reporting)

HGETC stores the current date and time in DT2:

```
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
DT2/HYYMDm = HGETC(10, 'HYYMDm');
WHERE DATE EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>DT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>2000/10/03 15:34:24.00000000</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>2000/10/03 15:34:24.00000000</td>
</tr>
</tbody>
</table>
Example: Storing the Current Local Date and Time in a Date-Time Field (Maintain)

HGETC stores the current date and time in DT2:

```
MAINTAIN
COMPUTE DT2/HYYMDm = HGETC(10, DT2);
TYPE "DT2 = <DT2";
END
```

HGETZ: Storing the Current Coordinated Universal Time in a Date-Time Field

Available Languages: reporting, Maintain

HGETZ provides the current Coordinated Universal Time (UTC/GMT time, often called Zulu time). UTC is the primary civil time standard by which the world regulates clocks and time.

The value is returned 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 Universal Date and Time in a Date-Time Field

```
HGETZ(length, output)
```

where:

`length`

Integer

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 that includes microseconds.
- **12** indicates a time value that includes nanoseconds.

`output`

Date-time

Is the returned date-time value. Can be a field that contains the result, or the format of the output value enclosed in single quotation marks. The format must be in date-time format (data type H).
**Example:** Storing the Current Universal Date and Time as a Timestamp

This example,

```plaintext
HGETZ(8, 'HYYMDS')
```

creates a timestamp representing the current date and time.

**Example:** Storing the Current Universal Date and Time in a Date-Time Field (Reporting)

HGETZ stores the current universal date and time in DT2:

```plaintext
TABLE FILE VIDEOTRK
PRINT CUSTID AND COMPUTE
DT2/HYYMDm = HGETZ(10, 'HYYMDm');
WHERE CUSTID GE '2000' AND CUSTID LE '3000';
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2165</td>
<td>2015/05/08 14:43:08.740000</td>
</tr>
<tr>
<td>2187</td>
<td>2015/05/08 14:43:08.740000</td>
</tr>
<tr>
<td>2280</td>
<td>2015/05/08 14:43:08.740000</td>
</tr>
<tr>
<td>2282</td>
<td>2015/05/08 14:43:08.740000</td>
</tr>
<tr>
<td>2884</td>
<td>2015/05/08 14:43:08.740000</td>
</tr>
</tbody>
</table>

**Example:** Calculating the Time Zone

The time zone can be calculated as a positive or negative hourly offset from GMT. Locations to the west of the prime meridian have a negative offset. The following request uses the HGETC function to retrieve the local time, and the HGETZ function to retrieve the GMT time. The HDIFF function calculates the number of boundaries between them in minutes. The zone is found by dividing the minutes by 60:

```plaintext
DEFINE FILE EMPLOYEE
LOCALTIME/HYYMDS = HGETC(8, LOCALTIME);
UTCTIME/HYYMDS = HGETZ(8, UTCTIME);
MINUTES/D4= HDIFF(LOCALTIME, UTCTIME, 'MINUTES', 'D4');
ZONE/P3 = MINUTES/60;
END
TABLE FILE EMPLOYEE
PRINT EMP_ID NOPRINT OVER
LOCALTIME OVER
UTCTIME OVER
MINUTES OVER
ZONE
IF RECORDLIMIT IS 1
END
```
The output is:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCALTIME</td>
<td>2015/05/12 12:47:04</td>
</tr>
<tr>
<td>UTCTIME</td>
<td>2015/05/12 16:47:04</td>
</tr>
<tr>
<td>MINUTES</td>
<td>-240</td>
</tr>
<tr>
<td>ZONE</td>
<td>-4</td>
</tr>
</tbody>
</table>

**HHMMSS: Retrieving the Current Time**

Available Languages: reporting

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

A compiled MODIFY procedure must use HHMMSS to obtain the time; it cannot use the &TOD variable, which also returns the time. The &TOD variable is made current only when you execute a MODIFY, SCAN, or FSCAN procedure.

There is also an HHMMSS function available in the Maintain language. For information on this function, see *HHMMSS: Retrieving the Current Time (Maintain)*.

**Syntax:**

How to Retrieve the Current Time

`HHMMSS(output)`

where:

**output**

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.

**Example:**

Retrieving the Current Time

This example,

`HHMMSS('A10')`

creates a character string representing current time, like 12.09.47. Note that shorter output_format format will cause truncation of output.
**Example:** Retrieving the Current Time

HHMMSS retrieves the current time and displays it in the page footing:

```plaintext
TABLE FILE EMPLOYEE
SUM CURR_SAL AS 'TOTAL SALARIES' AND COMPUTE
NOWTIME/A8 = HHMMSS(NOWTIME); NOPRINT
BY DEPARTMENT
FOOTING
"SALARY REPORT RUN AT TIME <NOWTIME"
END
```

The output is:

<table>
<thead>
<tr>
<th>DEPARTMENT</th>
<th>TOTAL SALARIES</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIS</td>
<td>$108,002.00</td>
</tr>
<tr>
<td>PRODUCTION</td>
<td>$114,282.00</td>
</tr>
</tbody>
</table>

SALARY REPORT RUN AT TIME 15.21.14

**HHMS: Converting a Date-Time Value to a Time Value**

Available Languages: reporting

The HHMS function converts a date-time value to a time value.

**Syntax:** How to Convert a Date-Time Value to a Time Value

`HHMS(datetime, length, output)`

where:

- **datetime**
  - Date-time
  - Is the date-time value to be converted.

- **length**
  - Numeric
  - Is the length of the returned time value. Valid values are:
    - 8 indicates a time value that includes milliseconds.
    - 10 indicates a time value that includes microseconds.
    - 12 indicates a time value that includes nanoseconds.
**Example:** Converting a Date-Time Value to a Time value

The following example converts the date-time field TRANSDATE to a time field with time format HHIS,

```
DEFINE FILE VIDEOTR2
TRANSYEAR/I4 = HPART(TRANSDATE, 'YEAR', 'I4');
END
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
TRANS_TIME/HHIS = HHMS(TRANSDATE, 8, 'HHIS');
WHERE TRANSYEAR EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>TRANS_TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>05:45:00</td>
</tr>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>03:30:00</td>
</tr>
</tbody>
</table>

HHMS converts the date-time field TRANSDATE to a time value with format HHIS:

```
HHMS(TRANSDATE, 8, 'HHIS')
```

For 2000/06/26 05:45, the output is 05:45:00

**HINPUT: Converting an Alphanumeric String to a Date-Time Value**

Available Languages: reporting, Maintain

The HINPUT function converts an alphanumeric string to a date-time value.
**Syntax:**

How to Convert an Alphanumeric String to a Date-Time Value

\[ \text{HINPUT(} \text{source\_length}, \ 'source\_string', \ \text{output\_length}, \ \text{output}) \]

where:

**source\_length**

Integer

Is the number of characters in the source string to be converted. You can supply the actual value, the name of a numeric field that contains the value, or an expression that returns the value.

**source\_string**

Alphanumeric

Is the string to be converted enclosed in single quotation marks, the name of an alphanumeric field that contains the string, or an expression that returns the string. The string can consist of any valid date-time input value.

**output\_length**

Integer

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

- **8** indicates a time value that includes one to three decimal digits (milliseconds).
- **10** indicates a time value that includes four to six decimal digits (microseconds).
- **12** indicates a time value that includes seven to nine decimal digits (nanoseconds).

**output**

Date-time

Is the returned date-time value. Is a field that contains the result, or the format of the output value enclosed in single quotation marks. The format must be in date-time format (data type H).

**Example:**

Converting an Alphanumeric String to a Timestamp

This example,

\[ \text{DTM/}HYYMDS = \text{HINPUT(}14, \ '20040229 13:34:00', \ 8, \ \text{DTM}); \]

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.
**Example:** Converting an Alphanumeric String to a Date-Time Value (Reporting)

HCNVRT converts the TRANSDATE field to alphanumeric format, then HINPUT converts the alphanumeric string to a date-time value:

```plaintext
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
ALPHA_DATE_TIME/A20 = HCNVRT(TRANSDATE, '(H17)', 17, 'A20');
DT_FROM_ALPHA/HYYMDS = HINPUT(14, ALPHA_DATE_TIME, 8, 'HYYMDS');
WHERE DATE EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>ALPHA_DATE_TIME</th>
<th>DT_FROM_ALPHA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>20000205030000000</td>
<td>2000/02/05 03:30:00</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>20000626054500000</td>
<td>2000/06/26 05:45:00</td>
</tr>
</tbody>
</table>

**Example:** Converting an Alphanumeric String to a Date-Time Value (Maintain)

HINPUT converts the DT1 field to alphanumeric format:

```plaintext
MAINTAIN FILE DATETIME
COMPUTE
RESULT/HMtDYYmA = HINPUT(20, '19971029133059888999', 10, RESULT);
TYPE RESULT;
END
```

**HMIDNT: Setting the Time Portion of a Date-Time Value to Midnight**

Available Languages: reporting, Maintain

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

**Syntax:** How to Set the Time Portion of a Date-Time Value to Midnight

```plaintext
HMIDNT(datetime, length, output)
```

where:

**datetime**

Date-time

Is the date-time value whose time is to be set to midnight, the name of a date-time field that contains the value, or an expression that returns the value.
**length**

Integer

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 that includes microseconds.
- **12** indicates a time value that includes nanoseconds.

**output**

Date-time

Is the date-time return value whose time is set to midnight and whose date is copied from timestamp. Is the field that contains the result, or the format of the output value enclosed in single quotation marks. The format must be in date-time format (data type H).

**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:

```plaintext
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:

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

**Example:** Setting the Time to Midnight (Reporting)

HMIDNT sets the time portion of the TRANSDATE field to midnight first in the 24-hour system and then in the 12-hour system:

```plaintext
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
TRANSDATE_MID_24/HYYMDS  = HMIDNT(TRANSDATE, 8, 'HYYMDS');
TRANSDATE_MID_12/HYYMDSA = HMIDNT(TRANSDATE, 8, 'HYYMDSA');
WHERE DATE EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>TRANSDATE_MID_24</th>
<th>TRANSDATE_MID_12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1118</td>
<td>2000/06/26   05:45</td>
<td>2000/06/26 00:00:00</td>
<td>2000/06/26 12:00:00AM</td>
</tr>
<tr>
<td>1237</td>
<td>2000/02/05   03:30</td>
<td>2000/02/05 00:00:00</td>
<td>2000/02/05 12:00:00AM</td>
</tr>
</tbody>
</table>
**Example:** Setting the Time to Midnight (Maintain)

HMIIDNT sets the time portion of DT1 to midnight in both the 24-hour and 12-hour systems:

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
DT_MID_24/HYYMDS = HMIIDNT(STK(1).DT1, 8, DT_MID_24);
DT_MID_12/HYYMDSA = HMIIDNT(STK(1).DT1, 8, DT_MID_12);
TYPE "STK(1).DT1 = "STK(1).DT1;
TYPE "DT_MID_24 = "DT_MID_24";
TYPE "DT_MID_12 = "DT_MID_12";
END
```

**HNAME: Retrieving a Date-Time Component in Alphanumeric Format**

Available Languages: reporting, Maintain

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

**Syntax:** How to Retrieve a Date-Time Component in Alphanumeric Format

```
HNAME(datetime, 'component', output)
```

where:

datetime
Date-time

Is the date-time value from which a component value is to be extracted, the name of a date-time field containing the value that contains the value, or an expression that returns the value.

component
Alphanumeric

Is the name of the component to be retrieved enclosed in single quotation marks. For a list of valid components, see *Arguments for Use With Date and Time Functions* on page 354.

output
Alphanumeric, at least A2

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

The function converts a month argument to an abbreviation of the month name and converts and all other components to strings of digits only. 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);
```

**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);
```

**Example:** Retrieving the Week Component in Alphanumeric Format (Reporting)

HNAME returns the week in alphanumeric format from the TRANSDATE field. Changing the WEEKFIRST parameter setting changes the value of the component.

```
SET WEEKFIRST = 7
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
WEEK_COMPONENT/A10 = HNAME(TRANSDATE, 'WEEK', 'A10');
WHERE DATE EQ 2000;
END
```

When WEEKFIRST is set to seven, the output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>WEEK_COMPONENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>06</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>26</td>
</tr>
</tbody>
</table>

When WEEKFIRST is set to three, the output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>WEEK_COMPONENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>05</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>25</td>
</tr>
</tbody>
</table>

For details on WEEKFIRST, see the *Developing Reporting Applications* manual.

**Example:** Retrieving the Day Component in Alphanumeric Format (Reporting)

HNAME retrieves the day in alphanumeric format from the TRANSDATE field:

```
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
DAY_COMPONENT/A2 = HNAME(TRANSDATE, 'DAY', 'A2');
WHERE DATE EQ 2000;
END
```
The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>DAY_COMPONENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>05</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>26</td>
</tr>
</tbody>
</table>

**Example:** Retrieving the Day Component in Alphanumeric Format (Maintain)

HNAME extracts the day in alphanumeric format from DT1:

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
DAY_COMPONENT/A2=HNAME(STK.DT1,'DAY',DAY_COMPONENT);
TYPE "STK(1).DT1 = "STK(1).DT1;
TYPE "DAY_COMPONENT = <DAY_COMPONENT"
END
```

**HPART: Retrieving a Date-Time Component as a Numeric Value**

Available Languages: reporting, Maintain

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

**Syntax:** How to Retrieve a Date-Time Component in Numeric Format

```
HPART(datetime, 'component', output)
```

where:

**datetime**

Date-time

Is the date-time value from which the component is to be extracted, the name of a date-time field that contains the value, or an expression that returns the value.

**component**

Alphanumeric

Is the name of the component to be retrieved enclosed in single quotation marks. For a list of valid components, see *Arguments for Use With Date and Time Functions* on page 354.
output

Integer

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

**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:

\[
\text{IHOUR/I2} = \text{HPART} (\text{HGETC}(8, 'HYMDS'), 'HOUR', \text{IHOUR});
\]

**Example:** Retrieving the Day Component in Numeric Format (Reporting)

HPART retrieves the day in integer format from the TRANSDATE field:

```
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
DAY_COMPONENT/I2 = \text{HPART} (\text{TRANSDATE}, 'DAY', 'I2');
WHERE DATE EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>DAY_COMPONENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>5</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>26</td>
</tr>
</tbody>
</table>

**Example:** Retrieving the Day Component in Numeric Format (Maintain)

HPART extracts the day in integer format from DT1:

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
DAY_COMPONENT/I2 = \text{HPART} (\text{STK.DT1}, 'DAY', DAY_COMPONENT);
TYPE "STK(1).DT1 = <STK(1).DT1";
TYPE "DAY_COMPONENT = <DAY_COMPONENT";
END
```
HSETPT: Inserting a Component Into a Date-Time Value

Available Languages: reporting, Maintain

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

\[ \text{HSETPT}( \text{datetime}, \text{'component'}, \text{value}, \text{length}, \text{output}) \]

where:

- \textit{datetime}
  - Date-time
  - Is the date-time value in which to insert the component, the name of a date-time field that contains the value, or an expression that returns the value.

- \textit{component}
  - Alphanumeric
  - Is the name of the component to be inserted enclosed in single quotation marks. See \textit{Arguments for Use With Date and Time Functions} on page 354 for a list of valid components.

- \textit{value}
  - Integer
  - Is the numeric value to be inserted for the requested component, the name of a numeric field that contains the value, or an expression that returns the value.

- \textit{length}
  - Integer
  - Is the length of the returned date-time value. Valid values are:
    - 8 indicates a time value that includes one to three decimal digits (milliseconds).
    - 10 indicates a time value that includes four to six decimal digits (microseconds).
    - 12 indicates a time value that includes seven to nine decimal digits (nanoseconds).
output

Date-time

Is the returned date-time value whose chosen component is updated. All other
components are copied from the source date-time value.

Is the field that contains the result, or the format of the output value enclosed in single
quotation marks. The format must be in date-time format (data type H).

**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.

**Example:** Inserting the Day Component Into a Date-Time Field (Reporting)

HSETPT inserts the day as 28 into the ADD.MONTH field and stores the result in INSERT.DAY:

```
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
ADD_MONTH/HYYMDS = HADD(TRANSDATE, 'MONTH', 2, 8, 'HYYMDS');
INSERT_DAY/HYYMDS = HSETPT(ADD_MONTH, 'DAY', 28, 8, 'HYYMDS');
WHERE DATE EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>ADD_MONTH</th>
<th>INSERT_DAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>2000/08/26 05:45:00</td>
<td>2000/08/28 05:45:00</td>
</tr>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>2000/04/05 03:30:00</td>
<td>2000/04/28 03:30:00</td>
</tr>
</tbody>
</table>

**Example:** Inserting the Day Component Into a Date-Time Field (Maintain)

HSETPT inserts the day into ADD.MONTH:

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
ADD_MONTH/HYYMDS = HADD(STK.DT1, 'MONTH', 2, 8, ADD_MONTH);
INSERT_DAY/HYYMDS = HSETPT(ADD_MONTH, 'DAY', 28, 8, INSERT_DAY);
TYPE "STK(1).DT1 = <STK(1).DT1";
TYPE "ADD_MONTH = <ADD_MONTH";
TYPE "INSERT_DAY = <INSERT_DAY";
END
```
HTIME: Converting the Time Portion of a Date-Time Value to a Number

Available Languages: reporting, Maintain

The HTIME function converts the time portion of a date-time value to the number of milliseconds if the length argument is eight, microseconds if the length argument is ten, or nanoseconds if the length argument is 12.

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

HTIME(length, datetime, output)

where:

length

Integer

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

- **8** indicates a time value that includes one to three decimal digits (milliseconds).
- **10** indicates a time value that includes four to six decimal digits (microseconds).
- **12** indicates a time value that includes seven to nine decimal digits (nanoseconds).

datetime

Date-time

Is the date-time value from which to convert the time, the name of a date-time field that contains the value, or an expression that returns the value.

output

Floating-point double-precision

Is the field that contains the result, or the format of the output value enclosed in single quotation marks. The format must be floating-point double-precision.

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.)

\[\text{NMILLI}/\text{D12.0} = \text{HTIME}(8, \text{HGETC}(10,'HYYMDS'), \text{NMICRO});\]

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.
Example: Converting the Time Portion of a Date-Time Field to a Number (Reporting)

HTIME converts the time portion of the TRANSDATE field to the number of milliseconds:

```
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
MILLISEC/D12.2 = HTIME(8, TRANSDATE, 'D12.2');
WHERE DATE EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>MILLISEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>12,600,000.00</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>20,700,000.00</td>
</tr>
</tbody>
</table>

Example: Converting the Time Portion of a Date-Time Field to a Number (Maintain)

HTIME converts the time portion of the DT1 field to the number of milliseconds:

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE MILLISEC/D12.2 = HTIME(8, STK.DT1, MILLISEC);
TYPE "STK(1).DT1 = <STK(1).DT1";
TYPE "MILLISEC = <MILLISEC";
END
```

HTMTOTS or TIMETOTS: Converting 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 date-time value.

Note: TIMETOTS is a synonym for HTMTOTS.

Syntax: How to Convert a Time to a Timestamp

```
HTMTOTS(time, length, output)
```

or

```
TIMETOTS(time, length, output)
```
where:

\textit{time}

\textbf{Date-Time}

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

\textit{length}

\textbf{Integer}

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

- \textbf{8} for input time values including milliseconds.
- \textbf{10} for input time values including microseconds.
- \textbf{12} for input time values including nanoseconds.

\textit{output\_format}

\textbf{Date-Time}

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

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

\textbf{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:

\begin{verbatim}
HMDYYS = HTMTOTS(DT(&MYTOD), 8, 'HMDYYS');
\end{verbatim}


\textbf{Example: Converting a Time to a Timestamp}

HTMTOTS converts the time portion of the TRANSDATE field to a timestamp, using the current date for the date portion of the returned value:

\begin{verbatim}
DEFINE FILE VIDEOTR2
   TSTMPSEC/HYYMDS = HTMTOTS(TRANSDATE, 8, 'HYYMDS');
END
TABLE FILE VIDEOTR2
PRINT TRANSDATE TSTMPSEC
BY LASTNAME BY FIRSTNAME
WHERE DATE EQ '1991'
END
\end{verbatim}
### HYYWD: Returning 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

```plaintext
HYYWD(dtvalue, output)
```

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.

---

<table>
<thead>
<tr>
<th>LASTNAME</th>
<th>FIRSTNAME</th>
<th>TRANSDATE</th>
<th>TSTMPSEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRUZ</td>
<td>IVY</td>
<td>1991/06/27 02:45</td>
<td>2011/01/11 02:45:00</td>
</tr>
<tr>
<td>GOODMAN</td>
<td>JOHN</td>
<td>1991/06/25 01:19</td>
<td>2011/01/11 01:19:00</td>
</tr>
<tr>
<td>GREEVEN</td>
<td>GEORGIA</td>
<td>1991/06/24 10:27</td>
<td>2011/01/11 10:27:00</td>
</tr>
<tr>
<td>HANDLER</td>
<td>EVAN</td>
<td>1991/06/20 05:15</td>
<td>2011/01/11 05:15:00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1991/06/21 07:11</td>
<td>2011/01/11 07:11:00</td>
</tr>
<tr>
<td>KRAMER</td>
<td>CHERYL</td>
<td>1991/06/21 01:10</td>
<td>2011/01/11 01:10:00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1991/06/19 07:18</td>
<td>2011/01/11 07:18:00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1991/06/19 04:11</td>
<td>2011/01/11 04:11:00</td>
</tr>
<tr>
<td>MONROE</td>
<td>CATHERINE</td>
<td>1991/06/25 01:17</td>
<td>2011/01/11 01:17:00</td>
</tr>
<tr>
<td>SPIVEY</td>
<td>TOM</td>
<td>1991/11/17 11:28</td>
<td>2011/01/11 11:28:00</td>
</tr>
<tr>
<td>WILLIAMS</td>
<td>KENNETH</td>
<td>1991/06/24 04:43</td>
<td>2011/01/11 04:43:00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1991/06/24 02:08</td>
<td>2011/01/11 02:08:00</td>
</tr>
</tbody>
</table>
output

Alphanumeric

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

The output format 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 converts the TRANSDATE date-time value to the ISO standard format for dates with week numbers. WEEKFIRST is set to ISO2, which produces ISO standard week numbering:

```
ISODATE/A10 = HYYWD(TRANSDATE, 'A10');
```

For date component 1999/01/30 04:16, the value is 1999-W04-6.

For date component 1999/12/15, the value is 1999-W50-3.
**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:

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

**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:

<table>
<thead>
<tr>
<th>TRANSDATE</th>
<th>TRANSTOT</th>
<th>QUANTITY</th>
<th>ISODATE</th>
<th>WEEK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1991/06/24</td>
<td>04:43</td>
<td>16.00</td>
<td>2</td>
<td>1991-W26-1</td>
</tr>
<tr>
<td>1991/06/25</td>
<td>01:17</td>
<td>2.50</td>
<td>2</td>
<td>1991-W26-2</td>
</tr>
<tr>
<td>1991/06/27</td>
<td>02:45</td>
<td>16.00</td>
<td>2</td>
<td>1991-W26-4</td>
</tr>
</tbody>
</table>
Simplified Conversion Functions

Simplified conversion functions have streamlined parameter lists, similar to those used by SQL functions. In some cases, these simplified functions provide slightly different functionality than previous versions of similar functions.

The simplified functions do not have an output argument. Each function returns a value that has a specific data type.

When used in a request against a relational data source, these functions are optimized (passed to the RDBMS for processing).

In this chapter:

- CHAR: Returning a Character Based on a Numeric Code
- COMPACTFORMAT: Displaying Numbers in an Abbreviated Format
- CTRLCHAR: Returning a Non-Printable Control Character
- FPRINT: Displaying a Value in a Specified Format
- HEXTYPE: Returning the Hexadecimal View of an Input Value
- PHONETIC: Returning a Phonetic Key for a String
- TO_INTEGER: Converting a Character String to an Integer Value
- TO_NUMBER: Converting a Character String to a Numeric Value

CHAR: Returning a Character Based on a Numeric Code

The CHAR function accepts a decimal integer and returns the character identified by that number converted to ASCII or EBCDIC, depending on the operating environment. The output is returned as variable length alphanumeric. If the number is above the range of valid characters, a null value is returned.

Syntax: How to Return a Character Based on a Numeric Code

```
CHAR(number_code)
```
where:

\[ \text{number\_code} \]

Integer

Is a field, number, or numeric expression whose whole absolute value will be used as a number code to retrieve an output character.

For example, a TAB character is returned by CHAR(9) in ASCII environments, or by CHAR(5) in EBCDIC environments.

**Example:** Using the CHAR Function to Insert Control Characters Into a String

The following request defines a field with carriage return (CHAR(13)) and line feed (CHAR(10)) characters inserted between the words HELLO and GOODBYE (in an ASCII environment). To show that these characters were inserted, the output is generated in PDF format and the StyleSheet attribute LINEBREAK='CRLF' is used to have these characters respected and print the field value on two lines.

```
DEFINE FILE WF_RETAIL_LITE
MYFIELD/A20 WITH COUNTRY_NAME='HELLO' | CHAR(13) | CHAR(10) | 'GOODBYE';
END
TABLE FILE WF_RETAIL_LITE
SUM MYFIELD
ON TABLE PCHOLD FORMAT PDF
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
TYPE=REPORT, LINEBREAK='CRLF',$
ENDSTYLE
END
```

The output is shown in the following image.

```
MYFIELD
HELLO
GOODBYE
```

CHAR returns a carriage control character in an ASCII environment.

CHAR(13)

**COMPACTFORMAT: Displaying Numbers in an Abbreviated Format**

COMPACTFORMAT displays numbers in a compact format where:

- K is an abbreviation for thousands.
- M is an abbreviation for millions.
- B is an abbreviation for billions.
- T is an abbreviation for trillions.

COMPACTFORMAT computes which abbreviation to use, based on the order of magnitude of the largest value in the column. The returned value is an alphanumeric string. Attempting to output this value to a numeric format will result in a format error, and the value zero (0) will be displayed.

**Syntax:**

**How to Display Numbers in an Abbreviated Format**

COMPACTFORMAT(input)

where:

input

Is the name of a numeric field.

**Example:**

**Displaying Numbers in an Abbreviated Format**

The following example uses the COMPACTFORMAT function to abbreviate the display of the summed values of the DAYSDELAYED, QUANTITY_SOLD, and COGS_US fields.

TABLE FILE WF_RETAIL_LITE
SUM DAYSDELAYED QUANTITY_SOLD COGS_US
COMPUTE
CDAYS/A30= COMPACTFORMAT(DAYSDELAYED);
CQUANT/A30= COMPACTFORMAT(QUANTITY_SOLD);
CCOGS/A30= COMPACTFORMAT(COGS_US);
ON TABLE SET PAGE NOPAGE
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END

The output is shown in the following image.

COMPACTFORMAT abbreviates the display of COGS_US.

COMPACTFORMAT(COGS_US)
For $2,950,358.00, the result is $3M.

CTRLCHAR: Returning a Non-Printable Control Character

The CTRLCHAR function returns a nonprintable control character specific to the running operating environment, based on a supported list of keywords. The output is returned as variable length alphanumeric.

Syntax: How to Return a Non-Printable Control Character

CTRLCHAR(ctrl_char)

where:

ctrl_char

Is one of the following keywords.

- **NUL** returns a null character.
- **SOH** returns a start of heading character.
- **STX** returns a start of text character.
- **ETX** returns an end of text character.
- **EOT** returns an end of transmission character.
- **ENQ** returns an enquiry character.
- **ACK** returns an acknowledge character.
- **BEL** returns a bell or beep character.
- **BS** returns a backspace character.
- **TAB** or **HT** returns a horizontal tab character.
- **LF** returns a line feed character.
- **VT** returns a vertical tab character.
- **FF** returns a form feed (top of page) character.
- **CR** returns a carriage control character.
- **SO** returns a shift out character.
- **SI** returns a shift in character.
DLE returns a data link escape character.

DC1 or XON returns a device control 1 character.

DC2 returns a device control 2 character.

DC3 or XOFF returns a device control 3 character.

DC4 returns a device control 4 character.

NAK returns a negative acknowledge character.

SYN returns a synchronous idle character.

ETB returns an end of transmission block character.

CAN returns a cancel character.

EM returns an end of medium character.

SUB returns a substitute character.

ESC returns an escape, prefix, or altmode character.

FS returns a file separator character.

GS returns a group separator character.

RS returns a record separator character.

US returns a unit separator character.

DEL returns a delete, rubout, or interrupt character.
**Example:** Using the CTRLCHAR Function to Insert Control Characters Into a String

The following request defines a field with carriage return (CTRLCHAR(CR)) and line feed (CTRLCHAR(LF)) characters inserted between the words HELLO and GOODBYE. To show that these characters were inserted, the output is generated in PDF format and the StyleSheet attribute LINEBREAK='CRLF' is used to have these characters respected and print the field value on two lines.

```
DEFINE FILE WF_RETAIL_LITE
MYFIELD/A20 WITH COUNTRY_NAME='HELLO' | CTRLCHAR(CR) | CTRLCHAR(LF) | 'GOODBYE';
END
TABLE FILE WF_RETAIL_LITE
SUM MYFIELD
ON TABLE PCHOLD FORMAT PDF
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
TYPE=REPORT, LINEBREAK='CRLF', 
ENDSTYLE
END
```

The output is shown in the following image.

```
MYFIELD

HELLO
GOODBYE
```

CTRLCHAR returns a carriage control character in an ASCII environment.

**CTRLCHAR (CR)**

**FPRINT: Displaying a Value in a Specified Format**

Given an output format, the simplified conversion function FPRINT converts a value to alphanumeric format for display.

**Note:** A legacy FPRINT function also exists and is still supported. For information, see **FPRINT: Converting Fields to Alphanumeric Format** on page 414. The legacy function has an additional argument for the name or format of the returned value.

**Syntax:** How to Display a Value in a Specified Format

```
FPRINT(value, 'out_format')
```
where:

\textit{value}

Any data type

Is the value to be converted.

\textit{\texttt{\textquoteleft out\_format\textquoteright}}

Fixed length alphanumeric

Is the display format. For information about valid display formats, see the \textit{Describing Data With WebFOCUS Language}manual.

**Example:** Displaying a Value in a Specified Format

The following request displays \texttt{COGS\_US} as format 'D9M', and \texttt{TIME\_DATE} as format 'YYMtrD', by converting them to alphanumeric using \texttt{FPRINT}.

```plaintext
DEFINE FILE WF\_RETAIL\_LITE
COGS\_A/A25 = FPRINT(COGS\_US, \textquoteleft D9M\textquoteright);
DATE1/A25 = FPRINT(TIME\_DATE, \textquoteleft YYMtrD\textquoteright);
END
TABLE FILE WF\_RETAIL\_LITE
PRINT LST.COGS\_US COGS\_A DATE1
BY TIME\_DATE
WHERE RECORDLIMIT EQ 10
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```
The output is shown in the following image.

FPRINT converts a date to alphanumeric format.

FPRINT(TIME_DATE, 'YYMtrD')

For 01/03/2009, the result is 2009, January 3.

**HEXTYPE: Returning the Hexadecimal View of an Input Value**

The HEXTYPE function returns the hexadecimal view of an input value of any data type. The result is returned as variable length alphanumeric. The alphanumeric field to which the hexadecimal value is returned must be large enough to hold two characters for each input character. The value returned depends on the running operating environment.

**Syntax:**

How to Returning the Hexadecimal View of an Input Value

HEXTYPE(in_value)

where:

in_value

Is an alphanumeric or integer field, constant, or expression.
Example: Returning a Hexadecimal View

The following request returns a hexadecimal view of the country names and the sum of the days delayed.

```
DEFINE FILE WF_RETAIL_LITE
Days/I8 = DAYSDELAYED;
Country/A20 = COUNTRY_NAME;
HexCountry/A30 = HEXTYPE(Country);
END
TABLE FILE WF_RETAIL_LITE
SUM COUNTRY_NAME NOPRINT  Country HexCountry Days
COMPUTE HexDays/A40 = HEXTYPE(Days);
BY COUNTRY_NAME NOPRINT
WHERE COUNTRY_NAME LT 'P'
ON TABLE SET PAGE NOPAGE
END
```
HEXTYPE returns a hexadecimal view of COUNTRY_NAME.

**HEXTYPE (COUNTRY_NAME)**

For Argentina, the result is 417267656E74696E61.
**PHONETIC: Returning a Phonetic Key for a String**

PHONETIC calculates a phonetic key for a string, or a null value on failure. Phonetic keys are useful for grouping alphanumeric values, such as names, that may have spelling variations. This is done by generating an index number that will be the same for the variations of the same name based on pronunciation. One of two phonetic algorithms can be used for indexing, Metaphone and Soundex. Metaphone is the default algorithm, except on z/OS where the default is Soundex.

You can set the algorithm to use with the following command.

```sql
SET PHONETIC_ALGORITHM = {METAPHONE | SOUNDEX}
```

Most phonetic algorithms were developed for use with the English language. Therefore, applying the rules to words in other languages may not give a meaningful result.

Metaphone is suitable for use with most English words, not just names. Metaphone algorithms are the basis for many popular spell checkers.

**Note:** Metaphone is not optimized in generated SQL. Therefore, if you need to optimize the request for an SQL DBMS, the SOUNDEX setting should be used.

Soundex is a legacy phonetic algorithm for indexing names by sound, as pronounced in English.

**Syntax:** How to Return a Phonetic Key

```sql
PHONETIC(string)
```

where:

- **string**
  - Alphanumeric

  Is a string for which to create the key. A null value will be returned on failure.
**Example: Generating a Phonetic Key**

The following request changes the spelling of the last name for MARY SMITH to SMYTHE and generates a phonetic key for each last name.

```
DEFINE FILE EMPLOYEE
LAST_NAME2/A16 = IF LAST_NAME EQ 'SMITH' AND FIRST_NAME EQ 'MARY' THEN 'SMYTHE' ELSE LAST_NAME;
PKEY/A10 = PHONETIC(LAST_NAME2);
END
TABLE FILE EMPLOYEE
PRINT FIRST_NAME LAST_NAME2
BY PKEY
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image. Note that the two spellings for SMITH are assigned the same index number.

<table>
<thead>
<tr>
<th>PKEY</th>
<th>FIRST_NAME</th>
<th>LAST_NAME2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B423</td>
<td>ROSEMARIE</td>
<td>BLACKWOOD</td>
</tr>
<tr>
<td>B552</td>
<td>JOHN</td>
<td>BANNING</td>
</tr>
<tr>
<td>C620</td>
<td>BARBARA</td>
<td>CROSS</td>
</tr>
<tr>
<td>G652</td>
<td>MARY</td>
<td>GREENSPAN</td>
</tr>
<tr>
<td>I615</td>
<td>JOAN</td>
<td>IRVING</td>
</tr>
<tr>
<td>J520</td>
<td>DIANE</td>
<td>JONES</td>
</tr>
<tr>
<td>M200</td>
<td>JOHN</td>
<td>MCOY</td>
</tr>
<tr>
<td>M252</td>
<td>ROGER</td>
<td>MCKNIGHT</td>
</tr>
<tr>
<td>R552</td>
<td>ANTHONY</td>
<td>ROMANS</td>
</tr>
<tr>
<td>S315</td>
<td>ALFRED</td>
<td>STEVENS</td>
</tr>
<tr>
<td>S530</td>
<td>MARY</td>
<td>SMYTHE</td>
</tr>
<tr>
<td></td>
<td>RICHARD</td>
<td>SMITH</td>
</tr>
</tbody>
</table>

PHONETIC generates a phonetic key for LAST_NAME:

```
PHONETIC(LAST_NAME)
```

For last names SMITH and SMYTHE, the same phonetic key, S530, is generated.
**TO_INTEGER: Converting a Character String to an Integer Value**

TO_INTEGER converts a character string that contains a valid number consisting of digits and an optional decimal point to an integer value. If the value contains a decimal point, the value after the decimal point is truncated. If the value does not represent a valid number, zero (0) is returned.

**Syntax:**

How to Convert a Character String to an Integer

```plaintext
TO_INTEGER(string)
```

where:

- **string**
  
  Is a character string enclosed in single quotation marks or a character field that represents a number containing digits and an optional decimal point.

**Example:**

Converting a Character String to an Integer Value

The following request converts character strings to integers. Digits following the decimal point are truncated.

```plaintext
DEFINE FILE WF_RETAIL_LITE
INT1/I8 = TO_INTEGER('56.78');
INT2/I8 = TO_INTEGER('.5678');
INT3/I8 = TO_INTEGER('5678');
END

TABLE FILE WF_RETAIL_LITE
PRINT INT1 INT2 INT3
BY BUSINESS_REGION AS Region
WHERE READLIMIT EQ 1
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image.

<table>
<thead>
<tr>
<th>Region</th>
<th>INT1</th>
<th>INT2</th>
<th>INT3</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMEA</td>
<td>56</td>
<td>0</td>
<td>5678</td>
</tr>
</tbody>
</table>

TO_INTEGER converts the character string '56.78' to an integer.

```plaintext
TO_INTEGER('56.78')
```

The result is 56.
TO_NUMBER: Converting a Character String to a Numeric Value

TO_NUMBER converts a character string that contains a valid number consisting of digits and an optional decimal point to the numeric format most appropriate to the context. If the value does not represent a valid number, zero (0) is returned.

Syntax: How to Convert a Character String to a Number

TO_NUMBER(string)

where:

string
Is a character string enclosed in single quotation marks or a character field that represents a number containing digits and an optional decimal point. This string will be converted to a double-precision floating point number.

Example: Converting a Character String to a Number

The following request converts character strings to double-precision floating point numbers.

DEFINE FILE WF_RETAIL_LITE
NUM1/D12.1 = TO_NUMBER('56.78');
NUM2/D12.2 = TO_NUMBER('0.5678');
END
TABLE FILE WF_RETAIL_LITE
PRINT NUM1 NUM2
BY BUSINESS_REGION AS Region
WHERE READLIMIT EQ 1
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END

The output is shown in the following image.

<table>
<thead>
<tr>
<th>Region</th>
<th>NUM1</th>
<th>NUM2</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMEA</td>
<td>56.8</td>
<td>.57</td>
</tr>
</tbody>
</table>

TO_NUMBER converts the string '56.78' to a number with one decimal place.

TO_NUMBER('56.78')

The result is 56.8.
Chapter 12

Format Conversion Functions

Format conversion functions convert fields from one format to another.

For information on field formats see the Describing Data With WebFOCUS Language manual.

For information on field formats see the Describing Data manual.

For many functions, the output argument can be supplied either as a field name or as a format enclosed in single quotation marks. However, if a function is called from a Dialogue Manager command, this argument must always be supplied as a format, and if a function is called from a Maintain Data procedure, this argument must always be supplied as a field name.

In this chapter:

- ATODBL: Converting an Alphanumeric String to Double-Precision Format
- EDIT: Converting the Format of a Field
- FPRINT: Converting Fields to Alphanumeric Format
- FTOA: Converting a Number to Alphanumeric Format
- HEXBYT: Converting a Decimal Integer to a Character
- ITONUM: Converting a Large Binary Integer Number to Double-Precision Format
- ITOPACK: Converting a Large Binary Integer to Packed-Decimal Format
- ITOZ: Converting a Number to Zoned Format
- PCKOUT: Writing a Packed Number of Variable Length
- PTOA: Converting a Packed-Decimal Number to Alphanumeric Format
- TSTOPACK: Converting an MSSQL or Sybase Timestamp Column to Packed Decimal
- UFMT: Converting an Alphanumeric String to Hexadecimal
- XTPACK: Writing a Packed Number With Up to 31 Significant Digits to an Output File
ATODBL: Converting an Alphanumeric String to Double-Precision Format

Available Languages: reporting, Maintain

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)

where:

* source_string
  Alphanumeric
  Is the string consisting of digits and, optionally, one sign and one decimal point to be converted, or a field or variable that contains the string.

* length
  Alphanumeric
  Is the two-character 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'.

* output
  Double precision floating-point
  Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.

**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.

ATODBL converts the EMP_ID field into double-precision format and stores the result in D_EMP_ID:

```
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND FIRST_NAME AND EMP_ID
AND COMPUTE D_EMP_ID/D12.2 = ATODBL(EMP_ID, '09', D_EMP_ID);
WHERE DEPARTMENT EQ 'MIS';
END
```
The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>EMP_ID</th>
<th>D_EMP_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>112847612</td>
<td>112,847,612.00</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>117593129</td>
<td>117,593,129.00</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>219984371</td>
<td>219,984,371.00</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>326179357</td>
<td>326,179,357.00</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>543729165</td>
<td>543,729,165.00</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>818692173</td>
<td>818,692,173.00</td>
</tr>
</tbody>
</table>

Example: Converting an Alphanumeric Value to Double-Precision Format With MODIFY

In the following example, the Master File contains the MISSING attribute for the CURR_SAL field. If you do not enter a value for this field, it is interpreted as the default value, a period.

FILENAME=EMPLOYEE, SUFFIX=FOC
SEGNAME=EMPINFO, SEGTYPE=S1
FIELDNAME=EMP_ID, ALIAS=EID, FORMAT=A9, $ . .
FIELDNAME=CURR_SAL, ALIAS=CSAL, FORMAT=D12.2M, MISSING=ON,$ . .

ATODBL converts the value supplied for TCSAL to double-precision format:

MODIFY FILE EMPLOYEE
COMPUTE TCSAL/A12=;
PROMPT EID
MATCH EID
ON NOMATCH REJECT
ON MATCH TYPE "EMPLOYEE <D.LAST_NAME <D.FIRST_NAME"
ON MATCH TYPE "ENTER CURRENT SALARY OR 'N/A' IF NOT AVAILABLE"
ON MATCH PROMPT TCSAL
ON MATCH COMPUTE
CSAL MISSING ON = IF TCSAL EQ 'N/A' THEN MISSING
ELSE ATODBL(TCSAL, '12', 'D12.2');
ON MATCH TYPE "SALARY NOW <CSAL"
DATA

Functions Reference
A sample execution is:

```
EMPLOYEE    ON 11/14/96 AT 13.42.55
DATA FOR TRANSACTION    1
EMP_ID      =
071382660
EMPLOYEE STEVENS ALFRED
ENTER CURRENT SALARY OR 'N/A' IF NOT AVAILABLE
TCSAL       =
N/A
SALARY NOW              .
DATA FOR TRANSACTION    2
EMP_ID      =
112847612
EMPLOYEE SMITH MARY
ENTER CURRENT SALARY OR 'N/A' IF NOT AVAILABLE
TCSAL       =
45000
SALARY NOW $45,000.00
DATA FOR TRANSACTION    3
EMP_ID      =
end
TRANSACTIONS:         TOTAL =     2  ACCEPTED=     2  REJECTED=     0
SEGMENTS:             INPUT =     0  UPDATED =     0  DELETED =     0
```

The procedure processes as follows:

1. For the first transaction, the procedure prompts for an employee ID. You enter 071382660.
2. The procedure displays the last and first name of the employee, STEVENS ALFRED.
3. The procedure prompts for a current salary. You enter N/A.
4. A period displays.
5. For the second transaction, the procedure prompts for an employee ID. You enter 112847612.
6. The procedure displays the last and first name of the employee, SMITH MARY.
7. Then it prompts for a current salary. Enter 45000.
8. $45,000.00 displays.

**EDIT: Converting the Format of a Field**

Available Languages: reporting

The EDIT function converts an alphanumeric field that contains numeric characters to numeric format or converts a numeric field to alphanumeric format.

This function is useful for manipulating a field in an expression that performs an operation that requires operands in a particular format.
When EDIT assigns a converted value to a new field, the format of the new field must correspond to the format of the returned value. For example, if EDIT converts a numeric field to alphanumeric format, you must give the new field an alphanumeric format:

```
DEFINE ALPHAPRICE/A6 = EDIT(PRICE);
```

EDIT deals with a symbol in the following way:

- When an alphanumeric field is converted to numeric format, a sign or decimal point in the field is stored as part of the numeric value.
  
  Any other non-numeric characters are invalid, and EDIT returns the value zero.

- When converting a floating-point or packed-decimal field to alphanumeric format, EDIT removes the sign, the decimal point, and any number 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.

EDIT also extracts characters from or add characters to an alphanumeric string. For more information, see *EDIT: Extracting or Adding Characters* on page 144.

**Syntax:** How to Convert the Format of a Field

```
EDIT(fieldname);
```

where:

- `fieldname`  
  Alphanumeric or Numeric  
  Is the field name.

**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 is 820401.

For 81/11/02, the result is 811102.
EDIT converts HIRE_DATE (a legacy date format) to alphanumeric format. CHGDAT is then able to use the field, which it expects in alphanumeric format:

```
TABLE FILE EMPLOYEE
PRINT HIRE_DATE AND COMPUTE
ALPHA_HIRE/A17 = EDIT(HIRE_DATE); NOPRINT AND COMPUTE
HIRE_MDY/A17 = CHGDAT('YMD', 'MDYYX', ALPHA_HIRE, 'A17');
BY LAST_NAME BY FIRST_NAME
WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

```
LAST_NAME  FIRST_NAME  HIRE_DATE  HIRE_MDY
---------  ----------  ---------  --------
BLACKWOOD  ROSEMARIE   82/04/01   APRIL 01 1982
CROSS      BARBARA     81/11/02   NOVEMBER 02 1981
GREENSPAN  MARY        82/04/01   APRIL 01 1982
JONES      DIANE       82/05/01   MAY 01 1982
MCCOY      JOHN        81/07/01   JULY 01 1981
SMITH      MARY        81/07/01   JULY 01 1981
```

### FPRINT: Converting Fields to Alphanumeric Format

The FPRINT function converts any type of field except for a text field to its alphanumeric equivalent for display. The alphanumeric representation will include any display options that are specified in the format of the original field.

Available Languages: reporting

**Syntax:**

```
FPRINT(in_value, 'usageformat', output)
```

where:

- **in_value**
  - Any format except TX
  - Is the value to be converted.

- **usageformat**
  - Alphanumeric
  - Is the usage format of the value to be converted, including display options. The format must be enclosed in single quotation marks.

- **output**
  - Alphanumeric
Is the name of the output field or its format enclosed in single quotation marks.

The output format must be long enough to hold the converted number itself, with a sign and decimal point, plus any additional characters generated by display options, such as commas, a currency symbol, or a percent sign.

For example, D12.2 format is converted to A14 because it outputs two decimal digits, a decimal point, a possible minus sign, up to eight integer digits, and two commas. If the output format is not large enough, excess right-hand characters may be truncated.

Reference: Usage Notes for the FPRINT Function

- The USAGE format must match the actual data in the field.
- The output of FPRINT for numeric values is right-justified within the area required for the maximum number of characters corresponding to the supplied format. This ensures that all possible values are aligned vertically along the decimal point or units digit.
- By default, the column title is left justified for alphanumeric fields. To right justify the column title, use the /R reformatting option for the field.
- Maintain Data does not support the FPRINT function. However, you can do the same type of conversion in Maintain Data using the COMPUTE command.

Example: Converting a Numeric Field to Alphanumeric Format

FPRINT converts CURR_SAL (format D12.2)M to a column with format A15:

FPRINT(CURR_SAL, 'D12.2M', 'A15')

Example: Converting Numeric Fields to Alphanumeric Format

The following request against the EMPLOYEE data source uses FPRINT to convert the CURR_SAL, ED_HRS, and BANK_ACCT fields to alphanumeric for display on the report output. Then, the STRREP function replaces the blanks in the alphanumeric representation of CURR_SAL with asterisks. CURR_SAL has format D12.2M, so the alphanumeric representation has format A15. The ED_HRS field has format F6.2, so the alphanumeric representation has format A6. The BANK_ACCT field has format I9S, so the alphanumeric representation has format A9. The alphanumeric representations of the numeric fields are right-justified. The /R options in the PRINT command cause the column titles to be right-justified over the values:
DEFINE FILE EMPLOYEE
ASAL/A15 = FPRINT(CURR_SAL, 'D12.2M', ASAL);
ASAL/A15 = STRREP(15, ASAL, 1, ' ', 1, '*', 15, ASAL);
AED/A6 = FPRINT(ED_HRS, 'F6.2', AED);
ABANK/A9 = FPRINT(BANK_ACCT, 'I9S', ABANK);
END
TABLE FILE EMPLOYEE
PRINT CURR_SAL ASAL
ED_HRS AED/R
BANK_ACCT ABANK/R
WHERE BANK_NAME NE ' ' 
ON TABLE SET PAGE NOPAGE
END

The output is:

<table>
<thead>
<tr>
<th>CURR_SAL</th>
<th>ASAL</th>
<th>ED_HRS</th>
<th>AED</th>
<th>BANK_ACCT</th>
<th>ABANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>$18,480.00</td>
<td>*****$18,480.00</td>
<td>50.00</td>
<td>50.00</td>
<td>40950036</td>
<td>40950036</td>
</tr>
<tr>
<td>$29,700.00</td>
<td>*****$29,700.00</td>
<td>.00</td>
<td>.00</td>
<td>160633</td>
<td>160633</td>
</tr>
<tr>
<td>$26,862.00</td>
<td>*****$26,862.00</td>
<td>30.00</td>
<td>30.00</td>
<td>819000702</td>
<td>819000702</td>
</tr>
<tr>
<td>$21,780.00</td>
<td>*****$21,780.00</td>
<td>75.00</td>
<td>75.00</td>
<td>122850108</td>
<td>122850108</td>
</tr>
<tr>
<td>$16,100.00</td>
<td>*****$16,100.00</td>
<td>50.00</td>
<td>50.00</td>
<td>136500120</td>
<td>136500120</td>
</tr>
<tr>
<td>$27,062.00</td>
<td>*****$27,062.00</td>
<td>45.00</td>
<td>45.00</td>
<td>163800144</td>
<td>163800144</td>
</tr>
</tbody>
</table>

**Example:** Converting Alphanumeric and Numeric Date Fields to Alphanumeric Format

The following request against the EMPLOYEE data source converts the HIRE_DATE field to alphanumeric format. It also creates an alphanumeric date field named ADATE and converts it to its alphanumeric representation. The HIRE_DATE field has format I6YMD and the ADATE field has format A6YMD, so the alphanumeric representations have format A8 to account for the slashes between the date components. The /R option right-justifies the column titles over the field values:

DEFINE FILE EMPLOYEE
AHDATE/A8 = FPRINT(HIRE_DATE,'I6YMD', AHDATE);
ADATE/A6YMD = EDIT(HIRE_DATE);
AADATE/A8 = FPRINT(ADATE,'A6YMD', AADATE);
END
TABLE FILE EMPLOYEE
PRINT HIRE_DATE AHDATE/R
ADATE ADATE/R
ON TABLE SET PAGE NOPAGE
END

The output is:
Example: Converting a Date Field to Alphanumeric Format

The following request against the VIDEOTRK data source converts the TRANSDATE (YMD) field to alphanumeric format. The alphanumeric representation has format A8 to account for the slashes between the date components:

```
DEFINE FILE VIDEOTRK
ALPHA_DATE/A8 = FPRINT(TRANSDATE,'YMD', ALPHA_DATE);
END
TABLE FILE VIDEOTRK
PRINT TRANSDATE ALPHA_DATE
WHERE TRANSDATE LE '91/06/20'
ON TABLE SET PAGE NOPAGE
END
```

The output is:

```
TRANSDATE   ALPHA_DATE
---------    ----------
91/06/20     91/06/20
91/06/19     91/06/19
91/06/18     91/06/18
91/06/19     91/06/19
91/06/18     91/06/18
91/06/19     91/06/19
91/06/17     91/06/17
91/06/20     91/06/20
91/06/20     91/06/20
91/06/19     91/06/19
91/06/18     91/06/18
91/06/20     91/06/20
91/06/18     91/06/18
91/06/17     91/06/17
91/06/17     91/06/17
91/06/19     91/06/19
91/06/17     91/06/17
```
Example: Converting a Date-Time Field to Alphanumeric Format and Creating a HOLD File

The following request against the VIDEOTR2 data source converts the TRANSDATE (HYYMDI) field to alphanumeric format. The alphanumeric representation has format A16 to account for a four-digit year, two-digit month, two-digit day, two slashes between the date components, a space between the date and time, a two-digit hour, a colon between the hour and minute components, and a two-digit minute:

```
DEFINE FILE VIDEOTR2
DATE/I4 = HPART(TRANSDATE, 'YEAR', 'I4');
ALPHA_DATE/A16 = FPRINT(TRANSDATE,'HYYMDI', ALPHA_DATE);
END
TABLE FILE VIDEOTR2
PRINT TRANSDATE ALPHA_DATE/R
WHERE DATE EQ '1991'
ON TABLE SET PAGE NOPAGE
END
```

The output is:

<table>
<thead>
<tr>
<th>TRANSDATE</th>
<th>ALPHA_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>91/06/19</td>
<td>91/06/19</td>
</tr>
<tr>
<td>91/06/17</td>
<td>91/06/17</td>
</tr>
<tr>
<td>91/06/18</td>
<td>91/06/18</td>
</tr>
<tr>
<td>91/06/17</td>
<td>91/06/17</td>
</tr>
<tr>
<td>91/06/17</td>
<td>91/06/17</td>
</tr>
<tr>
<td>91/06/17</td>
<td>91/06/17</td>
</tr>
<tr>
<td>91/06/17</td>
<td>91/06/17</td>
</tr>
<tr>
<td>91/06/17</td>
<td>91/06/17</td>
</tr>
<tr>
<td>91/06/17</td>
<td>91/06/17</td>
</tr>
</tbody>
</table>

FPRINT: Converting Fields to Alphanumeric Format
If you hold the output in a comma-delimited or other alphanumeric output file, you can see that while the original field propagates only the numeric representation of the value, the converted field propagates the display options as well:

```plaintext
DEFINE FILE VIDEOTR2
DATE/I4 = HPART(TRANSDATE, 'YEAR', 'I4');
ALPHA_DATE/A16 = FPRINT(TRANSDATE,'HYYMDI', ALPHA_DATE);
END
TABLE FILE VIDEOTR2
PRINT TRANSDATE ALPHA_DATE/R
WHERE DATE EQ '1991'
ON TABLE HOLD FORMAT COMMA
END
```

The HOLD file follows. The first field represents the original data, and the second field contains the converted values with display options:

```
"19910627024500000","1991/06/27 02:45"
"19910620051500000","1991/06/20 05:15"
"19910621071100000","1991/06/21 07:11"
"19910621011000000","1991/06/21 01:10"
"19910619071800000","1991/06/19 07:18"
"19910619041100000","1991/06/19 04:11"
"19910625011900000","1991/06/25 01:19"
"19910624044300000","1991/06/24 04:43"
"19910624020800000","1991/06/24 02:08"
"19910625011700000","1991/06/25 01:17"
"19910627011700000","1991/06/27 01:17"
"19911117112800000","1991/11/17 11:28"
"19910624102700000","1991/06/24 10:27"
```

**FTOA: Converting a Number to Alphanumeric Format**

Available Languages: reporting, Maintain
The FTOA function converts a number up to 16 digits long from numeric format to alphanumeric format. It retains the decimal positions of the number and right-justifies it with leading spaces. You can also 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 is converted to A14. If the output format is not large enough, decimals are truncated.

**Syntax:**

How to Convert a Number to Alphanumeric Format

```plaintext
FTOA(number, '(format)', output)
```

where:

- **number**
  Numeric F or D (single and double precision floating-point)
  Is the number to be converted, or the name of the field that contains the number.

- **format**
  Alphanumeric
  Is the format of the number to be converted enclosed in parentheses. Only floating point single-precision and double-precision formats are supported. Include any edit options that you want to appear in the output. The D (floating-point double-precision) format automatically supplies commas.
  If you use a field name for this argument, specify the name without quotation marks or parentheses. If you specify a format, the format must be enclosed in single quotation marks and parentheses.

- **output**
  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 `number` and must account for edit options and a possible negative sign.

**Example:**

Converting From Numeric to Alphanumeric Format

FTOA converts GROSS from floating point double-precision to alphanumeric format.

```plaintext
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.
FTOA converts the GROSS field from floating point double-precision to alphanumeric format and stores the result in ALPHA_GROSS:

```plaintext
TABLE FILE EMPLOYEE
PRINT GROSS AND COMPUTE
ALPHA_GROSS/A15 = FTOA(GROSS, '(D12.2)', ALPHA_GROSS);
BY HIGHEST 1 PAY_DATE NOPRINT
BY LAST_NAME
WHERE (GROSS GT 800) AND (GROSS LT 2300);
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>GROSS</th>
<th>ALPHA_GROSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACKWOOD</td>
<td>$1,815.00</td>
<td>1,815.00</td>
</tr>
<tr>
<td>CROSS</td>
<td>$2,255.00</td>
<td>2,255.00</td>
</tr>
<tr>
<td>IRVING</td>
<td>$2,238.50</td>
<td>2,238.50</td>
</tr>
<tr>
<td>JONES</td>
<td>$1,540.00</td>
<td>1,540.00</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>$1,342.00</td>
<td>1,342.00</td>
</tr>
<tr>
<td>ROMANS</td>
<td>$1,760.00</td>
<td>1,760.00</td>
</tr>
<tr>
<td>SMITH</td>
<td>$1,100.00</td>
<td>1,100.00</td>
</tr>
<tr>
<td>STEVENS</td>
<td>$916.67</td>
<td>916.67</td>
</tr>
</tbody>
</table>

**HEXBYT: Converting a Decimal Integer to a Character**

Available Languages: reporting, Maintain

The HEXBYT function obtains the ASCII, EBCDIC, or Unicode character equivalent of a decimal integer, depending on your configuration and operating environment. The decimal value you specify must be the value associated with the character on the configured code page. HEXBYT returns a single alphanumeric character in the ASCII, EBCDIC, or Unicode character set. You can use this function to produce characters that are not on your keyboard, similar to the CTRAN function.

In Unicode configurations, this function uses values in the range:

- 0 to 255 for 1-byte characters.
- 256 to 65535 for 2-byte characters.
- 65536 to 16777215 for 3-byte characters.
- 16777216 to 4294967295 for 4-byte characters (primarily for EBCDIC).

The display of special characters depends on your software and hardware; not all special characters may appear. For printable ASCII and EBCDIC characters and their integer equivalents see the *Character Chart for ASCII and EBCDIC*. 
**Syntax:**

How to Convert a Decimal Integer to a Character

HEXBYT(decimal_value, output)

where:

*decimal_value*

Integer

Is the decimal integer to be converted to a single character. In non-Unicode environments, a value greater than 255 is treated as the remainder of `decimal_value` divided by 256. The decimal value you specify must be the value associated with the character on the configured code page.

*output*

Alphanumeric

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

**Example:**

Converting a Decimal Integer to a Character in ASCII and Unicode

The following request uses HEXBYT to convert the decimal integer value 130 to the comma character on ASCII code page 1252. The comma is then concatenated between LAST_NAME and FIRST_NAME to create the NAME field:

```plaintext
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND
COMPUTE COMMA1/A1 = HEXBYT(130, COMMA1); NOPRINT
COMPUTE NAME/A40 = LAST_NAME || COMMA1 | ' ' | FIRST_NAME;
BY LAST_NAME NOPRINT
BY FIRST_NAME
WHERE DEPARTMENT EQ 'MIS';
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```
The output is shown in the following image.

```
<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROSEMARIE</td>
<td>BLACKWOOD</td>
<td>BLACKWOOD, ROSEMARIE</td>
</tr>
<tr>
<td>BARBARA</td>
<td>CROSS</td>
<td>CROSS, BARBARA</td>
</tr>
<tr>
<td>MARY</td>
<td>GREENSPAN</td>
<td>GREENSPAN, MARY</td>
</tr>
<tr>
<td>DIANE</td>
<td>JONES</td>
<td>JONES, DIANE</td>
</tr>
<tr>
<td>JOHN</td>
<td>MCCOY</td>
<td>MCCOY, JOHN</td>
</tr>
<tr>
<td>MARY</td>
<td>SMITH</td>
<td>SMITH, MARY</td>
</tr>
</tbody>
</table>
```

To produce the same output in a Unicode environment configured for code page 65001, replace the COMPUTE command for the field COMMA1 with the following syntax, in which the call to HEXBYT converts the integer value 14844058 to the comma character:

```
COMPUTE COMMA1/A1 = HEXBYT(14844058, COMMA1); NOPRINT
```

**Example:** Converting a Decimal Integer to a Character

HEXBYT converts LAST_INIT_CODE to its character equivalent and stores the result 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.

HEXBYT converts LAST_INIT_CODE to its character equivalent and stores the result in LAST_INIT:

```
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND
COMPUTE LAST_INIT_CODE/I3 = BYTVAL(LAST_NAME, 'I3');
COMPUTE LAST_INIT/A1 = HEXBYT(LAST_INIT_CODE, LAST_INIT);
WHERE DEPARTMENT EQ 'MIS';
END
```

The output for an ASCII platform is:

```
<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>LAST_INIT_CODE</th>
<th>LAST_INIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>83 S</td>
<td>Smith</td>
</tr>
<tr>
<td>JONES</td>
<td>74 J</td>
<td>Jones</td>
</tr>
<tr>
<td>MCCOY</td>
<td>77 M</td>
<td>Mccoy</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>66 B</td>
<td>Blackwood</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>71 G</td>
<td>Greenspan</td>
</tr>
<tr>
<td>CROSS</td>
<td>67 C</td>
<td>Cross</td>
</tr>
</tbody>
</table>
```
The output for an EBCDIC platform is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>LAST_INIT_CODE</th>
<th>LAST_INIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>226</td>
<td>S</td>
</tr>
<tr>
<td>JONES</td>
<td>209</td>
<td>J</td>
</tr>
<tr>
<td>MCCOY</td>
<td>212</td>
<td>M</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>194</td>
<td>B</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>199</td>
<td>G</td>
</tr>
<tr>
<td>CROSS</td>
<td>195</td>
<td>C</td>
</tr>
</tbody>
</table>

**Example: Inserting Braces for Mainframe**

HEXBYT converts the decimal integer 192 to its EBCDIC character equivalent, which is a left brace; and the decimal integer 208 to its character equivalent, which is a right brace. If the value of CURR_SAL is less than 12000, the value of LAST_NAME is enclosed in braces.

```
DEFINE FILE EMPLOYEE
BRACE/A17 = HEXBYT(192, 'A1') | LAST_NAME | HEXBYT(208, 'A1');
BNAME/A17 = IF CURR_SAL LT 12000 THEN BRACE ELSE LAST_NAME;
END
TABLE FILE EMPLOYEE
PRINT BNAME CURR_SAL BY EMP_ID END
```

The output is:

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>BNAME</th>
<th>CURR_SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>071382660</td>
<td>{STEVENS}</td>
<td>$11,000.00</td>
</tr>
<tr>
<td>112847612</td>
<td>SMITH</td>
<td>$13,200.00</td>
</tr>
<tr>
<td>117593129</td>
<td>JONES</td>
<td>$18,480.00</td>
</tr>
<tr>
<td>119265415</td>
<td>{SMITH}</td>
<td>$9,500.00</td>
</tr>
<tr>
<td>119329144</td>
<td>BANNING</td>
<td>$29,700.00</td>
</tr>
<tr>
<td>123764317</td>
<td>IRVING</td>
<td>$26,862.00</td>
</tr>
<tr>
<td>126724188</td>
<td>ROMANS</td>
<td>$21,120.00</td>
</tr>
<tr>
<td>219984371</td>
<td>MCCOY</td>
<td>$18,480.00</td>
</tr>
<tr>
<td>326179357</td>
<td>BLACKWOOD</td>
<td>$21,780.00</td>
</tr>
<tr>
<td>451123478</td>
<td>MCKNIGHT</td>
<td>$16,100.00</td>
</tr>
<tr>
<td>543729165</td>
<td>{GREENSPAN}</td>
<td>$9,000.00</td>
</tr>
<tr>
<td>818692173</td>
<td>CROSS</td>
<td>$27,062.00</td>
</tr>
</tbody>
</table>

**ITONUM: Converting a Large Binary Integer Number to Double-Precision Format**

Available Languages: reporting, Maintain

The ITONUM function converts a large binary integer 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.
Some programming languages and some non-FOCUS data storage systems use large binary integer formats. However, large binary integers (more than 4 bytes in length) are not supported in the Master File so they 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 Binary Integer to Double-Precision Format**

\[ \text{ITONUM}(\text{maxbytes}, \text{infield}, \text{output}) \]

where:

- **maxbytes**
  - Numeric
  - Is the maximum number of bytes in the 8-byte binary input field that have significant numeric data, including the binary sign. Valid values are:
    - 5 ignores the left-most 3 bytes.
    - 6 ignores the left-most 2 bytes.
    - 7 ignores the left-most byte.

- **infield**
  - A8
  - Is the field that contains the binary number. Both the USAGE and ACTUAL formats of the field must be A8.

- **output**
  - Double precision floating-point (Dn)
  - 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 Dn.

**Example:**

**Converting a Large Binary Integer to Double-Precision Format**

ITONUM converts BINARYFLD to double-precision format.

\[ \text{ITONUM}(6, \text{BINARYFLD}, \text{'}D14'\text{)} \]

Suppose a binary number in an external file has the following COBOL format:

\[ \text{PIC 9(8)V9(4) COMP} \]
It is defined in the EUROCAR Master File as a field named BINARYFLD. Its field formats are USAGE=A8 and ACTUAL=A8, since its length is greater than 4 bytes.

The following request converts the field to double-precision format:

```
DEFINE FILE EUROCAR
MYFLD/D14 = ITONUM(6, BINARYFLD, MYFLD);
END
TABLE FILE EUROCAR
PRINT MYFLD BY CAR
END
```

**ITOPACK: Converting a Large Binary Integer to Packed-Decimal Format**

Available Languages: reporting, Maintain

The ITOPACK function converts a large binary integer in a non-FOCUS data source to packed-decimal 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.

Some programming languages and some non-FOCUS data storage systems use double-word binary integer formats. These are similar to the single-word binary integers used by FOCUS, but they allow larger numbers. However, large binary integers (more than 4 bytes in length) are not supported in the Master File so they 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

\[ \text{ITOPACK}(\text{maxbytes}, \text{infield}, \text{output}) \]

where:

\[ \text{maxbytes} \]

Numeric
Is the maximum number of bytes in the 8-byte binary input field that have significant numeric data, including the binary sign.

Valid values are:

- **5** ignores the left-most 3 bytes (up to 11 significant positions).
- **6** ignores the left-most 2 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**

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 Pn or Pn.d.

**Example:** Converting a Large Binary Integer to Packed-Decimal Format

ITOPACK converts BINARYFLD to packed-decimal format.

```plaintext
ITOPACK(6, BINARYFLD, 'P14.4')
```

Suppose a binary number in an external file has the following COBOL format:

```plaintext
PIC 9(8)V9(4) COMP
```

It is defined in the EUROCAR Master File as a field named BINARYFLD. Its field formats are USAGE=A8 and ACTUAL=A8, since its length is greater than 4 bytes.

The following request converts the field to packed-decimal format:

```plaintext
DEFINE FILE EUROCAR
PACKFLD/P14.4 = ITOPACK(6, BINARYFLD, PACKFLD);
END
TABLE FILE EUROCAR
PRINT PACKFLD BY CAR
END
```
ITOZ: Converting a Number to Zoned Format

Available Languages: reporting, Maintain

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

\[ \text{ITOZ} \left( \text{length}, \text{in\_value}, \text{output} \right) \]

where:

\text{length}

Integer

Is the length of \text{in\_value} in bytes. The maximum number of bytes is 15. The last byte includes the sign.

\text{in\_value}

Numeric

Is the number to be converted, or the field that contains the number. The number is truncated to an integer before it is converted.

\text{output}

Alphanumeric

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

Example: Converting a Number to Zoned Format

ITOZ converts CURR\_SAL to zoned format.

\[ \text{ITOZ}(8, \text{CURR\_SAL}, \text{'A8'}) \]
The following request creates an extract file containing employee IDs and salaries in zoned format for a COBOL program:

```
DEFINE FILE EMPLOYEE
ZONE_SAL/A8 = ITOZ(8, CURR_SAL, ZONE_SAL);
END

TABLE FILE EMPLOYEE
PRINT CURR_SAL ZONE_SAL BY EMP_ID
ON TABLE SAVE AS SALARIES
END
```

The resulting extract file is:

NUMBER OF RECORDS IN TABLE= 12 LINES= 12

<table>
<thead>
<tr>
<th>ALPHANUMERIC RECORD NAMED SALARIES</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIELDNAME</td>
</tr>
<tr>
<td>EMP_ID</td>
</tr>
<tr>
<td>CURR_SAL</td>
</tr>
<tr>
<td>ZONE_SAL</td>
</tr>
<tr>
<td>TOTAL</td>
</tr>
</tbody>
</table>

NUMBER OF RECORDS IN TABLE= 12 LINES= 12

| [EBCDIC|ALPHANUMERIC] RECORD NAMED SALARIES |
|-----------------------------------------|
| FIELDNAME | ALIAS | FORMAT | LENGTH |
| EMP_ID | EID | A9 | 9 |
| CURR_SAL | CSAL | D12.2M | 12 |
| ZONE_SAL | | A8 | 8 |
| TOTAL | | | 29 |

DCB USED WITH FILE SALARIES IS DCB=(RECFM=FB,LRECL=00029,BLKSIZE=00580)

If you remove the SAVE command and run the request, the output for an EBCDIC platform follows. The left brace in EBCDIC is hexadecimal C0; this indicates a positive sign and a final digit of 0. The capital B in EBCDIC is hexadecimal C2; this indicates a positive sign and a final digit of 2.

```
EMP_ID             URR_SAL  ZONE_SAL
------            --------  --------
071382660         $11,000.00  0001100{   
112847612         $13,200.00  0001320{   
117593129         $18,480.00  0001848{   
119265415          $9,500.00  0000950{   
119329144         $29,700.00  0002970{   
123764317          $26,862.00  0002686B   
126724188          $21,120.00  0002112{   
219984371          $18,480.00  0001848{   
326179357          $21,780.00  0002178{   
451123478          $16,100.00  0001610{   
543729165          $9,000.00  0000900{   
818692173         $27,062.00  0002706B   
```
PCKOUT: Writing a Packed Number of Variable Length

Available Languages: reporting, Maintain

The PCKOUT function writes a packed-decimal number of variable length to an extract file. When a request saves a packed 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 between 1 to 16 bytes.

Syntax: How to Write a Packed Number of Variable Length

PCKOUT(in_value, length, output)

where:

in_value
Numeric

Is the input value field that contains the values. It can be in packed, integer, single- or double-precision floating point format. If it is not in integer format, it is rounded to the nearest whole number.

length
Numeric

Is the length of the output value, from 1 to 16 bytes.

output
Alphanumeric

Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. The 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')
PCKOUT converts the CURR_SAL field to a 5-byte packed field and stores the result in SHORT_SAL:

```
DEFINE FILE EMPLOYEE
SHORT_SAL/A5 = PCKOUT(CURR_SAL, 5, SHORT_SAL);
END
TABLE FILE EMPLOYEE
PRINT LAST_NAME SHORT_SAL HIRE_DATE
ON TABLE SAVE
END
```

The resulting extract file is:

```
NUMBER OF RECORDS IN TABLE= 12 LINES= 12

ALPHANUMERIC RECORD NAMED SAVE
FIELDNAME ALIAS FORMAT LENGTH
LAST_NAME LN A15 15
SHORT_SAL A5 5
HIRE_DATE HDT I6YMD 6
TOTAL 26
```

PTOA: Converting a Packed-Decimal Number to Alphanumeric Format

Available Languages: reporting, Maintain

The PTOA function converts a packed-decimal number from numeric format to alphanumeric format. It retains the decimal positions of the number and right-justifies it with leading spaces. You can also add edit options to a number converted by PTOA.

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 P12.2C format is converted to A14. If the output format is not large enough, the right-most characters are truncated.
How to Convert a Packed-Decimal Number to Alphanumeric Format

PTOA(number, '(format)', output)

where:

number

Numeric P (packed-decimal) or F or D (single and double precision floating-point)

Is the number to be converted, or the name of the field that contains the number.

format

Alphanumeric

Is the format of the number enclosed in both single quotation marks and parentheses.

Only packed-decimal format is supported. Include any edit options that you want to display in the output.

The format value does not require the same length or number of decimal places as the original field. If you change the number of decimal places, the result is rounded. If you make the length too short to hold the integer portion of the number, asterisks appear instead of the number.

If you use a field name for this argument, specify the name without quotation marks or parentheses. However, parentheses must be included around the format stored in this field. For example:

FMT/A10 = '(P12.2C)';

You can then use this field as the format argument when using the function in your request:

COMPUTE ALPHA_GROSS/A20 = PTOA(PGROSS, FMT, ALPHA_GROSS);

output

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 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')
PTOA is called twice to convert the PGROSS field from packed-decimal to alphanumeric format. The format specified in the first call to the function is stored in a virtual field named FMT. The format specified in the second call to the function does not include decimal places, so the value is rounded when it appears:

```
DEFINE FILE EMPLOYEE
PGROSS/P18.2=GROSS;
FMT/A10='(P14.2C)';
END
```

```
TABLE FILE EMPLOYEE PRINT PGROSS NOPRINT
COMPUTE AGROSS/A17 = PTOA(PGROSS, FMT, AGROSS); AS ''
COMPUTE BGROSS/A37 = '<- THIS AMOUNT IS' |
    PTOA(PGROSS, '(P5C)', 'A6') |
    ' WHEN ROUNDED'; AS '' IN +1
```

```
BY HIGHEST 1 PAY_DATE NOPRINT
BY LAST_NAME NOPRINT
END
```

The output is:

```
2,475.00 <- THIS AMOUNT IS 2,475 WHEN ROUNDED
1,815.00 <- THIS AMOUNT IS 1,815 WHEN ROUNDED
2,255.00 <- THIS AMOUNT IS 2,255 WHEN ROUNDED
  750.00 <- THIS AMOUNT IS 750 WHEN ROUNDED
2,238.50 <- THIS AMOUNT IS 2,239 WHEN ROUNDED
1,540.00 <- THIS AMOUNT IS 1,540 WHEN ROUNDED
1,540.00 <- THIS AMOUNT IS 1,540 WHEN ROUNDED
1,342.00 <- THIS AMOUNT IS 1,342 WHEN ROUNDED
1,760.00 <- THIS AMOUNT IS 1,760 WHEN ROUNDED
1,100.00 <- THIS AMOUNT IS 1,100 WHEN ROUNDED
  791.67 <- THIS AMOUNT IS 792 WHEN ROUNDED
  916.67 <- THIS AMOUNT IS 917 WHEN ROUNDED
```

**TSTOPACK:** Converting an MSSQL or Sybase Timestamp Column to Packed Decimal

This function applies to the Microsoft SQL Server and Sybase adapters only.

Microsoft SQL Server and Sybase have a data type called TIMESTAMP. Rather than containing an actual timestamp, columns with this data type contain a number that is incremented for each record inserted or updated in the data source. This timestamp comes from a common area, so no two tables in the database have the same timestamp column value. The value is stored in Binary(8) or Varbinary(8) format in the table, but is returned as a double wide alphanumeric column (A16). You can use the TSTOPACK function to convert the timestamp value to packed decimal.

**Syntax:**

```
TSTOPACK(tscol, output);
```

where:

tscol

A16

Is the timestamp column to be converted.

output

P21

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

Example:  Converting a Microsoft SQL Server Timestamp Column to Packed Decimal

The following CREATE TABLE command creates a SQL Server table name TSTEST that contains an integer counter column named I and a timestamp column named TS:

```
CREATE TABLE TSTEST (I INT, TS timestamp) ;
```

The Master File for the TSTEST data source follows. The field TS represents the TIMESTAMP column:

```
FILENAME=TSTEST, SUFFIX=SQLMSS , $
SEGMENT=TSTEST, SEGTYPE=S0, $
   FIELDNAME=I, ALIAS=I, USAGE=I11, ACTUAL=I4, $
   MISSING=ON, $
   FIELDNAME=TS, ALIAS=TS, USAGE=A16, ACTUAL=A16, FIELDTYPE=R, $
```

Note: When you generate a synonym for a table with a TIMESTAMP column, the TIMESTAMP column is created as read-only (FIELDTYPE=R).

TSTOPACK converts the timestamp column TS to packed decimal:

```
DEFINE FILE TSTEST
TSNUM/P21=TSTOPACK(TS,'P21');
END
TABLE FILE TEST64
PRINT I TS TSNUM
END
```

```
TSNUM/P21=TSTOPACK(TS,'P21');
```
The output is:

<table>
<thead>
<tr>
<th>Source String</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000007815</td>
<td>30741</td>
</tr>
<tr>
<td>0000000000007816</td>
<td>30742</td>
</tr>
</tbody>
</table>

**UFMT: Converting an Alphanumeric String to Hexadecimal**

**Available Languages:** reporting, Maintain

The UFMT function converts characters in an alphanumeric 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:**

`UFMT(source_string, length, output)`

where:

- **source_string**
  
  Alphanumeric

  Is the alphanumeric string to convert enclosed in single quotation marks (''), or the field that contains the string.
**length**

Integer

Is the number of characters in *source_string*.

**output**

Alphanumeric

Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks (''). The format of *output* must be alphanumeric and its length must be twice that of *length*.

**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.

\[
\text{UFMT (JOBCODE, 3, 'A6')}\]

For A01, the result is C1F0F1.

For A02, the result is C1F0F2.

UFMT converts each value in JOBCODE to its hexadecimal representation and stores the result in HEXCODE:

\[
\text{DEFINE FILE JOBFILE} \\
\text{HEXCODE/A6 = UFMT (JOBCODE, 3, HEXCODE);} \\
\text{END} \\
\text{TABLE FILE JOBFILE} \\
\text{PRINT JOBCODE HEXCODE} \\
\text{END} \\
\]

The output is:

<table>
<thead>
<tr>
<th>JOBCODE</th>
<th>HEXCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A01</td>
<td>C1F0F1</td>
</tr>
<tr>
<td>A02</td>
<td>C1F0F2</td>
</tr>
<tr>
<td>A07</td>
<td>C1F0F7</td>
</tr>
<tr>
<td>A12</td>
<td>C1F1F2</td>
</tr>
<tr>
<td>A14</td>
<td>C1F1F4</td>
</tr>
<tr>
<td>A15</td>
<td>C1F1F5</td>
</tr>
<tr>
<td>A16</td>
<td>C1F1F6</td>
</tr>
<tr>
<td>A17</td>
<td>C1F1F7</td>
</tr>
<tr>
<td>B01</td>
<td>C2F0F1</td>
</tr>
<tr>
<td>B02</td>
<td>C2F0F2</td>
</tr>
<tr>
<td>B03</td>
<td>C2F0F3</td>
</tr>
<tr>
<td>B04</td>
<td>C2F0F4</td>
</tr>
<tr>
<td>B14</td>
<td>C2F1F4</td>
</tr>
</tbody>
</table>
**XTPACK: Writing a Packed Number With Up to 31 Significant Digits to an Output File**

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(in_value, outlength, outdec, output)
```

where:

- **infield**
  Numeric
  Is 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 **output**.

- **output**
  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

XTPACK converts LONGPCK to alphanumeric so that it can be saved in an output file:

```
XTPACK(LONGPCK,13,2,'A13');
```
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:

<table>
<thead>
<tr>
<th>FIELDNAME</th>
<th>ALIAS</th>
<th>FORMAT</th>
<th>LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCT_INC</td>
<td>PI</td>
<td>F6.2</td>
<td>6</td>
</tr>
<tr>
<td>LONGPCK</td>
<td></td>
<td>P25.2</td>
<td>25</td>
</tr>
<tr>
<td>ALPHAPCK</td>
<td>A13</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td></td>
<td>44</td>
</tr>
</tbody>
</table>

SAVED...
Chapter 13

Simplified Numeric Functions

Numeric functions have been developed that make it easier to understand and enter the required arguments. These functions have streamlined parameter lists, similar to those used by SQL functions. In some cases, these simplified functions provide slightly different functionality than previous versions of similar functions.

The simplified functions do not have an output argument. Each function returns a value that has a specific data type.

When used in a request against a relational data source, these functions are optimized (passed to the RDBMS for processing).

Note:

- The simplified numeric functions are supported in Dialogue Manager.
- The simplified numeric functions are not supported in Maintain Data.

In this chapter:

- CEILING: Returning the Smallest Integer Value Greater Than or Equal to a Value
- EXPONENT: Raising e to a Power
- FLOOR: Returning the Largest Integer Less Than or Equal to a Value
- MOD: Calculating the Remainder From a Division
- POWER: Raising a Value to a Power

CEILING: Returning the Smallest Integer Value Greater Than or Equal to a Value

CEILING returns the smallest integer value that is greater than or equal to a number.

Syntax: How to Return the Smallest Integer Greater Than or Equal to a Number

CEILING(number)
where:

\textit{number} \\
\text{Numeric}

Is the number whose ceiling will be returned. The output data type is the same as the input data type.

\textbf{Example: Returning the Ceiling of a Number}

In the following request, \texttt{CEILING} returns the smallest integer greater than or equal to the \texttt{GROSS\_PROFIT\_US} value.

\begin{verbatim}
DEFINE FILE WF_RETAIL_LITE
CEIL1/D7.2= CEILING(GROSS_PROFIT_US);
END
TABLE FILE WF_RETAIL_LITE
PRINT GROSS_PROFIT_US/D9.2  CEIL1
ON TABLE SET PAGE NOPAGE
END
\end{verbatim}

\texttt{CEILING} returns the smallest integer larger than the value in \texttt{GROSS\_PROFIT\_US}:

\begin{verbatim}
CEILING(GROSS_PROFIT_US)
\end{verbatim}
The partial output follows. Note that even though the value returned is an integer, it is returned with the same format as the CEIL1 field (D7.2):

<table>
<thead>
<tr>
<th>Gross Profit</th>
<th>CEIL1</th>
</tr>
</thead>
<tbody>
<tr>
<td>165.00</td>
<td>165.00</td>
</tr>
<tr>
<td>13.99</td>
<td>14.00</td>
</tr>
<tr>
<td>60.99</td>
<td>61.00</td>
</tr>
<tr>
<td>225.98</td>
<td>226.00</td>
</tr>
<tr>
<td>79.99</td>
<td>80.00</td>
</tr>
<tr>
<td>44.59</td>
<td>45.00</td>
</tr>
<tr>
<td>94.30</td>
<td>95.00</td>
</tr>
<tr>
<td>238.50</td>
<td>239.00</td>
</tr>
<tr>
<td>199.99</td>
<td>200.00</td>
</tr>
<tr>
<td>68.99</td>
<td>69.00</td>
</tr>
<tr>
<td>63.58</td>
<td>64.00</td>
</tr>
<tr>
<td>129.99</td>
<td>130.00</td>
</tr>
<tr>
<td>37.49</td>
<td>38.00</td>
</tr>
<tr>
<td>75.99</td>
<td>76.00</td>
</tr>
<tr>
<td>13.99</td>
<td>14.00</td>
</tr>
<tr>
<td>119.00</td>
<td>119.00</td>
</tr>
<tr>
<td>-30.01</td>
<td>-30.00</td>
</tr>
<tr>
<td>54.99</td>
<td>55.00</td>
</tr>
<tr>
<td>189.98</td>
<td>190.00</td>
</tr>
<tr>
<td>44.59</td>
<td>45.00</td>
</tr>
<tr>
<td>91.98</td>
<td>92.00</td>
</tr>
<tr>
<td>89.00</td>
<td>89.00</td>
</tr>
<tr>
<td>59.50</td>
<td>60.00</td>
</tr>
<tr>
<td>129.99</td>
<td>130.00</td>
</tr>
<tr>
<td>54.00</td>
<td>54.00</td>
</tr>
<tr>
<td>109.98</td>
<td>110.00</td>
</tr>
<tr>
<td>98.99</td>
<td>99.00</td>
</tr>
<tr>
<td>98.99</td>
<td>99.00</td>
</tr>
<tr>
<td>99.99</td>
<td>100.00</td>
</tr>
<tr>
<td>44.59</td>
<td>45.00</td>
</tr>
</tbody>
</table>

For 225.98, the output is 226.00.

For -30.01, the output is -30.00.

**EXPO**NENT: Raising e to a Power

EXPO**NENT** raises the constant e to a power.

**Syntax:** How to Raise the Constant e to a Power

**EXPO**NENT(*power*)
where:

**power**

Numeric

Is the power to which to raise e. The output data type is numeric.

**Example:**  **Raising e to a Power**

The following request prints the value of e and the value of e raised to the fifth power.

```
DEFINE FILE WF_RETAIL_LITE
EXP1/D12.5 = EXPONENT(1);
EXP2/D12.5 = EXPONENT(5);
END
TABLE FILE WF_RETAIL_LITE
PRINT EXP1 EXP2
BY BUSINESS_REGION AS Region
WHERE BUSINESS_REGION EQ 'EMEA'
WHERE RECORDLIMIT EQ 1
ON TABLE SET PAGE NOPAGE
END
```

The output is shown in the following image.

<table>
<thead>
<tr>
<th>Region</th>
<th>EXP1</th>
<th>EXP2</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMEA</td>
<td>2.71828</td>
<td>148.41316</td>
</tr>
</tbody>
</table>

For EXPONENT(1), the value is 2.71828

For EXPONENT(5), the value is 148.41316

**FLOOR: Returning the Largest Integer Less Than or Equal to a Value**

FLOOR returns the largest integer value that is less than or equal to a number.

**Syntax:**  **How to Return the Largest Integer Less Than or Equal to a Number**

```
FLOOR(number)
```

where:

**number**

Numeric

Is the number whose floor will be returned. The output data type is the same as the input data type.
Example: Returning the Floor of a Number

In the following request, FLOOR returns the largest integer less than or equal to the GROSS_PROFIT_US value.

```
DEFINE FILE WF_RETAIL_LITE
FLOOR1/D7.2= FLOOR(GROSS_PROFIT_US);
END
TABLE FILE WF_RETAIL_LITE
PRINT GROSS_PROFIT_US/D9.2  FLOOR1
ON TABLE SET PAGE NOPAGE
END
```

FLOOR returns the smallest integer larger than the value in GROSS_PROFIT_US:

```
FLOOR(GROSS_PROFIT_US)
```

Partial output follows. Note that even though the value returned is an integer, it is returned with the same format as the FLOOR1 field (D7.2):

<table>
<thead>
<tr>
<th>Gross Profit</th>
<th>FLOOR1</th>
</tr>
</thead>
<tbody>
<tr>
<td>165.00</td>
<td>165.00</td>
</tr>
<tr>
<td>13.99</td>
<td>13.00</td>
</tr>
<tr>
<td>60.99</td>
<td>60.00</td>
</tr>
<tr>
<td>225.98</td>
<td>225.00</td>
</tr>
<tr>
<td>79.99</td>
<td>79.00</td>
</tr>
<tr>
<td>44.59</td>
<td>44.00</td>
</tr>
<tr>
<td>94.30</td>
<td>94.00</td>
</tr>
<tr>
<td>238.50</td>
<td>238.00</td>
</tr>
<tr>
<td>199.99</td>
<td>199.00</td>
</tr>
<tr>
<td>68.99</td>
<td>68.00</td>
</tr>
<tr>
<td>63.58</td>
<td>63.00</td>
</tr>
<tr>
<td>129.99</td>
<td>129.00</td>
</tr>
<tr>
<td>37.49</td>
<td>37.00</td>
</tr>
<tr>
<td>75.99</td>
<td>75.00</td>
</tr>
<tr>
<td>13.99</td>
<td>13.00</td>
</tr>
<tr>
<td>119.00</td>
<td>119.00</td>
</tr>
<tr>
<td>-30.01</td>
<td>-31.00</td>
</tr>
<tr>
<td>54.99</td>
<td>54.00</td>
</tr>
<tr>
<td>189.98</td>
<td>189.00</td>
</tr>
<tr>
<td>44.59</td>
<td>44.00</td>
</tr>
<tr>
<td>91.98</td>
<td>91.00</td>
</tr>
<tr>
<td>89.00</td>
<td>89.00</td>
</tr>
<tr>
<td>59.50</td>
<td>59.00</td>
</tr>
<tr>
<td>129.99</td>
<td>129.00</td>
</tr>
<tr>
<td>54.00</td>
<td>54.00</td>
</tr>
<tr>
<td>109.98</td>
<td>109.00</td>
</tr>
<tr>
<td>98.99</td>
<td>98.00</td>
</tr>
<tr>
<td>98.99</td>
<td>98.00</td>
</tr>
<tr>
<td>99.99</td>
<td>99.00</td>
</tr>
<tr>
<td>44.59</td>
<td>44.00</td>
</tr>
</tbody>
</table>

For 225.98, the output is 225.00.
For -30.01, the output is -31.00.

**MOD: Calculating the Remainder From a Division**

MOD calculates the remainder from a division. The output data type is the same as the input data type.

**Syntax:**

**How to Calculate the Remainder From a Division**

\[
\text{MOD}(\text{dividend}, \text{divisor})
\]

where:

- **dividend**
  - Numeric
  - Is the value to divide.
  - **Note:** The sign of the returned value will be the same as the sign of the dividend.

- **divisor**
  - Numeric
  - Is the value to divide by.

If the divisor is zero (0), the dividend is returned.

**Example:**

**Calculating the Remainder From a Division**

In the following request, MOD returns the remainder of PRICE_DOLLARS divided by DAYSDELAYED:

```plaintext
DEFINE FILE WF_RETAIL_LITE
MOD1/D7.2= MOD(PRICE_DOLLARS, DAYSDELAYED);
END
TABLE FILE WF_RETAIL_LITE
PRINT PRICE_DOLLARS/D7.2 DAYSDELAYED/I5 MOD1
WHERE DAYSDELAYED GT 1
END
```

```plaintext
ON TABLE SET PAGE NOPAGE
ON TABLE PCHOLD FORMAT WP
END
```
Partial output follows:

<table>
<thead>
<tr>
<th>Price Dollars</th>
<th>Days Delayed</th>
<th>MOD1</th>
</tr>
</thead>
<tbody>
<tr>
<td>399.00</td>
<td>3</td>
<td>.00</td>
</tr>
<tr>
<td>489.99</td>
<td>3</td>
<td>.99</td>
</tr>
<tr>
<td>786.50</td>
<td>2</td>
<td>.50</td>
</tr>
<tr>
<td>599.99</td>
<td>4</td>
<td>3.99</td>
</tr>
<tr>
<td>29.99</td>
<td>4</td>
<td>1.99</td>
</tr>
<tr>
<td>169.00</td>
<td>2</td>
<td>1.00</td>
</tr>
<tr>
<td>219.99</td>
<td>2</td>
<td>1.99</td>
</tr>
<tr>
<td>280.00</td>
<td>3</td>
<td>1.00</td>
</tr>
<tr>
<td>79.99</td>
<td>4</td>
<td>3.99</td>
</tr>
<tr>
<td>145.99</td>
<td>2</td>
<td>1.99</td>
</tr>
<tr>
<td>399.99</td>
<td>3</td>
<td>.99</td>
</tr>
<tr>
<td>349.99</td>
<td>3</td>
<td>1.99</td>
</tr>
<tr>
<td>169.00</td>
<td>3</td>
<td>1.00</td>
</tr>
</tbody>
</table>

MOD returns the remainder of PRICE_DOLLARS divided by DAYSDELAYED

MOD(PRICE_DOLLARS, DAYSDELAYED)

For 399.00/3, the value is zero (0).

for 489.00/3, the value is .99.

**POWER: Raising a Value to a Power**

POWER raises a base value to a power.

**Syntax:** How to Raise a Value to a Power

\[
\text{POWER(} \text{base, power} \text{)}
\]

where:

- **base**
  
  Numeric
  
  Is the value to raise to a power. The output value has the same data type as the base value. If the base value is integer, negative power values will result in truncation.

- **power**
  
  Numeric
  
  Is the power to which to raise the base value.
**Example:**  
**Raising a Base Value to a Power**

In the following request, POWER returns the value `COGS_US/20.00` raised to the power stored in `DAYSDELAYED`:

```
DEFINE FILE WF_RETAIL_LITE
BASE=COGS_US/20.00;
POWER1= POWER(COGS_US/20.00,DAYSDELAYED);
END
TABLE FILE WF_RETAIL_LITE
PRINT BASE IN 15 DAYSDELAYED POWER1
BY PRODUCT_CATEGORY
WHERE PRODUCT_CATEGORY EQ 'Computers'
WHERE DAYSDELAYED NE 0
ON TABLE SET PAGE NOPAGE
END
```

Partial output follows:

<table>
<thead>
<tr>
<th>Product Category</th>
<th>Days Delayed</th>
<th>BASE</th>
<th>POWER1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computers</td>
<td>3</td>
<td>12.15</td>
<td>1,793.61</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>16.70</td>
<td>278.89</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>8.35</td>
<td>8.35</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8.10</td>
<td>65.61</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4.05</td>
<td>4.05</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4.05</td>
<td>16.40</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4.05</td>
<td>269.04</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>8.35</td>
<td>8.35</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>16.70</td>
<td>16.70</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>8.35</td>
<td>582.18</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4.05</td>
<td>4.05</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4.05</td>
<td>4.05</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8.35</td>
<td>4,861.23</td>
</tr>
<tr>
<td></td>
<td>-1</td>
<td>8.35</td>
<td>0.12</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>8.35</td>
<td>8.35</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>8.35</td>
<td>582.18</td>
</tr>
</tbody>
</table>

Power returns the value `COGS_US/20.00` raised to the power stored in `DAYSDELAYED`.

```
POWER1= POWER(COGS_US/20.00,DAYSDELAYED)
```

For base 12.15 and power 3, the value is 1,793.61.
Numeric Functions

Numeric functions perform calculations on numeric constants and fields.

For many functions, the output argument can be supplied either as a field name or as a format enclosed in single quotation marks. However, if a function is called from a Dialogue Manager command, this argument must always be supplied as a format, and if a function is called from a Maintain Data procedure, this argument must always be supplied as a field name. For detailed information about calling a function and supplying arguments, see Accessing and Calling a Function.

Note: With CDN ON, numeric arguments must be delimited by a comma followed by a space.

In this chapter:

- **ABS**: Calculating Absolute Value
- **CHKPCK**: Validating a Packed Field
- **DMOD, FMOD, and IMOD**: Calculating the Remainder From a Division
- **EXP**: Raising e to the Nth Power
- **EXPN**: Evaluating a Number in Scientific Notation
- **INT**: Finding the Greatest Integer
- **LOG**: Calculating the Natural Logarithm
- **MAX and MIN**: Finding the Maximum or Minimum Value
- **NORMSDST and NORMSINV**: Calculating Normal Distributions
- **PRDNOR and PRDUNI**: Generating Reproducible Random Numbers
- **RDNORM and RDUNIF**: Generating Random Numbers
- **SQRT**: Calculating the Square Root

**ABS: Calculating Absolute Value**

Available Languages: reporting, Maintain
The ABS function returns the absolute value of a number.

**Syntax:** How to Calculate Absolute Value

ABS(\textit{in\_value})

where:

\textit{in\_value}

Numeric

Is the value for which the absolute value is returned, the name of a field that contains the value, or an expression that returns the value. 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.

\texttt{ABS(DIFF)};

For 15, the result is 15.

For -2, the result is 2.

The COMPUTE command creates the DIFF field, then ABS calculates the absolute value of DIFF:

\begin{verbatim}
TABLE FILE SALES
PRINT UNIT_SOLD AND DELIVER_AMT AND
COMPUTE DIFF/I5 = DELIVER_AMT - UNIT_SOLD; AND
COMPUTE ABS_DIFF/I5 = ABS(DIFF);BY PROD_CODE
WHERE DATE LE '1017';
END
\end{verbatim}

The output is:

<table>
<thead>
<tr>
<th>PROD_CODE</th>
<th>UNIT_SOLD</th>
<th>DELIVER_AMT</th>
<th>DIFF</th>
<th>ABS_DIFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>B10</td>
<td>30</td>
<td>30</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B17</td>
<td>20</td>
<td>40</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>B20</td>
<td>15</td>
<td>30</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>C17</td>
<td>12</td>
<td>10</td>
<td>-2</td>
<td>2</td>
</tr>
<tr>
<td>D12</td>
<td>20</td>
<td>30</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>E1</td>
<td>30</td>
<td>25</td>
<td>-5</td>
<td>5</td>
</tr>
<tr>
<td>E3</td>
<td>35</td>
<td>25</td>
<td>-10</td>
<td>10</td>
</tr>
</tbody>
</table>

**CHKPCK: Validating a Packed Field**

Available Languages: reporting, Maintain
The CHKPCK function validates the data in a field described as packed 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 Master File (USAGE and ACTUAL attributes) or the MODIFY FIXFORM command 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)
```

where:

- **length**
  - Numeric
  - Is the length of number of bytes in the packed field. It can be between 1 and 16 bytes.

- **infield**
  - Alphanumeric
  - Is the name of the packed field or the value to be verified as packed decimal. The value must be described as alphanumeric, not packed.

- **error**
  - Numeric
  - Is the error code that the function returns if a value is not packed. 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 depending on the output format.

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

\[
\text{CHKPCK}(8, \text{PACK_SAL}, -999, \text{'}P8CM\text{')} \]

1. Prepare a data source that includes invalid packed data. The following example creates TESTPACK, which contains the PACK_SAL field. PACK_SAL is defined as alphanumeric but actually contains packed data. The invalid packed data is stored as AAA.

```plaintext
DEFINE FILE EMPLOYEE
PACK_SAL/A8 = IF EMP_ID CONTAINS '123'
    THEN 'AAA' ELSE PCKOUT(CURR_SAL, 8, 'A8');
END

TABLE FILE EMPLOYEE
PRINT DEPARTMENT PACK_SAL BY EMP_ID
ON TABLE SAVE AS TESTPACK
END
```

The output is:

```
NUMBER OF RECORDS IN TABLE=       12  LINES=     12
ALPHANUMERIC RECORD NAMED TESTPACK
FIELDNAME         ALIAS          FORMAT        LENGTH
EMP_ID                      EID            A9               9
DEPARTMENT                    DPT            A10             10
PACK_SAL                                      A8               8
TOTAL                                           27
> NUMBER OF RECORDS IN TABLE=         12  LINES=      12
[EBCDIC|ALPHANUMERIC] RECORD NAMED TESTPACK
FIELDNAME         ALIAS         FORMAT        LENGTH
EMP_ID                      EID           A9               9
DEPARTMENT                    DPT           A10             10
PACK_SAL                                      A8               8
TOTAL                                           27
[DCB USED WITH FILE TESTPACK IS
DCB=(RECFM=FB,LRECL=00027,BLKSIZE=00540)] SAVED... >
```

2. Create a Master File for the TESTPACK data source. Define the PACK_SAL field as alphanumeric in the USAGE and ACTUAL attributes.

```plaintext
FILE  = TESTPACK,  SUFFIX = FIX
FIELD = EMP_ID    ,ALIAS = EID,USAGE = A9 ,ACTUAL = A9 ,$
FIELD = DEPARTMENT,ALIAS = DPT,USAGE = A10,ACTUAL = A10,$
FIELD = PACK_SAL  ,ALIAS = PS ,USAGE = A8 ,ACTUAL = A8 ,$
```

3. Create a request that uses CHKPCK to validate the values in the PACK_SAL field, and store the result in the GOOD_PACK field. Values not in packed format return the error code -999. Values in packed format appear accurately.
DEFINE FILE TESTPACK
GOOD_PACK/P8CM = CHKPCK(8, PACK_SAL, -999, GOOD_PACK);
END

TABLE FILE TESTPACK
PRINT DEPARTMENT GOOD_PACK BY EMP_ID
END

The output is:

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>DEPARTMENT</th>
<th>GOOD_PACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>071382660</td>
<td>PRODUCTION</td>
<td>$11,000</td>
</tr>
<tr>
<td>112847612</td>
<td>MIS</td>
<td>$13,200</td>
</tr>
<tr>
<td>117593129</td>
<td>MIS</td>
<td>$18,480</td>
</tr>
<tr>
<td>119265415</td>
<td>PRODUCTION</td>
<td>$9,500</td>
</tr>
<tr>
<td>119329144</td>
<td>PRODUCTION</td>
<td>$29,700</td>
</tr>
<tr>
<td>123764317</td>
<td>PRODUCTION</td>
<td>-$999</td>
</tr>
<tr>
<td>126724188</td>
<td>PRODUCTION</td>
<td>$21,120</td>
</tr>
<tr>
<td>219984371</td>
<td>MIS</td>
<td>$18,480</td>
</tr>
<tr>
<td>326179357</td>
<td>MIS</td>
<td>$21,780</td>
</tr>
<tr>
<td>451123478</td>
<td>PRODUCTION</td>
<td>-$999</td>
</tr>
<tr>
<td>543729165</td>
<td>MIS</td>
<td>$9,000</td>
</tr>
<tr>
<td>818692173</td>
<td>MIS</td>
<td>$27,062</td>
</tr>
</tbody>
</table>

DMOD, FMOD, and IMOD: Calculating the Remainder From a Division

Available Languages: reporting, Maintain

The MOD functions calculate the remainder from a division. Each function returns the remainder in a different format.

The functions use the following formula.

\[
\text{remainder} = \text{dividend} - \text{INT}(\text{dividend}/\text{divisor}) \times \text{divisor}
\]

- **DMOD** returns the remainder as a decimal number.
- **FMOD** returns the remainder as a floating-point number.
- **IMOD** returns the remainder as an integer.

For information on the INT function, see *INT: Finding the Greatest Integer* on page 455.
**Syntax:** How to Calculate the Remainder From a Division

\[ function(dividend, divisor, output) \]

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*

Numeric

Is the result whose format is determined by the function used. Can be the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.

If the divisor is zero (0), the dividend is returned.

**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.
IMOD divides ACCTNUMBER by 1000 and returns the remainder to LAST3_ACCT:

```
TABLE FILE EMPLOYEE
PRINT ACCTNUMBER AND COMPUTE
LAST3_ACCT/I3L = IMOD(ACCTNUMBER, 1000, LAST3_ACCT);
BY LAST_NAME BY FIRST_NAME
WHERE (ACCTNUMBER NE 000000000) AND (DEPARTMENT EQ 'MIS');
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>ACCTNUMBER</th>
<th>LAST3_ACCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>122850108</td>
<td>108</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>163800144</td>
<td>144</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>150150302</td>
<td>302</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>040950036</td>
<td>036</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>109200036</td>
<td>096</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>027300024</td>
<td>024</td>
</tr>
</tbody>
</table>

**EXP: Raising e to the Nth Power**

Available Languages: reporting, Maintain

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 the logarithm of the argument.

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)
```

where:

- **power**
  - Numeric
  - Is the power to which \(e\) is raised.

- **output**
  - Double-precision floating-point
  - Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.
**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.

\[ \text{EXP}(&\text{POW}, \ 'D15.3') + 0.5; \]

For 3, the result is APPROXIMATELY 20.

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 and returned to the variable &RESULT. The format of the output value is D15.3.

-SET &POW = '3';
-SET &RESULT = \text{EXP}(&\text{POW}, \ 'D15.3') + 0.5;
-\text{TYPE} \ E \ \text{TO THE} \ &\text{POW} \ \text{POWER \ IS \ APPROXIMATELY} \ &\text{RESULT}

-SET &POW = '3';
-SET &RESULT = \text{EXP}(&\text{POW}, \ 'D15.3') + 0.5;
-\text{HTMLFORM BEGIN}
  \begin{html}
  \begin{body}
  E \ \text{TO THE} \ &\text{POW} \ \text{POWER \ IS \ APPROXIMATELY} \ &\text{RESULT}
  \end{body}
  \end{html}
-\text{HTMLFORM END}

The output is:

E \ \text{TO THE 3 POWER IS APPROXIMATELY 20}

**EXPN: Evaluating a Number in Scientific Notation**

The EXPN function evaluates a numeric literal or Dialogue Manager variable expressed in scientific notation.

**Syntax:** How to Evaluate a Number in Scientific Notation

\[ \text{EXPN}(n.nn \ (E|D) \ (+|-) \ p) \]

where:

- **n.nn**
  - Numeric
  - Is a numeric literal 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 argument. The format of the result is floating-point double precision.

**Example:** Evaluating a Number in Scientific Notation

EXPN evaluates 1.03E+2.

\[
\text{EXPN}(1.03E+2)
\]

The result is 103.

**INT:** Finding the Greatest Integer

Available Languages: reporting, Maintain

The INT function returns the integer component of a number.

**Syntax:** How to Find the Greatest Integer

\[
\text{INT}(\text{in\_value})
\]

where:

\( \text{in\_value} \)

Numeric

Is the value for which the integer component is returned, the name of a field that contains the value, or an expression that returns the value. If you supply an expression, use parentheses as needed to ensure the correct order of evaluation.

**Note:** INT does not use an output argument. The format of the result is floating-point double precision.
**Example: Finding the Greatest Integer**

INT finds the greatest integer in DED_AMT.

\[
\text{INT}(\text{DED}_\text{AMT})
\]

For $1,261.40, the result is 1261.

For $1,668.69, the result is 1668.

INT finds the greatest integer in the DED_AMT field and stores it in INT_DED_AMT:

\[
\text{TABLE FILE EMPLOYEE}
\]

\[
\begin{align*}
\text{SUM } & \text{DED_AMT AND COMPUTE } \\
\text{INT_DED_AMT/19 } & = \text{INT(DED_AMT)}; \text{BY LAST_NAME BY FIRST_NAME} \\
& \text{WHERE (DEPARTMENT EQ 'MIS') AND (PAY_DATE EQ 820730)}; \\
& \text{END}
\end{align*}
\]

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>DED_AMT</th>
<th>INT_DED_AMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>$1,261.40</td>
<td>1261</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>$1,668.69</td>
<td>1668</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>$127.50</td>
<td>127</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>$725.34</td>
<td>725</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>$334.10</td>
<td>334</td>
</tr>
</tbody>
</table>

**LOG: Calculating the Natural Logarithm**

Available Languages: reporting, Maintain

The LOG function returns the natural logarithm of a number.

**Syntax: How to Calculate the Natural Logarithm**

\[
\text{LOG}\left(\text{in\_value}\right)
\]

where:

\[
\text{in\_value}
\]

Numeric

Is the value for which the natural logarithm is calculated, the name of a field that contains the value, or an expression that returns the value. If you supply an expression, use parentheses as needed to ensure the correct order of evaluation. If in_value is less than or equal to 0, LOG returns 0.

**Note:** LOG does not use an output argument. The format of the result is floating-point double precision.
**Example:** Calculating the Natural Logarithm

LOG calculates the logarithm of CURR_SAL.

\[
\text{LOG(CURR\_SAL)}
\]

For $29,700.00, the result is 10.30.

For $26,862.00, the result is 10.20.

LOG calculates the logarithm of the CURR_SAL field:

```
TABLE FILE EMPLOYEE
PRINT CURR_SAL AND COMPUTE
LOG_CURR_SAL/D12.2 = \text{LOG(CURR\_SAL)}; BY LAST\_NAME BY FIRST\_NAME
WHERE DEPARTMENT EQ 'PRODUCTION';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>CURR_SAL</th>
<th>LOG_CURR_SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>$29,700.00</td>
<td>10.30</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>$26,862.00</td>
<td>10.20</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>$16,100.00</td>
<td>9.69</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>$21,120.00</td>
<td>9.96</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>$9,500.00</td>
<td>9.16</td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>$11,000.00</td>
<td>9.31</td>
</tr>
</tbody>
</table>

**MAX and MIN: Finding the Maximum or Minimum Value**

Available Languages: reporting, Maintain

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 argument. 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.

MIN returns either the value of the ED_HRS field or the constant 30, whichever is lower:

TABLE FILE EMPLOYEE
PRINT ED_HRS AND COMPUTE
MIN_EDHRS_30/D12.2 = MIN(ED_HRS, 30);
END

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>ED_HRS</th>
<th>MIN_EDHRS_30</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>75.00</td>
<td>30.00</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>45.00</td>
<td>30.00</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>25.00</td>
<td>25.00</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>50.00</td>
<td>30.00</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>.00</td>
<td>.00</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>36.00</td>
<td>30.00</td>
</tr>
</tbody>
</table>

NORMSDST and NORMSINV: Calculating Normal Distributions

The NORMSDST and NORMSINV functions perform calculations on a standard normal distribution curve. NORMSDST calculates the percentage of data values that are less than or equal to a normalized value; NORMSINV is the inverse of NORMSDST, calculates the normalized value that forms the upper boundary of a percentile in a standard normal distribution curve.
NORMSDST: Calculating Standard Cumulative 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. For information about NORMSINV, see NORMSINV: Calculating Inverse Cumulative Normal Distribution on page 462.

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:

\[ z = \frac{\text{raw score} - \text{mean}}{\text{standard deviation}} \]

To convert from a z-score back to a raw score, use the following formula:

\[ \text{raw score} = z \times \text{standard deviation} + \text{mean} \]

The mean of data points \( x_i \), where \( i \) is from 1 to \( n \) is:

\[ \frac{\sum x_i}{n} \]

The standard deviation of data points \( x_i \), where \( i \) is from 1 to \( n \) is:

\[ \sqrt{\left( \frac{\sum x_i^2 - (\sum x_i)^2}{n - 1} \right)} \]
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.
Example: Using the NORMSDST Function

NORMSDST finds the percentile for Z and stores the result in a column with the format D8.

\[
\text{NORMSDST}(Z, \ 'D8')
\]

For -0.07298, the result is 0.47091.

For -0.80273 the result is 0.21106.

NORMSDST calculates the Z value and finds its percentile:

```
DEFINE FILE GGPRODS
-\* CONVERT SIZE FIELD TO DOUBLE PRECISION
X/D12.5 = SIZE;
END

TABLE FILE GGPRODS
SUM X NOPRINT CNT.X NOPRINT
-\* CALCULATE MEAN AND STANDARD DEVIATION
COMPUTE NUM/D12.5 = CNT.X; NOPRINT
COMPUTE MEAN/D12.5 = AVE.X; NOPRINT
COMPUTE VARIANCE/D12.5 = ((NUM*ASQ.X) - (X*X/NUM))/(NUM-1); NOPRINT
COMPUTE STDEV/D12.5 = SQRT(VARIANCE); NOPRINT
PRINT SIZE X NOPRINT
-\* COMPUTE NORMALIZED VALUES AND USE AS INPUT TO NORMSDST FUNCTION
COMPUTE Z/D12.5 = (X - MEAN)/STDEV;
COMPUTE NORMSD/D12.5 = NORMSDST(Z, \ 'D8');
BY PRODUCT_ID NOPRINT
END
```

The output is:

<table>
<thead>
<tr>
<th>Size</th>
<th>Z</th>
<th>NORMSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>-.07298</td>
<td>.47091</td>
</tr>
<tr>
<td>12</td>
<td>-.80273</td>
<td>.21106</td>
</tr>
<tr>
<td>12</td>
<td>-.80273</td>
<td>.21106</td>
</tr>
<tr>
<td>20</td>
<td>.65678</td>
<td>.74434</td>
</tr>
<tr>
<td>24</td>
<td>1.38654</td>
<td>.91721</td>
</tr>
<tr>
<td>20</td>
<td>.65678</td>
<td>.74434</td>
</tr>
<tr>
<td>24</td>
<td>1.38654</td>
<td>.91721</td>
</tr>
<tr>
<td>16</td>
<td>-.07298</td>
<td>.47091</td>
</tr>
<tr>
<td>12</td>
<td>-.80273</td>
<td>.21106</td>
</tr>
<tr>
<td>8</td>
<td>-1.53249</td>
<td>.06270</td>
</tr>
</tbody>
</table>
NORMSINV: Calculating Inverse Cumulative Normal Distribution

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. For information about NORMSDST, see NORMSDST: Calculating Standard Cumulative Normal Distribution on page 459.

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');

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
NORMSDST finds the percentile for the Z field. NORMSINV then returns this percentile to a normalized value:

```
DEFINE FILE GGPRODS
  * CONVERT SIZE FIELD TO DOUBLE PRECISION
  X/D12.5 = SIZE;
END

TABLE FILE GGPRODS
  SUM X NOPRINT CNT.X NOPRINT
  * CALCULATE MEAN AND STANDARD DEVIATION
  COMPUTE NUM/D12.5 = CNT.X; NOPRINT
  COMPUTE MEAN/D12.5 = AVE.X; NOPRINT
  COMPUTE VARIANCE/D12.5 = ((NUM*ASQ.X) - (X*X/NUM))/(NUM-1); NOPRINT
  COMPUTE STDEV/D12.5 = SQRT(VARIANCE); NOPRINT
  PRINT SIZE X NOPRINT
  * COMPUTE NORMALIZED VALUES AND USE AS INPUT TO NORMSDST FUNCTION
  * THEN USE RETURNED VALUES AS INPUT TO NORMSINV FUNCTION
  * AND CONVERT BACK TO DATA VALUES
  COMPUTE Z/D12.5 = (X - MEAN)/STDEV;
  COMPUTE NORMSD/D12.5 = NORMSDST(Z, 'D8');
  COMPUTE NORMSI/D12.5 = NORMSINV(NORMSD, 'D8');
  COMPUTE DSIZE/D12 = NORMSI * STDEV + MEAN;
  BY PRODUCT_ID NOPRINT
END
```

The output shows that NORMSINV is the inverse of NORMSDST and returns the original values:

<table>
<thead>
<tr>
<th>Size</th>
<th>Z</th>
<th>NORMSD</th>
<th>NORMSI</th>
<th>DSIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>-.07298</td>
<td>.47091</td>
<td>-.07298</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>-.80273</td>
<td>.21106</td>
<td>-.80273</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>-.80273</td>
<td>.21106</td>
<td>-.80273</td>
<td>12</td>
</tr>
<tr>
<td>20</td>
<td>.65678</td>
<td>.74434</td>
<td>.65678</td>
<td>20</td>
</tr>
<tr>
<td>24</td>
<td>1.38654</td>
<td>.91721</td>
<td>1.38654</td>
<td>24</td>
</tr>
<tr>
<td>20</td>
<td>.65678</td>
<td>.74434</td>
<td>.65678</td>
<td>20</td>
</tr>
<tr>
<td>24</td>
<td>1.38654</td>
<td>.91721</td>
<td>1.38654</td>
<td>24</td>
</tr>
<tr>
<td>16</td>
<td>-.07298</td>
<td>.47091</td>
<td>-.07298</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>-.80273</td>
<td>.21106</td>
<td>-.80273</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>-1.53249</td>
<td>.06270</td>
<td>-1.53249</td>
<td>8</td>
</tr>
</tbody>
</table>

**PRDNOR and PRDUNI: Generating Reproducible Random Numbers**

Available Languages: reporting, Maintain

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.
If PRDNOR generates a large set of numbers, they have the following properties:

- The numbers lie roughly on a bell curve, as shown in the following figure. The bell curve is highest at the 0 mark, meaning that there are more numbers closer to 0 than farther away.

- The average of the numbers is close to 0.

- The numbers can be any size, but most are between 3 and -3.

- 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).

In z/OS, the numbers do not reproduce.

**Syntax:**

How to Generate Reproducible Random Numbers

```plaintext
{PRDNOR|PRDUNI}(seed, output)
```

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 9 digits. The seed is truncated to an integer.

On z/OS, the numbers do not reproduce.

**output**

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 Reproducible Random Numbers**

PRDNOR assigns random numbers and stores them in a column with the format D12.2.

```
PRDNOR(40, 'D12.2')
```

PRDNOR assigns random numbers and stores them in RAND. These values are then used to randomly pick five employee records identified by the values in the LAST_NAME and FIRST_NAME fields. The seed is 40. To produce a different set of numbers, change the seed.

```
DEFINE FILE EMPLOYEE
RAND/D12.2 WITH LAST_NAME = PRDNOR(40, RAND);
END

TABLE FILE EMPLOYEE
PRINT LAST_NAME AND FIRST_NAME
BY HIGHEST 5 RAND
END
```

The output is:

```
RAND  LAST_NAME    FIRST_NAME
----   --------    ----------
1.38   STEVENS     ALFRED
1.12   MCCOY       JOHN
.55    SMITH       RICHARD
.21    JONES       DIANE
.01    IRVING      JOAN
```
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.
  
  If RDNORM generates a large set of numbers (between 1 and 32768), they have the following properties:
  
  - The numbers lie roughly on a bell curve, as shown in the following figure. The bell curve is highest at the 0 mark, meaning that there are more numbers closer to 0 than farther away.

  ![Bell Curve Diagram](image)

  - The average of the numbers is close to 0.
  - The numbers can be any size, but most are between 3 and -3.

- 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)

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

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')

RDNORM assigns random numbers and stores them in RAND. These numbers are then used to randomly choose five employee records identified by the values in the LAST NAME and FIRST NAME fields.

DEFINE FILE EMPLOYEE
RAND/D12.2 WITH LAST_NAME = RDNORM(RAND);END
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND FIRST_NAME
BY HIGHEST 5 RAND
END

The request produces output similar to the following:

<table>
<thead>
<tr>
<th>RAND</th>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>.65</td>
<td>CROSS</td>
<td>BARBARA</td>
</tr>
<tr>
<td>.20</td>
<td>BANNING</td>
<td>JOHN</td>
</tr>
<tr>
<td>.19</td>
<td>IRVING</td>
<td>JOAN</td>
</tr>
<tr>
<td>.00</td>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
</tr>
<tr>
<td>-.14</td>
<td>GREENSPAN</td>
<td>MARY</td>
</tr>
</tbody>
</table>
SQRT: Calculating the Square Root

Available Languages: reporting, Maintain

The SQRT function calculates the square root of a number.

Syntax: How to Calculate the Square Root

\[
\text{SQRT}(\text{in\_value})
\]

where:

\[
\text{in\_value}
\]

Numeric

Is the value for which the square root is calculated, the name of a field that contains the value, or an expression that returns the value. If you supply an expression, use parentheses as needed to ensure the correct order of evaluation. If you supply a negative number, the result is zero.

Note: SQRT does not use an output argument. The result of the function is floating-point double precision.

Example: Calculating the Square Root

SQRT calculates the square root of LISTPR.

\[
\text{SQRT(LISTPR)}
\]

For 19.98, the result is 4.47.

For 14.98, the result is 3.87.

SQRT calculates the square root of LISTPR:

\[
\begin{align*}
\text{TABLE FILE MOVIES} \\
\text{PRINT LISTPR AND COMPUTE} \\
\text{SQRT\_LISTPR/D12.2 = SQRT(LISTPR) \text{ BY TITLE}} \\
\text{WHERE CATEGORY EQ 'MUSICALS'}; \\
\text{END}
\end{align*}
\]

The output is:

<table>
<thead>
<tr>
<th>TITLE</th>
<th>LISTPR</th>
<th>SQRT_LISTPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL THAT JAZZ</td>
<td>19.98</td>
<td>4.47</td>
</tr>
<tr>
<td>CABARET</td>
<td>19.98</td>
<td>4.47</td>
</tr>
<tr>
<td>CHORUS LINE, A</td>
<td>14.98</td>
<td>3.87</td>
</tr>
<tr>
<td>FIDDLER ON THE ROOF</td>
<td>29.95</td>
<td>5.47</td>
</tr>
</tbody>
</table>
Simplified statistical functions can be called in a COMPUTE command to perform statistical calculations on the internal matrix that is generated during TABLE request processing. The STDDEV and CORRELATION functions can also be called as a verb object in a display command. Prior to calling a statistical function, you need to establish the size of the partition on which these functions will operate, if the request contains sort fields.

**Note:** It is recommended that all numbers and fields used as parameters to these functions be double-precision.

**In this chapter:**
- Specify the Partition Size for Simplified Statistical Functions
- CORRELATION: Calculating the Degree of Correlation Between Two Sets of Data
- KMEANS_CLUSTER: Partitioning Observations Into Clusters Based on the Nearest Mean Value
- MULTIREGRESS: Creating a Multivariate Linear Regression Column
- OUTLIER: Identifying Outliers in Numeric Data
- RSERVE: Running an R Script
- STDDEV: Calculating the Standard Deviation for a Set of Data Values

### Specify the Partition Size for Simplified Statistical Functions

```
SET PARTITION_ON = {FIRST | PENULTIMATE | TABLE}
```

where:

**FIRST**

Uses the first (also called the major) sort field in the request to partition the values.

**PENULTIMATE**

Uses the next to last sort field where the COMPUTE is evaluated to partition the values. This is the default value.
TABLE

Uses the entire internal matrix to calculate the statistical function.

**CORRELATION: Calculating the Degree of Correlation Between Two Sets of Data**

The CORRELATION function calculates the correlation coefficient between two numeric fields. The function returns a numeric value between zero (-1.0) and 1.0.

**Syntax:**

How to Calculate the Correlation Coefficient Between Two Fields

```plaintext
CORRELATION(field1, field2)
```

where:

- **field1**
  - Numeric
  - Is the first set of data for the correlation.

- **field2**
  - Numeric
  - Is the second set of data for the correlation.

**Note:** Arguments for CORRELATION cannot be prefixed fields. If you need to work with fields that have a prefix operator applied, apply the prefix operators to the fields in COMPUTE commands and save the results in a HOLD file. Then, run the correlation against the HOLD file.

**Example:**

Calculating a Correlation

The following request calculates the correlation between the DOLLARS and BUDDOLLARS fields converted to double precision.

```plaintext
DEFINE FILE ibisamp/ggsales
DOLLARS/D12.2 = DOLLARS;
BUDDOLLARS/D12.2 = BUDDOLLARS;
END
TABLE FILE ibisamp/ggsales
SUM DOLLARS BUDDOLLARS
CORRELATION(DOLLARS, BUDDOLLARS)
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF, $
ENDSTYLE
END
```
The output is shown in the following image.

```
  DOLLARS  BUDDOLLARS
46,156,290.00  46,220,778.00

  CORRELATION
  DOLLARS  BUDDOLLARS
          .895691073
```

CORRELATION calculates the correlation between DOLLARS and BUDDOLLARS.

```
CORRELATION(DOLLARS, BUDDOLLARS)
```

For DOLLARS=46,156,290.00 and BUDDOLLARS=46,220,778.00, the result is 0.895691073.

**KMEANS_CLUSTER: Partitioning Observations Into Clusters Based on the Nearest Mean Value**

The KMEANS_CLUSTER function partitions observations into a specified number of clusters based on the nearest mean value. The function returns the cluster number assigned to the field value passed as a parameter.

**Note:** If there are not enough points to create the number of clusters requested, the value -10 is returned for any cluster that cannot be created.

**Syntax:**

```
KMEANS_CLUSTER(number, percent, iterations, tolerance, [prefix1.]field1[, [prefix1.]field2 ...])
```

where:

```
number
  Integer
  Is number of clusters to extract.

percent
  Numeric
  Is the percent of training set size (the percent of the total data to use in the calculations).
  The default value is AUTO, which uses the internal default percent.
```
**iterations**
Integer

Is the maximum number of times to recalculate using the means previously generated. The default value is AUTO, which uses the internal default number of iterations.

**tolerance**
Numeric

Is a weight value between zero (0) and 1.0. The value AUTO uses the internal default tolerance.

**prefix1, prefix2**
Defines an optional aggregation operator to apply to the field before using it in the calculation. Valid operators are:

- **SUM.** which calculates the sum of the field values. SUM is the default value.
- **CNT.** which calculates a count of the field values.
- **AVE.** which calculates the average of the field values.
- **MIN.** which calculates the minimum of the field values.
- **MAX.** which calculates the maximum of the field values.
- **FST.** which retrieves the first value of the field.
- **LST.** which retrieves the last value of the field.

**Note:** The operators PCT., RPCT., TOT., MDN., MDE., RNK., and DST. are not supported.

**field1**
Numeric

Is the set of data to be analyzed.

**field2**
Numeric

Is an optional set of data to be analyzed.
Example: Partitioning Data Values Into Clusters

The following request partitions the DOLLARS field values into four clusters and displays the result as a scatter chart in which the color represents the cluster. The request uses the default values for the percent, iterations, and tolerance parameters by passing them as the value 0 (zero).

```
SET PARTITION_ON = PENULTIMATE
GRAPH FILE GGSALES
PRINT UNITS DOLLARS
COMPUTE KMEAN1/D20.2 TITLE 'K-MEANS' = KMEANS_CLUSTER(4, AUTO, AUTO, AUTO, DOLLARS);
ON GRAPH SET LOOKGRAPH SCATTER
ON GRAPH PCHOLD FORMAT JSCHART
ON GRAPH SET STYLE *
INCLUDE=IBFS:/FILE/IBI_HTML_DIR/ibi_themes/Warm.sty,$
type = data, column = N2, bucket=y-axis,$
type=data, column= N1, bucket=x-axis,$
type=data, column=N3, bucket=color,$
GRID=OFF,$
*GRAPH_JS_FINAL
colorScale: {
  colorMode: 'discrete',
  colorBands: [{start: 1, stop: 1.99, color: 'red'}, {start: 2, stop: 2.99, color: 'green'},
               {start: 3, stop: 3.99, color: 'yellow'}, {start: 3.99, stop: 4, color: 'blue'} ]
}
*END
ENDSTYLE
END
```
MULTIREGRESS: Creating a Multivariate Linear Regression Column

MULTIREGRESS derives a linear equation that best fits a set of numeric data points, and uses this equation to create a new column in the report output. The equation can be based on one or more independent variables.

The equation generated is of the following form, where y is the dependent variable and x1, x2, and x3 are the independent variables.

\[ y = a_1 x_1 + a_2 x_2 + a_3 x_3 + \ldots + b \]

When there is one independent variable, the equation represents a straight line. When there are two independent variables, the equation represents a plane, and with three independent variables, it represents a hyperplane. You should use this technique when you have reason to believe that the dependent variable can be approximated by a linear combination of the independent variables.

**Syntax:**

How to Create a Multivariate Linear Regression Column

MULTIREGRESS(input_field1, [input_field2, ...])
where:

\[ \text{input\_field1, input\_field2 ...} \]

Are any number of field names to be used as the independent variables. They should be independent of each other. If an input field is non-numeric, it will be categorized to transform it to numeric values that can be used in the linear regression calculation.

**Example:** Creating a Multivariate Linear Regression Column

The following request uses the DOLLARS and BUDDOLLARS fields to generate a regression column named Estimated_Dollars.

```plaintext
GRAPH FILE GGSALES
SUM BUDUNITS UNITS BUDDOLLARS DOLLARS
COMPUTE Estimated_Dollars/F8 = MULTIREGRESS(DOLLARS, BUDDOLLARS);
BY DATE
ON GRAPH SET LOOKGRAPH LINE
ON GRAPH PCHOLD FORMAT JSCHART
ON GRAPH SET STYLE *
INCLUDE=IBFS:/FILE/IBI_HTML_DIR/ibi_themes/Warm.sty,$
type=data, column = n1, bucket = x-axis,$
type=data, column= dollars, bucket=y-axis,$
type=data, column= buddollars, bucket=y-axis,$
type=data, column= Estimated_Dollars, bucket=y-axis,$
*GRAPH_JS
"series":[
{"series":2, "color":"orange"}]
*END
ENDSTYLE
END
```
OUTLIER: Identifying Outliers in Numeric Data

The output is shown in the following image. The orange line represents the regression equation.

The 1.5 * IQR (Inner Quartile Range) rule is a common way to identify outliers in data. This rule defines an outlier as a value that is above or below 1.5 times the inner quartile range in the data. The inner quartile range is based on sorting the data values, dividing it into equal quarters, and calculating the range of values between the first quartile (the value one quarter of the way through the sorted data) and third quartile (the value three quarters of the way through the sorted data). The value that is 1.5 times below the inner quartile range is called the lower fence, and the value that is 1.5 times above the inner quartile range is called the upper fence.

OUTLIER is not supported in a DEFINE expression. It can be used in a COMPUTE expression or a WHERE, WHERE TOTAL, or WHERE_GROUPED phrase.

Given a numeric field as input, OUTLIER returns one of the following values for each value of the field, using the 1.5 * IQR rule:

- **0 (zero)**. The value is not an outlier.
- **-1**. The value is below the lower fence.
- **1**. The value is above the upper fence.
**Syntax:** How to Identify Outliers in Numeric Data

\[
\text{OUTLIER}(\text{input\_field})
\]

where:

\[
\text{input\_field}
\]

Numeric

Is the numeric field to be analyzed.

**Example:** Identifying Outliers

The following request defines the SALES field to have different values depending on the store code, and uses OUTLIER to determine whether each field value is an outlier.

```
DEFINE FILE GGSALES
SALES/D12 = IF ((CATEGORY EQ 'Coffee') AND (STCD EQ 'R1019')) THEN 19000
ELSE IF ((CATEGORY EQ 'Coffee') AND (STCD EQ 'R1020')) THEN 20000
ELSE IF ((CATEGORY EQ 'Coffee') AND (STCD EQ 'R1040')) THEN 7000
ELSE DOLLARS;
END
TABLE FILE GGSALES
SUM SALES
COMPUTE OUT1/I3 = OUTLIER(SALES);
BY CATEGORY
BY STCD
WHERE CATEGORY EQ 'Coffee'
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```
The output is shown in the following image. Values above 2 million are above the upper fence, values below 1 million are below the lower fence, and other values are not outliers:

<table>
<thead>
<tr>
<th>Category</th>
<th>Store ID</th>
<th>SALES</th>
<th>OUT1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coffee</td>
<td>R1019</td>
<td>2,280,000</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>R1020</td>
<td>2,400,000</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>R1040</td>
<td>840,000</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>R1041</td>
<td>1,576,915</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>R1044</td>
<td>1,340,437</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>R1088</td>
<td>1,375,040</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>R1100</td>
<td>1,364,420</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>R1109</td>
<td>1,459,160</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>R1200</td>
<td>1,463,453</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>R1244</td>
<td>1,553,962</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>R1248</td>
<td>1,535,631</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>R1250</td>
<td>1,386,124</td>
<td>0</td>
</tr>
</tbody>
</table>

RSERVE: Running an R Script

You can use the RSERVE function in a COMPUTE command to run an R script that returns vector output. This requires that you have a configured Adapter for Rserve.

Syntax: How to Run an R Script

```
RSERVE(rserve_mf, input_field1, ...input_fieldn, output)
```

where:

- `rserve_mf`  
  Is the synonym for the R script.

- `input_field1, ...input_fieldn`  
  Are the independent variables used by the R script.

- `output`  
  Is the dependent variable returned by the R script. It must be a single column (vector) of output.
**Example:** **Using RSERVE to Run an R Script**

The R script named `wine_run_model.R` predicts Bordeaux wine prices based on the average growing season temperature, the amount of rain during the harvest season, the amount of rain during the winter, and the age of the wine.

Using a configured connection (named `MyRserve`) for the Adapter for Rserve, and a sample data file named `wine_input_sample.csv`, you create the following synonym for the R script, as described in the *Adapter Administration* manual.

**Master File**

```
FILENAME=WINE_RUN_MODEL, SUFFIX=RSERVE , $
SEGMENT=INPUT_DATA, SECTYPE=S0, $
  FIELDNAME=AGST, ALIAS=AGST, USAGE=D9.4, ACTUAL=STRING, 
  MISSING=ON, 
  TITLE='AGST', $
  FIELDNAME=HARVESTRAIN, ALIAS=HarvestRain, USAGE=I11, ACTUAL=STRING, 
  MISSING=ON, 
  TITLE='HarvestRain', $
  FIELDNAME=WINTERRAIN, ALIAS=WinterRain, USAGE=I11, ACTUAL=STRING, 
  MISSING=ON, 
  TITLE='WinterRain', $
  FIELDNAME=AGE, ALIAS=Age, USAGE=I11, ACTUAL=STRING, 
  MISSING=ON, 
  TITLE='Age', $
SEGMENT=OUTPUT_DATA, SECTYPE=U, PARENT=INPUT_DATA, $
  FIELDNAME=PRICE, ALIAS=Price, USAGE=D18.14, ACTUAL=STRING, 
  MISSING=ON, 
  TITLE='Price', $
```

**Access File**

```
SEGNAME=INPUT_DATA, 
CONNECTION=MyRserve, 
R_SCRIPT=/prediction/wine_run_model.r, 
R_SCRIPT_LOCATION=WFRS, 
R_INPUT_SAMPLE_DAT=prediction/wine_input_sample.csv, $
```
Now that the synonym has been created for the model, the model will be used to run against the following data file named wine_forecast.csv.

Year, Price, WinterRain, AGST, HarvestRain, Age, FrancePop
1952, 7.495, 600, 17.1167, 160, 31, 43183.569
1953, 8.0393, 690, 16.7333, 80, 30, 43495.03
1955, 7.6858, 502, 17.15, 130, 28, 44217.857
1957, 6.9845, 420, 16.1333, 110, 26, 45152.252
1958, 6.7772, 582, 16.4167, 187, 25, 45653.805
1959, 8.0757, 485, 17.4833, 187, 24, 46128.638
1960, 6.5188, 763, 16.4167, 290, 23, 46583.995
1961, 8.4937, 830, 17.3333, 38, 22, 47128.005
1962, 7.388, 697, 16.3, 52, 21, 48088.673
1963, 6.7127, 608, 15.7167, 155, 20, 48798.99
1964, 7.3094, 402, 17.2667, 96, 19, 49356.943
1965, 6.2518, 602, 15.3667, 118, 16, 49801.821
1966, 7.7443, 819, 16.5333, 86, 17, 50254.966
1967, 6.8398, 714, 16.2333, 118, 16, 50650.406
1968, 6.2435, 610, 16.2, 292, 15, 51034.413
1969, 6.3459, 575, 16.55, 244, 14, 51470.276
1970, 7.5883, 622, 16.6667, 89, 13, 51918.389
1971, 7.1934, 551, 16.7667, 112, 12, 52431.647
1972, 6.2049, 536, 14.9833, 158, 11, 52894.183
1973, 6.6367, 376, 17.0667, 123, 10, 53332.805
1974, 6.2941, 574, 16.3, 184, 9, 53689.61
1975, 7.292, 572, 16.95, 171, 8, 53955.042
1976, 7.1211, 418, 17.65, 247, 7, 54159.049
1977, 6.2587, 821, 15.5833, 87, 6, 54378.362
1978, 7.186, 763, 15.8167, 51, 5, 54602.193

The data file can be any type of file that R can read. In this case it is another .csv file. This file needs a synonym in order to be used in a report request. You create the synonym for this file using the Adapter for Delimited Files.

The following is the generated Master File, wine_forecast.mas.

FILENAME=WINE_FORECAST, SUFFIX=DFIX, CODEPAGE=1252,
DATASET=prediction/wine_forecast.csv, $
SEGMENT=WINE_FORECAST, SEGTYPE=S0, $
FIELDNAME=YEAR1, ALIAS=Year, USAGE=I6, ACTUAL=A5V,
MISSING=ON, TITLE='Year', $
FIELDNAME=PRICE, ALIAS=Price, USAGE=D8.4, ACTUAL=A7V,
MISSING=ON, TITLE='Price', $
FIELDNAME=WINTERRAIN, ALIAS=WinterRain, USAGE=I5, ACTUAL=A3V,
MISSING=ON, TITLE='WinterRain', $
FIELDNAME=AGST, ALIAS=AGST, USAGE=D9.4, ACTUAL=A8V,
MISSING=ON, TITLE='AGST', $
FIELDNAME=HARVESTRAIN, ALIAS=HarvestRain, USAGE=I5, ACTUAL=A3V,
MISSING=ON, TITLE='HarvestRain', $
FIELDNAME=AGE, ALIAS=Age, USAGE=I4, ACTUAL=A2V, MISSING=ON,
TITLE='Age', $
FIELDNAME=FRANCEPOP, ALIAS=FrancePop, USAGE=D11.3, ACTUAL=A11V,
MISSING=ON, TITLE='FrancePop', $

RSERVE: Running an R Script
The following is the generated Access File, wine_forecast.acx.

```
SEGNAME=WINE_FORECAST, DELIMITER=',', ENCLOSURE=',', HEADER=YES, CDN=COMMAS_DOT, CONNECTION=<local>, $
```

The following request, wine_forecast_price_report.fex, uses the RSERVE built-in function to run the script and return a report.

```
-*-wine_forecast_price_report.fex
TABLE FILE PREDICTION/WINE_FORECAST
PRINT
    YEAR
    WINTERRAIN
    AGST
    HARVESTRAIN
    AGE

    COMPUTE PREDICTED_PRICE/D18.2 MISSING ON ALL=
        RSERVE(prediction/wine_run_model, AGST, HARVESTRAIN, WINTERRAIN, AGE, Price); AS 'Predicted,Price'

ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```
The output is shown in the following image.

<table>
<thead>
<tr>
<th>Year</th>
<th>WinterRain</th>
<th>AGST</th>
<th>HarvestRain</th>
<th>Age</th>
<th>Predicted Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1952</td>
<td>600</td>
<td>17.1167</td>
<td>160</td>
<td>31</td>
<td>7.72</td>
</tr>
<tr>
<td>1953</td>
<td>690</td>
<td>16.7333</td>
<td>80</td>
<td>30</td>
<td>7.87</td>
</tr>
<tr>
<td>1955</td>
<td>502</td>
<td>17.1500</td>
<td>130</td>
<td>28</td>
<td>7.68</td>
</tr>
<tr>
<td>1957</td>
<td>420</td>
<td>16.1333</td>
<td>110</td>
<td>26</td>
<td>7.00</td>
</tr>
<tr>
<td>1958</td>
<td>582</td>
<td>16.4167</td>
<td>187</td>
<td>25</td>
<td>7.02</td>
</tr>
<tr>
<td>1959</td>
<td>485</td>
<td>17.4833</td>
<td>187</td>
<td>24</td>
<td>7.54</td>
</tr>
<tr>
<td>1960</td>
<td>763</td>
<td>16.4167</td>
<td>290</td>
<td>23</td>
<td>6.76</td>
</tr>
<tr>
<td>1961</td>
<td>830</td>
<td>17.3333</td>
<td>38</td>
<td>22</td>
<td>8.36</td>
</tr>
<tr>
<td>1962</td>
<td>697</td>
<td>16.3000</td>
<td>52</td>
<td>21</td>
<td>7.51</td>
</tr>
<tr>
<td>1963</td>
<td>608</td>
<td>15.7167</td>
<td>155</td>
<td>20</td>
<td>6.63</td>
</tr>
<tr>
<td>1964</td>
<td>402</td>
<td>17.2667</td>
<td>96</td>
<td>19</td>
<td>7.56</td>
</tr>
<tr>
<td>1965</td>
<td>602</td>
<td>15.3667</td>
<td>267</td>
<td>18</td>
<td>5.92</td>
</tr>
<tr>
<td>1966</td>
<td>819</td>
<td>16.5333</td>
<td>86</td>
<td>17</td>
<td>7.56</td>
</tr>
<tr>
<td>1967</td>
<td>714</td>
<td>16.2333</td>
<td>118</td>
<td>16</td>
<td>7.11</td>
</tr>
<tr>
<td>1969</td>
<td>575</td>
<td>16.5500</td>
<td>244</td>
<td>14</td>
<td>6.60</td>
</tr>
<tr>
<td>1970</td>
<td>622</td>
<td>16.6667</td>
<td>89</td>
<td>13</td>
<td>7.32</td>
</tr>
<tr>
<td>1971</td>
<td>551</td>
<td>16.7667</td>
<td>112</td>
<td>12</td>
<td>7.19</td>
</tr>
<tr>
<td>1972</td>
<td>536</td>
<td>14.9833</td>
<td>158</td>
<td>11</td>
<td>5.88</td>
</tr>
<tr>
<td>1973</td>
<td>376</td>
<td>17.0667</td>
<td>123</td>
<td>10</td>
<td>7.09</td>
</tr>
<tr>
<td>1974</td>
<td>574</td>
<td>16.3000</td>
<td>184</td>
<td>9</td>
<td>6.57</td>
</tr>
<tr>
<td>1975</td>
<td>572</td>
<td>16.9500</td>
<td>171</td>
<td>8</td>
<td>6.99</td>
</tr>
<tr>
<td>1976</td>
<td>418</td>
<td>17.6500</td>
<td>247</td>
<td>7</td>
<td>6.92</td>
</tr>
<tr>
<td>1977</td>
<td>821</td>
<td>15.5833</td>
<td>87</td>
<td>6</td>
<td>6.71</td>
</tr>
<tr>
<td>1978</td>
<td>763</td>
<td>15.8167</td>
<td>51</td>
<td>5</td>
<td>6.91</td>
</tr>
</tbody>
</table>
STDDEV: Calculating the Standard Deviation for a Set of Data Values

The STDDEV function returns a numeric value that represents the amount of dispersion in the data. The set of data can be specified as the entire population or a sample. The standard deviation is the square root of the variance, which is a measure of how observations deviate from their expected value (mean). If specified as a population, the divisor in the standard deviation calculation (also called degrees of freedom) will be the total number of data points, N. If specified as a sample, the divisor will be N-1.

If $x_i$ is an observation, $N$ is the number of observations, and $\mu$ is the mean of all of the observations, the formula for calculating the standard deviation for a population is:

$$\sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2}$$

To calculate the standard deviation for a sample, the mean is calculated using the sample observations, and the divisor is N-1 instead of N.

**Reference:** Calculate the Standard Deviation in a Set of Data

STDDEV(*field*, *sampling*)

where:

*field*
- Numeric
  - Is the set of observations for the standard deviation calculation.

*sampling*
- Keyword
  - Indicates the origin of the data set. Can be one of the following values.
    - P Entire population.
    - S Sample of population.

**Note:** Arguments for STDDEV cannot be prefixed fields. If you need to work with fields that have a prefix operator applied, apply the prefix operators to the fields in COMPUTE commands and save the results in a HOLD file. Then, run the standard deviation against the HOLD file.
STDDEV: Calculating the Standard Deviation for a Set of Data Values

**Example:** Calculating a Standard Deviation

The following request calculates the standard deviation of the DOLLARS field converted to double precision.

```
DEFINE FILE ibisamp/ggsales
DOLLARS/D12.2 = DOLLARS;
END
TABLE FILE ibisamp/ggsales
SUM DOLLARS STDDEV(DOLLARS,S)
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image.

```
         STDS
DOLLARS  DOLLARS
46,156,290.00  6,157.711080272
```

STDDEV calculates the standard deviation of DOLLARS.

```
STDDEV (DOLLARS,S)
```

The result is 6,157.711080272.
Simplified System Functions

Simplified system functions have streamlined parameter lists, similar to those used by SQL functions. In some cases, these simplified functions provide slightly different functionality than previous versions of similar functions.

The simplified functions do not have an output argument. Each function returns a value that has a specific data type.

When used in a request against a relational data source, these functions are optimized (passed to the RDBMS for processing).

In this chapter:

- EDAPRINT: Inserting a Custom Message in the EDAPRINT Log File
- ENCRYPT: Encrypting a Password
- GETENV: Retrieving the Value of an Environment Variable
- PUTENV: Assigning a Value to an Environment Variable
- SLACK: Posting a Message to a Slack Channel

EDAPRINT: Inserting a Custom Message in the EDAPRINT Log File

The EDAPRINT function enables you to add a text message into the EDAPRINT log file and assign it a message type. The returned value of the function is zero (0).

The EDAPRINT function enables you to allocate a sequential file to DDNAME EDAPRINT, add a text message into it, and assign it a message type.

Syntax: How to Insert a Message in the EDAPRINT Log File

EDAPRINT(message_type, 'message')
where:

**message_type**

Keyword

Can be one of the following message types.

- **I.** Informational message.
- **W.** Warning message.
- **E.** Error message.

**'message'**

Is the message to insert, enclosed in single quotation marks.

**Example:** Inserting a Custom Message in the EDAPRINT Log File

The following procedure inserts three messages in the EDAPRINT log file.

DYNAM ALLOC DD EDAPRINT DA USER1.EDAPRINT.LOG SHR REU

-SET &I = EDAPRINT(I, 'This is a test informational message');
-SET &W = EDAPRINT(W, 'This is a test warning message');
-SET &E = EDAPRINT(E, 'This is a test error message');

The output is shown in the following image.

```
01/18/2018 15:28:42.192 I DISCONNECT cmphkt000008 tscnmid=11, sesid=15, fcctk=5a6102d5:1-5, foders=ht00002010
01/18/2018 15:28:42.592 I DISCONNECT cmphkt000008 tscnmid=15, sesid=15, cpw=0.000s, ew=0.000s, svw=0.000s
01/18/2018 15:28:42.922 I DISCONNECT cmphkt000008 to exec <hbjweb> session=5a6102d5:1-5
01/18/2018 15:28:42.922 I DISCONNECT cmphkt000008 to exec connect to agent (HC_DEFAULT)
01/18/2018 15:28:42.922 I DISCONNECT cmphkt000008 tscnmid=11, sesid=15, fcctk=5a6102d5:1-5, foders=ht00002010
01/18/2018 15:28:42.922 I This is a test informational message

01/18/2018 15:28:42.927 W This is a test warning message
```

The following is the contents of the file allocated to DDNAME EDAPRINT after running the request.

```
00001 04/04/2019 10:52:15.483 I This is a test informational message
00002 04/04/2019 10:52:15.490 W This is a test warning message
00003 04/04/2019 10:52:15.518 E This is a test error message
```
**ENCRYPT: Encrypting a Password**

The ENCRYPT function encrypts an alphanumeric input value using the encryption algorithm configured in FOCUS the server. The result is returned as variable length alphanumeric.

**Syntax:**  
ENCRYPT(password)

where:

password  
Fixed length alphanumeric  
Is the value to be encrypted.

**Example:**  
Encrypting a Password

The following request encrypts the value guestpassword using the encryption algorithm configured in FOCUS the server.

-SET &P1 = ENCRYPT('guestpassword');  
-TYPE &P1

ENCRIPT encrypts the password guestpassword.

ENCRIPT('guestpassword')

The returned encrypted value is {AES}963AFA754E1763ABE697E8C5E764115E.

**GETENV: Retrieving the Value of an Environment Variable**

The GETENV function takes the name of an environment variable and returns its value as a variable length alphanumeric value.

**Syntax:**  
GETENV(var_name)

where:

var_name  
fixed length alphanumeric  
Is the name of the environment variable whose value is being retrieved.
**Example:**  **Retrieving the Value of an Environment Variable**

The following request retrieves the value of the server variable EDAEXTSEC.

```
-SET &E1 = GETENV('EDAEXTSEC');
-TYPE &E1
```

GETENV retrieves the value of the server variable EDAEXTSEC.

```
GETENV('EDAEXTSEC')
```

The value returned is ON if the server was started with security on or OFF if the server was started with security off.

**PUTENV: Assigning a Value to an Environment Variable**

The PUTENV function assigns a value to an environment variable. The function returns an integer return code whose value is 1 (one) if the assignment is not successful or 0 (zero) if it is successful.

**Syntax:**  **How to Assign a Value to an Environment Variable**

```
PUTENV(var_name, var_value)
```

where:

- `var_name`  
  Fixed length alphanumeric  
  Is the name of the environment variable to be set.

- `var_value`  
  Alphanumeric  
  Is the value you want to assign to the variable.

**Example:**  **Assigning a Value to the UNIX PS1 Variable**

The following request assigns the value `FOCUS/Shell:` to the UNIX PS1 variable.

```
-SET &P1 = PUTENV('PS1','FOCUS/Shell:');
```

PUTENV assigns the value `FOCUS/Shell:` to the UNIX PS1 variable.

```
PUTENV('PS1','FOCUS/Shell:')
```

This causes UNIX to display the following prompt when the user issues the UNIX shell command `SH`:

```
FOCUS/Shell:
```
The following request creates a variable named xxxx and sets it to the value *this is a test*. It then retrieves the value using GETENV.

```
-SET &XXXX=PUTENV(xxxx,'this is a test');
-SET &YYYY=GETENV(xxxx);
-TYPE  Return Code: &XXXX, Variable value: &YYYY
```

The output is:

Return Code: 0, Variable value: *this is a test*

**SLACK: Posting a Message to a Slack Channel**

SLACK posts a message to a Slack channel from a WebFOCUS procedure:

- If the message is sent successfully, the function returns the value *true*.
- If the message is not sent successfully, the function returns a blank.

**Syntax:**

How to Post a Message to a Slack Channel

```
SLACK(workspace, channel, message)
```

where:

*workspace*

Is a Workspace name.

*channel*

Is a Channel name.

*message*

Is an alphanumeric field containing the message.
**Example:** Sending a Slack Message From a WebFOCUS Request

The Adapter for Slack has been configured to have a connection to the devibi workspace, as shown in the following image.

The following request sends a Slack message to the *general* channel of the *devibi* Workspace, when the department is MIS.

```
TABLE FILE ibisamp/EMPLOYEE
SUM
   CURR_SAL
AND COMPUTE SLACK_MESSAGE/A200 = 'Salary for Department ' | DEPARTMENT |
   ' is ' | LJUST(20, FPRINT(CURR_SAL,'D12.2M'), 'A20');
AND COMPUTE CURR_SAL_SLACK/A20=
   IF DEPARTMENT EQ 'MIS'
   THEN SLACK('devibi', 'general', SLACK_MESSAGE) ELSE 'false';
AS 'Message Sent, to Slack highlighting, Salary'
BY DEPARTMENT
HEADING
"Slack"
"Slack Function Example"
ON TABLE SET PAGE-NUM NOLEAD
ON TABLE NOTOTAL
ON TABLE SET STYLE *
INCLUDE=IBFS:/FILE/IBI_HTML_DIR/javaassist/intl/EN/ENIADefault_combine.sty,
$ ENDSTYLE
END
```
The output is shown in the following image.

<table>
<thead>
<tr>
<th>DEPARTMENT</th>
<th>CURR_SAL</th>
<th>SLACK_MESSAGE</th>
<th>Message Sent to Slack highlighting Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIS</td>
<td>$108,002.00</td>
<td>Salary for Department MIS is $108,002.00</td>
<td>true</td>
</tr>
<tr>
<td>PRODUCTION</td>
<td>$114,282.00</td>
<td>Salary for Department PRODUCTION is $114,282.00</td>
<td>false</td>
</tr>
</tbody>
</table>

The message in the Slack channel is shown in the following image.

MedportIBI 12:32 PM
Salary for Department MIS is $108,002.00
System Functions

System functions call the operating system to obtain information about the operating environment or to use a system service.

For many functions, the output argument can be supplied either as a field name or as a format enclosed in single quotation marks. However, if a function is called from a Dialogue Manager command, this argument must always be supplied as a format, and if a function is called from a Maintain Data procedure, this argument must always be supplied as a field name. For detailed information about calling a function and supplying arguments, see Accessing and Calling a Function.

In this chapter:

- CLSDDREC: Closing All Files Opened by the PUTDDREC Function
- FEXERR: Retrieving an Error Message
- FGETENV: Retrieving the Value of an Environment Variable
- FPUTENV: Assigning a Value to an Environment Variable
- GETUSER: Retrieving a User ID
- JOBNAME: Retrieving the Current Process Identification String
- PUTDDREC: Writing a Character String as a Record in a Sequential File
- SLEEP: Suspending Execution for a Given Number of Seconds
- SYSVAR: Retrieving the Value of a z/OS System Variable

CLSDDREC: Closing 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.

For information about PUTDDREC, see PUTDDREC: Writing a Character String as a Record in a Sequential File on page 500.
Syntax: How to Close All Files Opened by the PUTDDREC Function

\[
\text{CLSDDREC}(\text{output})
\]

where:

\[
\text{output}
\]

Integer

Is the return code, which can be one of the following values:

- 0, which indicates that the files are closed.
- 1, which indicates an error while closing the files.

Example: Closing Files Opened by the PUTDDREC Function

This example closes files opened by the PUTDDREC function:

\[
\text{CLSDDREC('I1')}
\]

FEXERR: Retrieving an Error Message

Available Languages: reporting, Maintain

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

\[
\text{FEXERR(error, 'A72')}
\]

where:

\[
\text{error}
\]

Numeric

Is the error number, up to 5 digits long.
Is the format of the output value enclosed in single quotation marks. The format is A72, the maximum length of an Information Builders error message.

**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.

\[
\text{FEXERR}(&\text{ERR}, \ 'A72')
\]

The result is (FOC650) THE DISK IS NOT ACCESSED.

FEXERR retrieves the error message whose number is contained in the &ERR variable, in this case 650. The result is returned to the variable &MSGVAR and has the format A72.

\[
\text{-SET } &\text{ERR} = 650;
\text{-SET } &\text{MSGVAR} = \text{FEXERR}( &\text{ERR}, \ 'A72') ;
\text{-TYPE } &\text{MSGVAR}
\]

The output is:

(FOC650) THE DISK IS NOT ACCESSED

**FGETENV:** **Retrieving the Value of an Environment Variable**

Available Languages: reporting

The FGETENV function retrieves the value of an environment variable and returns it as an alphanumeric string.

**Syntax:**  **How to Retrieve the Value of an Environment Variable**

\[
\text{FGETENV}(\text{length}, \ '\text{varname}', \ \text{outlen}, \ \text{output})
\]

\[
\text{FGETENV}(\text{varlength}, \ '\text{varname}', \ \text{outfieldlen}, \ '\text{outfield}')
\]

where:

\[
\text{length, varlength}
\]

Integer

Is the number of characters in the environment variable name.

Is the length of the environment variable name.

\[
\text{varname}
\]

Alphanumeric
Is the name of the environment variable whose value is being retrieved.

\textit{outlenoutfieldlen}

Integer

Is the length of the environment variable value returned or a field in which the environment variable value is stored.

Is the length of the field in which the environment variable’s value is stored.

\textit{outputoutfield}

Alphanumeric

Is the format of the field in which the environment variable’s value is stored.

Is the format of the output value enclosed in single quotation marks.

**FPUTENV: Assigning a Value to an Environment Variable**

Available Operating Systems: IBM i (formerly referred to as i5/OS), Tandem, UNIX, Windows

Available Languages: reporting

The FPUTENV function assigns a character string to an environment variable. Use FPUTENV to set values that are used elsewhere in the system.

\textbf{Limit}: You cannot use FPUTENV to set or change FOCPRINT, FOCPATH, or USERPATH. Once started, these variables are held in memory and not reread from the environment.

**Syntax:** How to Assign a Value to an Environment Variable

\texttt{FPUTENV (varname_length,'varname',value_length, 'value', output)}

where:

\textit{varname_length}

Integer

Is the maximum number of characters in the name of the environment variable.

\textit{varname}

Alphanumeric

Is the name of the environment variable enclosed in single quotation marks. The name must be right-justified and padded with blanks to the maximum length specified by \textit{varname_length}.  

value\textsubscript{length}

Is the maximum length of the environment variable value.

**Note:** The sum of varname\textsubscript{length} and value\textsubscript{length} cannot exceed 64.

value

Alphanumeric

Is the value you wish to assign to the environment variable. The string must be right-justified and contain no embedded blanks. Strings that contain embedded blanks are truncated at the first blank.

output

Integer

Is the return code. It can be the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. If the variable is set successfully, the return code is 0. Any other value indicates a failure occurred.

**Example:** Assigning a Value to an Environment Variable

FPUTENV assigns the value FOCUS/Shell to the PS1 variable and stores it in a field with the format A12:

```
-SET &RC = FPUTENV(3,'PS1', 12, 'FOCUS/Shell:', 'I4');
```

The request displays the following prompt when the user issues the UNIX shell command SH:

FOCUS/Shell:

**GETUSER: Retrieving a User ID**

Available Languages: reporting, Maintain

The GETUSER function retrieves the ID of the connected user. GETUSER can also retrieve the name of a z/OS batch job if you run the function from the batch job.

**Syntax:** How to Retrieve a User ID

```
GETUSER(output)
```

where:

output

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)
```

GETUSER retrieves the user ID of the person running the request:

```
DEFINE FILE EMPLOYEE
USERID/A8 WITH EMP_ID = GETUSER(USERID);
END
```

```
TABLE FILE EMPLOYEE
SUM CURR_SAL AS 'TOTAL SALARIES'
BY DEPARTMENT
HEADING
"SALARY REPORT RUN FROM USERID: <USERID"
  " "
END
```

The output is:

```
SALARY REPORT RUN FROM USERID: doccar
DEPARTMENT TOTAL SALARIES
---------- --------------
MIS $108,002.00
PRODUCTION $114,282.00

SALARY REPORT RUN FROM USERID: USER1
DEPARTMENT TOTAL SALARIES
---------- --------------
MIS $108,002.00
PRODUCTION $114,282.00
```

**JOBNAME: Retrieving the Current Process Identification String**

The JOBNAME function retrieves the raw identification string of the current process from the operating system. This is also commonly known as a process PID at the operating system level. The function is valid in all environments, but is typically used in Dialogue Manager and returns the value as an alphanumeric string (even though a PID is pure numeric on some operating systems).
**Note:** JOBNAME strings differ between some operating systems in terms of look and length. For example, Windows, UNIX, and z/OS job names are pure numeric (typically a maximum of 8 characters long), while an OpenVMS job name is a hex number (always 8 characters long), and an IBM i job name is a three-part string that has a 26 character maximum length. Since an application may eventually be run in another (unexpected) environment in the future, it is good practice to use the maximum length of 26 to avoid accidental length truncation in the future. Applications using this function for anything more than simple identification may also need to account for the difference in the application code.

**Syntax:**

**How to Retrieve the Current Process Identification String**

```plaintext
JOBNAME(length, output)
```

where:

- **length**
  - Integer
  - Is the maximum number of characters to return from the PID system call.

- **output**
  - Alphanumeric
  - Is the returned process identification string, 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 Process Identification String**

The following example uses the JOBNAME function to retrieve the current process identification string to an A26 string and then truncate it for use in a -TYPE statement:

```plaintext
-SET &JOBNAME = JOBNAME(26, 'A26');
-SET &JOBNAME = TRUNCATE(&JOBNAME);
-TYPE The Current system PID &JOBNAME is processing.
```

For example, on Windows, the output is similar to the following:

```plaintext
The Current system PID 2536 is processing.
```
PUTDDREC: Writing 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 z/OS) command. TSO ALLOCATE does not work. 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.

For information about the FILEDEFDYNAM command, see the Overview and Operating Environments manual.

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 session connection. At the end of the request or session connection, all files opened by PUTDDREC are automatically closed.

For information about closing files opened by PUTDDREC in order to free the memory used, see CLSDDREC: Closing All Files Opened by the PUTDDREC Function on page 493.

The open, close, and write operations are handled by the operating system. Therefore, the requirements for writing to the file and the results of deviating from the instructions when calling PUTDDREC are specific to your operating environment. Make sure you are familiar with and follow the guidelines for your operating system when performing input/output operations.

You can call PUTDDREC in a DEFINE FILE command or in a DEFINE in the Master File. However, PUTDDREC does not open the file until its field name is referenced in a request.

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 session 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)

where:

ddname

Alphanumeric

Is the logical name assigned to the sequential file in a FILEDEFDYNAM 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.

It cannot be larger than the number of characters in `record_string`. To write all of `record_string` to the file, `record_len` should equal the number of characters in `record_string` and should not exceed the record length declared in the FILEDEFDYNAM command. If `record_len` is shorter than the declared length declared, the resulting file may contain extraneous characters at the end of each record. If `record_string` is longer than the declared length, `record_string` may be truncated in the resulting file.

`output`

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,

```plaintext
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
Example: Calling PUTDDREC in a TABLE Request

The following example defines a new file whose logical name is PUTDD1. The TABLE request then calls PUTDDREC for each employee in the EMPLOYEE data source and writes a record to the file composed of the employee's last name, first name, employee ID, current job code, and current salary (converted to alphanumeric using the EDIT function). The return code of zero (in OUT1) indicates that the calls to PUTDDREC were successful:

```
DYNAM ALLOC PUTDD1 DA USER1.PUTDD1.DATADATAFILEDEF PUTDD1 DISK
putdd1.dat
TABLE FILE EMPLOYEE
PRINT EMP_ID CURR_JOBCODE AS 'JOB' CURR_SAL
COMPUTE SALA/A12 = EDIT(CURR_SAL); NOPRINT
COMPUTE EMP1/A50= LAST_NAME|FIRST_NAME|EMP_ID|CURR_JOBCODE|SALA;
NOPRINT
COMPUTE OUT1/I1 = PUTDDREC('PUTDD1',6, EMP1, 50, OUT1);
BY LAST_NAME BY FIRST_NAME
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>EMP_ID</th>
<th>JOB</th>
<th>CURR_SAL</th>
<th>OUT1</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>119329144</td>
<td>A17</td>
<td>$29,700.00</td>
<td>0</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>326179357</td>
<td>B04</td>
<td>$21,780.00</td>
<td>0</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>818692173</td>
<td>A17</td>
<td>$27,062.00</td>
<td>0</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>543729165</td>
<td>A07</td>
<td>$9,000.00</td>
<td>0</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>123764317</td>
<td>A15</td>
<td>$26,862.00</td>
<td>0</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>117593129</td>
<td>B03</td>
<td>$18,480.00</td>
<td>0</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>219984371</td>
<td>B02</td>
<td>$18,480.00</td>
<td>0</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>451123478</td>
<td>B02</td>
<td>$16,100.00</td>
<td>0</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>126724188</td>
<td>B04</td>
<td>$21,120.00</td>
<td>0</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>112847612</td>
<td>B14</td>
<td>$13,200.00</td>
<td>0</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>119265415</td>
<td>A01</td>
<td>$9,500.00</td>
<td>0</td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>071382660</td>
<td>A07</td>
<td>$11,000.00</td>
<td>0</td>
</tr>
</tbody>
</table>

After running this request, the sequential file contains the following records:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>EMP_ID</th>
<th>JOB</th>
<th>CURR_SAL</th>
<th>OUT1</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>119329144</td>
<td>A17</td>
<td>$29,700.00</td>
<td>0</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>326179357</td>
<td>B04</td>
<td>$21,780.00</td>
<td>0</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>818692173</td>
<td>A17</td>
<td>$27,062.00</td>
<td>0</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>543729165</td>
<td>A07</td>
<td>$9,000.00</td>
<td>0</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>123764317</td>
<td>A15</td>
<td>$26,862.00</td>
<td>0</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>117593129</td>
<td>B03</td>
<td>$18,480.00</td>
<td>0</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>219984371</td>
<td>B02</td>
<td>$18,480.00</td>
<td>0</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>451123478</td>
<td>B02</td>
<td>$16,100.00</td>
<td>0</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>126724188</td>
<td>B04</td>
<td>$21,120.00</td>
<td>0</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>112847612</td>
<td>B14</td>
<td>$13,200.00</td>
<td>0</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>119265415</td>
<td>A01</td>
<td>$9,500.00</td>
<td>0</td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>071382660</td>
<td>A07</td>
<td>$11,000.00</td>
<td>0</td>
</tr>
</tbody>
</table>
Example: Calling PUTDDREC and CLSDDREC in Dialogue Manager -SET Commands

The following example defines a new file whose logical name is PUTDD1. The first -SET command creates a record to add to this file. The second -SET command calls PUTDDREC to add the record. The last -SET command calls CLSDDREC to close the file. The return codes are displayed to make sure operations were successful:

```
DYNAM ALLOC PUTDD1 DA USER1.PUTDD1.DATA FILEDEF PUTDD1 DISK putdd1.dat -
-SET &EMP1 = 'SMITH'|'MARY'|'A07'|'27000';
-TYPE DATA = &EMP1
-SET &OUT1 = PUTDDREC('PUTDD1',6, &EMP1, 17, 'I1');
-TYPE PUT RESULT = &OUT1
-SET &OUT1 = CLSDDREC('I1');
-TYPE CLOSE RESULT = &OUT1
```

The output is:

```
DATA = SMITHMARYA0727000
PUT RESULT = 0
CLOSE RESULT = 0
```

After running this procedure, the sequential file contains the following record:

```
SMITHMARYA0727000
```

SLEEP: Suspending Execution for a Given Number of Seconds

Available Languages: reporting, Maintain

The SLEEP function suspends execution for the number of seconds you specify as its input argument.

This function is most useful in Dialogue Manager when you need to wait to start a specific procedure. For example, you can start a FOCUS Database Server and wait until the server is started before initiating a client application.

This function is only supported in Dialogue Manager. It is useful when you need to wait to start a specific procedure or application.

Syntax: How to Suspend Execution for a Specified Number of Seconds

```
SLEEP(delay, output);
```

where:

```
delay
```

Numeric

Is the number of seconds to delay execution. The number can be specified down to the millisecond.
output

Numeric

Is the name of a field or a format enclosed in single quotation marks. The value returned is the same value you specify for delay.

Example:  Suspending Execution for Four Seconds

SLEEP suspends execution for four seconds:

-SET &DELAY = SLEEP(4.0, 'I2');

The following example computes the current date and time, suspends execution for 4 seconds, and computes the current date and time after the delay:

```plaintext
TABLE  FILE VIDEOTRK
PRINT TRANSDATE NOPRINT
COMPUTE
START_TIME/HYYMDSa = HGETC(8, START_TIME);
DELAY/I2 = SLEEP(4.0, 'I2');
END_TIME/HYYMDSa = HGETC(8, END_TIME);
IF RECORDLIMIT EQ 1
END
```

The output is:

```
START_TIME             DELAY  END_TIME
----------             -----  --------
2007/10/26  5:04:36pm      4  2007/10/26  5:04:40pm
```

SYSVAR: Retrieving the Value of a z/OS System Variable

Available Operating Systems: z/OS

The SYSVAR function populates a Dialogue Manager amper variable with the contents of any z/OS system variable. System variables are in the format [&]name [,], where the dot is optional. They can be provided by the operating system or can be user defined. The function can be called in a -SET command.
**Syntax:** How to Retrieve the Value of a z/OS System Variable

```
-SET &dmvar = SYSVAR('length','[&]sysvar[.]','outfmt');
```

where:

- **&dmvar**  
  Alphanumeric  
  Is the name of the Dialogue Manager variable to be populated with the value of the z/OS system variable.

- **length**  
  Alphanumeric  
  Is the length of the next parameter in the call. Do not include the escape character in the length, if one is present in the `sysvar` argument.

- **[&]sysvar[.]**  
  Alphanumeric  
  Is the name of the system variable to be retrieved. Note that the ampersand (&) and the dot (.) are optional. If the ampersand is included, it must be followed by the escape character (|).

- **outfmt**  
  Alphanumeric  
  Is the format of the returned value enclosed in single quotation marks.

**Example:** Retrieving the Value of the z/OS SYSNAME Variable

The following example populates the Dialogue Manager variable named &MYSNAME2 with the value of the z/OS SYSNAME variable:

```
-SET &MYSNAME2 = SYSVAR('7','SYSNAME','A8');
-TYPE SYSNAME:&MYSNAME2
```

The output is similar to the following:

```
SYSNAME:IBI1
```
Chapter 18

Simplified Geography Functions

The simplified geography functions perform location-based calculations and retrieve
géocoded points for various types of location data. They are used by the WebFOCUS
location intelligence products that produce maps and charts. Some of the geography
functions use GIS services and require valid credentials for accessing Esri ArcGIS
proprietary data.

In this chapter:

- Sample Geography Files
- GIS_DISTANCE: Calculating the Distance Between Geometry Points
- GIS_DRIVE_ROUTE: Calculating the Driving Directions Between Geometry Points
- GIS_GEOCODE_ADDR: Geocoding a Complete Address
- GIS_GEOCODE_ADDR_CITY: Geocoding an Address Line, City, and State
- GIS_GEOCODE_ADDR_POSTAL: Geocoding an Address Line and Postal Code
- GIS_GEOMETRY: Building a JSON Geometry Object
- GIS_IN_POLYGON: Determining if a Point is in a Complex Polygon
- GIS_LINE: Building a JSON Line
- GIS_POINT: Building a Geometry Point
- GIS_REVERSE_COORDINATE: Returning a Geographic Component
- GIS_SERVICE_AREA: Calculating a Geometry Area Around a Given Point
- GIS_SERV_AREA_XY: Calculating a Service Area Around a Given Coordinate
Sample Geography Files

Some of the examples for the geography functions use geography sample files. One file, esri-citibike.csv has station names, latitudes and longitudes, and trip start times and end times.

The other file, esri-geo10036.ftm has geometry data. To run the examples that use these files, create an application named esri, and place the following files into the application folder.

```
esri-citibike.mas
FILENAME=ESRI-CITIBIKE, SUFFIX=DFIX    ,
DATASET=esri/esri-citibike.csv, $
SEGMENT=CITIBIKE_TRIPDATA, SEGTYPE=S0, $
  FIELDNAME=TRIPDURATION, ALIAS=tripduration, USAGE=I7, ACTUAL=A5V, $ 
    TITLE='tripduration', $
  FIELDNAME=STARTTIME, ALIAS=starttime, USAGE=HMDYYS, ACTUAL=A18, $
    TITLE='starttime', $
  FIELDNAME=STOPTIME, ALIAS=stoptime, USAGE=HMDYYS, ACTUAL=A18, $
    TITLE='stoptime', $
  FIELDNAME=START_STATION_ID, ALIAS='start station id', USAGE=I6, ACTUAL=A4V, $
    TITLE='start station id', $
  FIELDNAME=START_STATION_NAME, ALIAS='start station name', USAGE=A79V, 
    ACTUAL=A79BV, TITLE='start station name', $
  FIELDNAME=START_STATION_LATITUDE, ALIAS='start station latitude', USAGE=P20.15, $
    ACTUAL=A18V, TITLE='start station latitude',  
    GEOGRAPHIC_ROLE=LATITUDE,  
  FIELDNAME=START_STATION_LONGITUDE, ALIAS='start station longitude', USAGE=P20.14, 
    ACTUAL=A18V, TITLE='start station longitude',  
    GEOGRAPHIC_ROLE=LONGITUDE,  
  FIELDNAME=END_STATION_ID, ALIAS='end station id', USAGE=I6, 
    ACTUAL=A4V, TITLE='end station id', $
  FIELDNAME=END_STATION_NAME, ALIAS='end station name', USAGE=A79V, 
    ACTUAL=A79BV, TITLE='end station name', $
  FIELDNAME=END_STATION_LATITUDE, ALIAS='end station latitude', USAGE=P20.15, 
    ACTUAL=A18V, TITLE='end station latitude',  
    GEOGRAPHIC_ROLE=LATITUDE,  
  FIELDNAME=END_STATION_LONGITUDE, ALIAS='end station longitude', USAGE=P20.14, 
    ACTUAL=A18V, TITLE='end station longitude',  
    GEOGRAPHIC_ROLE=LONGITUDE,  
  FIELDNAME=BIKEID, ALIAS=bikeid, USAGE=I7, ACTUAL=A5, 
    TITLE='bikeid', $
  FIELDNAME=USERTYPE, ALIAS=usertype, USAGE=A10V, ACTUAL=A10BV, 
    TITLE='usertype', $
  FIELDNAME=BIRTH_YEAR, ALIAS='birth year', USAGE=I6, ACTUAL=A4, 
    TITLE='birth year', $
  FIELDNAME=GENDER, ALIAS=gender, USAGE=I3, ACTUAL=A1, 
    TITLE='gender', $
SEGMENT=ESRIGEO, SEGTYPE=KU, SEGSUF=FIX, PARENT=CITIBIKE_TRIPDATA, 
DATASET=esri/esri-geo10036.ftm (LRECL 80 RECFM V, CRFILE=ESRI-GEO10036, $
esri-citibike.acx

SEGNAME=CITIBIKE_TRIPDATA,
DELIMITER=',',
ENCLOSURE=",",
HEADER=NO,
CDN=OFF, $

esri-citibike.csv

Note: Each complete record must be on a single line. Therefore, you must remove any line breaks that may have been inserted due to the page width in this document.

1094,11/1/2015 0:00,11/1/2015 0:18,537,Lexington Ave & E 24 St, 40.74025878,-73.98409214,531,Forsyth St & Broome St, 40.71893904,-73.99266288,23959,Subscriber,1980,1

520,11/1/2015 0:00,11/1/2015 0:08,536,1 Ave & E 30 St, 40.74144387,-73.97536082,498,Broadway & W 32 St, 40.74854862,-73.98808416,22251,Subscriber,1988,1

753,11/1/2015 0:00,11/1/2015 0:12,229,Great Jones St, 40.72743423,-73.99379025,328,Watts St & Greenwich St, 40.72405549,-74.00965965,15869,Subscriber,1981,1

353,11/1/2015 0:00,11/1/2015 0:06,285,Broadway & E 14 St, 40.73454567,-73.99074142,151,Cleveland Pl & Spring St, 40.72210379,-73.99724901,21645,Subscriber,1987,1

1285,11/1/2015 0:00,11/1/2015 0:21,268,Howard St & Centre St, 40.71910537,-73.99973337,476,E 31 St & 3 Ave,40.74394314,-73.97966069,14788,Customer,,0

477,11/1/2015 0:00,11/1/2015 0:08,379,W 31 St & 7 Ave,40.749156,-73.9916,546,E 30 St & Park Ave S,40.74444921,-73.98303529,21128,Subscriber,1962,2

362,11/1/2015 0:00,11/1/2015 0:06,407,Henry St & Poplar St, 40.700469,-73.991454,310,State St & Smith St,40.68926942,-73.98912867,21016,Subscriber,1978,1

2316,11/1/2015 0:00,11/1/2015 0:39,147,Greenwich St & Warren St, 40.71542197,-74.01121978,441,E 52 St & 2 Ave,40.756014,-73.967416,24117,Subscriber,1988,2

627,11/1/2015 0:00,11/1/2015 0:11,521,8 Ave & W 31 St, 40.75096735,-73.99444208,285,Broadway & E 14 St, 40.73454567,-73.99074142,17048,Subscriber,1986,2

1484,11/1/2015 0:01,11/1/2015 0:26,281,Grand Army Plaza & Central Park S, 40.7643971,-73.97371465,367,E 53 St & Lexington Ave, 40.75828065,-73.97069431,16779,Customer,,0
Sample Geography Files

284, 11/1/2015 0:11, 11/1/2015 0:06, 11/1/2015 0:01, Perry St & Bleecker St,
40.73535398, -74.00483091, 453, W 22 St & 8 Ave, 40.74475148, -73.99915362, 17272, Subscriber, 1976, 1

886, 11/1/2015 0:11, 11/1/2015 0:16, 492, W 33 St & 7 Ave, 40.75019995, -73.99093085, 377, 6 Ave & Canal St, 40.72243797, -74.00566443, 23019, Subscriber, 1982, 1

1379, 11/1/2015 0:11, 11/1/2015 0:24, 512, W 29 St & 9 Ave, 40.7500727, -73.99839279, 445, E 10 St & Avenue A, 40.72740794, -73.98142006, 23843, Subscriber, 1962, 2

179, 11/1/2015 0:11, 11/1/2015 0:04, 319, Fulton St & Broadway,
40.711066, -74.009447, 264, Maiden Ln & Pearl St,
40.70706456, -74.00731853, 22538, Subscriber, 1981, 1

309, 11/1/2015 0:11, 11/1/2015 0:07, 160, E 37 St & Lexington Ave,
40.748239, -73.978311, 362, Broadway & W 37 St, 40.75172632, -73.98753523, 22042, Subscriber, 1981, 1

616, 11/1/2015 0:11, 11/1/2015 0:12, 479, 9 Ave & W 45 St, 40.76019252, -73.9912551, 440, E 45 St & 3 Ave, 40.75255434, -73.97282625, 22699, Subscriber, 1982, 1

852, 11/1/2015 0:11, 11/1/2015 0:16, 346, Bank St & Hudson St,
40.73652899, -74.00618026, 375, Mercer St & Bleecker St,
40.72679454, -73.99695094, 21011, Subscriber, 1991, 1

1854, 11/1/2015 0:11, 11/1/2015 0:33, 409, DeKalb Ave & Skillman St,
40.6906495, -73.95643107, 3103, N 11 St & Wythe Ave,
40.72153267, -73.95782357, 22011, Subscriber, 1992, 1

1161, 11/1/2015 0:11, 11/1/2015 0:21, 521, 8 Ave & W 31 St, 40.75096735, -73.99444208, 461, E 20 St & 2 Ave, 40.73587678, -73.98205027, 19856, Subscriber, 1957, 1

917, 11/1/2015 0:11, 11/1/2015 0:17, 532, S 5 Pl & S 4 St, 40.710451, -73.960876, 393, E 5 St & Avenue C, 40.72299208, -73.97995466, 18598, Subscriber, 1991, 1

esri-geo10036.mas

FILENAME=ESRI-GE010036, SUFFIX=FIX     ,
DATASET=esri/esri-geo10036.ftm (LRECL 80 RECFM V, IOTYPE=STREAM, $
SEGMENT=ESRIGEO, SEGTYPE=S0, $ 
FIELDNAME=GEOMETRY, ALIAS=GEOMETRY, USAGE=TX80L, ACTUAL=TX80, 
MISSING=ON, $
esri-geo10036.ftm

"rings":
[[-73.983889998524, 40.7541490002762], [-73.980779999197, 40.7534830001404], [-73.981449998484, 40.753714000011], [-73.9824040001445, 40.7541199998382], [-73.982461000075, 40.7541340001978], [-73.9825620002361, 40.7544549999752], [-73.9834399997027, 40.7545150000673], [-73.983644399969, 40.7546397999869], [-73.9836849998628, 40.7545700003204], [-73.9837259998931, 40.7541600001147], [-73.9841276002361, 40.7545150000673], [-73.9843040003293, 40.7544549999752], [-73.9846140004357, 40.7545150000673], [-73.9847819999743, 40.7542749999714], [-73.9866590003126, 40.7550369998577], [-73.9874499996869, 40.7553720000178], [-73.9902640001834, 40.7565709999552], [-73.9914340001789, 40.7570400002697], [-73.9924290001982, 40.7574699996364], [-73.9927679996434, 40.7574699996364], [-73.9929149997025, 40.7574699996364], [-73.9929560001982, 40.7575709999552], [-73.9931059999419, 40.7577600004237], [-73.9933250001486, 40.7576700001934], [-73.9935390001247, 40.757766999869],
[[-73.9937259997542, 40.7578459998931], [-73.9939599997542, 40.757937999639], [-73.9940999998689, 40.757983999617], [-73.9941529996611, 40.7579959996157], [-73.9942220001452, 40.7580159996387], [-73.9943040003293, 40.7580300002843], [-73.994365000444, 40.7580300002843], [-73.994464999966, 40.7580369997078], [-73.9945560002591, 40.7580300002843], [-73.9946130001898, 40.7580209998693], [-73.9946589999594, 40.7580809999383], [-73.9945449997519, 40.7581149997075], [-73.994416999902, 40.758288200404], [-73.9943810002829, 40.7583400001909], [-73.9953849998179, 40.7587409997973], [-73.9959560000693, 40.7589690004191], [-73.9960649996999, 40.7590149998424], [-73.996873000888, 40.7593419996363], [-73.996975000296, 40.7593809996335], [-73.9971499998784, 40.7595379996789], [-73.9977009996014, 40.7597030000935], [-73.998039999946, 40.7598479995856], [-73.99834000014, 40.7599709998618], [-73.9987769997587, 40.7601570003453], [-73.999089999656, 40.7602540003219], [-73.9991559997021, 40.761292999672], [-73.99916340002089, 40.7613299995799], [-73.99915350001401, 40.7614539999022], [-73.99914580001865, 40.7615479997405], [-73.99903640003483, 40.7616560002242], [-73.9990130003255, 40.7617199995784], [-73.99901890003721, 40.7618369995779], [-73.9990157999269, 40.7619609999003], [-73.99906549999808, 40.7620389999], [-73.9990849998198, 40.762130001764], [-73.99908390004195, 40.7621430001993], [-73.99906839995669, 40.7622610245], [-73.9990531999752, 40.7623750001062], [-73.99903759997525, 40.7624849997829], [-73.9990284000066, 40.7625510001286], [-73.9990865999661, 40.7626509999574], [-73.99908279996624, 40.7627879999198], [-73.99905749996864, 40.763059000127], [-73.99903120000789, 40.7632720001028], [-73.99901639996189, 40.7633989996642], [-73.999081000127, 40.7636250001936], [-73.99907589998279, 40.7638500001466], [-73.99906331999622, 40.7640277004181], [-73.99906084002574, 40.7640632002565], [-73.9988419996445, 40.7642340030989], [-73.9983469999742, 40.764419999831], [-73.998179999738, 40.7646699996823], [-73.9981319995571, 40.7648580003964], [-73.9979881999855, 40.7649204996813], [-73.99799368000432, 40.7649942000224], [-73.99798479999501, 40.7650573998791], [-73.9977017001]
GIS_DISTANCE: Calculating the Distance Between Geometry Points

The GIS_DISTANCE function uses a GIS service to calculate the distance between two geometry points.

Syntax: How to Calculate the Distance Between Geometry Points

GIS_DISTANCE(geo_point1, geo_point2)

where:

geo_point1, geo_point2

Fixed length alphanumeric, large enough to hold the JSON describing the point (for example, A200).

Are the geometry points for which you want to calculate the distance.

Note: You can generate a geometry point using the GIS_POINT function.

733, 40.7653310995507], [-73.9975069996483, 40.765481000348], [-73.995379996789, 40.764625000443], [-73.99435799999348, 40.76435000003282], [-73.9936289997785, 40.7638200001929], [-73.992701000151, 40.7637539998473], [-73.9915200002646, 40.76362700002859], [-73.9905190003613, 40.7634099998023], [-73.9898629996777, 40.7622390001298], [-73.988612000044, 40.7617140000201], [-73.9880210001169, 40.7614600000179], [-73.9870289999348, 40.7609346998765], [-73.98482400002274, 40.7601130001149], [-73.9816350003452, 40.75984250002312], [-73.9781259998894, 40.7586439998208], [-73.9754799999902, 40.758315999834], [-73.9735699999526, 40.7578140000216], [-73.9721150000271, 40.7572939996184], [-73.9705670000368, 40.7566709996669], [-73.970140002958, 40.7560309998308], [-73.9694719998329, 40.75541200000638], [-73.9699399998311, 40.7547649999048], [-73.9682380000836, 40.7543610001601], [-73.9673899998524, 40.7541490002762]]

%$
Example: Calculating the Distance Between Two Geometry Points

The following uses a citibike .csv file that contains station names, latitudes and longitudes, and trip start times and end times. It uses the GIS_POINT function to define geometry points for start stations and end stations. It then uses GIS_DISTANCE to calculate the distance between them.

```
DEFINE FILE esri/esri-citibike
STARTPOINT/A200 = GIS_POINT('4326', START_STATION_LONGITUDE, START_STATION_LATITUDE);
ENDPOINT/A200 = GIS_POINT('4326', END_STATION_LONGITUDE, END_STATION_LATITUDE);
Distance/P10.2 = GIS_DISTANCE(ENDPOINT, STARTPOINT);
END
TABLE FILE esri/esri-citibike
PRINT END_STATION_NAME AS End Distance
BY START_STATION_NAME AS Start
ON TABLE SET PAGE NOLEAD
END
```
GIS_DRIVE_ROUTE: Calculating the Driving Directions Between Geometry Points

The GIS_DRIVE_ROUTE function uses a GIS service to calculate the driving route between two geometry points.

**Syntax:**

How to Calculate the Drive Route Between Geometry Points

```
GIS_DRIVE_ROUTE(geo_start_point, geo_end_point)
```
where:

**geo_start_point, geo_point2**

Fixed length alphanumeric, large enough to hold the JSON describing the point (for example, A200).

Is the starting point for which you want to calculate the drive route.

**Note:** You can generate a geometry point using the GIS_POINT function.

**geo_end_point, geo_point2**

Fixed length alphanumeric, large enough to hold the JSON describing the point (for example, A200).

Is the ending point for which you want to calculate the drive route.

**Note:** You can generate a geometry point using the GIS_POINT function.

The format of the field to which the drive route will be returned is TX.

**Example:**  **Calculating the Drive Route Between Two Geometry Points**

The following uses a citibike .csv file that contains station names, latitudes and longitudes, and trip start times and end times. It uses the GIS_POINT function to define geometry points for start stations and end stations. It then uses GIS_DRIVE_ROUTE to calculate the route to get from the end point to the start point.

```plaintext
DEFINE FILE esri/esri-citibike
STARTPOINT/A200 = GIS_POINT('4326', START_STATION_LONGITUDE, START_STATION_LATITUDE);
ENDPOINT/A200 = GIS_POINT('4326', END_STATION_LONGITUDE, END_STATION_LATITUDE);
Route/TX140 (GEOGRAPHIC_ROLE=GEOMETRY_LINE) =
    GIS_DRIVE_ROUTE(ENDPOINT, STARTPOINT);
END
TABLE FILE esri/esri-citibike
PRINT START_STATION_NAME AS Start END_STATION_NAME AS End Route
WHERE START_STATION_ID EQ 147
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
TYPE=REPORT, GRID=OFF,SIZE-11,$
ENDSTYLE
END
```
GIS_DRIVE_ROUTE: Calculating the Driving Directions Between Geometry Points

Example:
Charting a Driving Route Between Two Geometry Points

The following request uses GIS_DRIVE_ROUTE to generate a driving route between a station start point and end point and charts the route on an Esri map.

```
DEFINE FILE esri-citibike
STARTPOINT/A200 = GIS_POINT('4326', START_STATION_LONGITUDE, START_STATION_LATITUDE);
ENDPOINT/A200 = GIS_POINT('4326', END_STATION_LONGITUDE, END_STATION_LATITUDE);
Route/TX80 (GEOGRAPHIC_ROLE=GEOMETRY_LINE) = GIS_DRIVE_ROUTE(ENDPOINT, STARTPOINT);
END
```
GRAPH FILE ESRI-CITIBIKE
PRINT
  START_STATION_NAME
  END_STATION_NAME
WHERE START_STATION_ID EQ 147
  ON TABLE PCHOLD FORMAT JSCHART
  ON TABLE SET LOOKGRAPH CHOROPLETH
  ON TABLE SET EMBEDHEADING ON
  ON TABLE SET AUTOFIT ON
  ON TABLE SET STYLE *
  TYPE=REPORT, TITLETEXT='Map', PAGESIZE=E, CHART-LOOK=com.esri.map, $
  TYPE=DATA, COLUMN=N1, /*START_STATION_NAME*/
  BUCKET=tooltip, $
  TYPE=DATA, COLUMN=N2, /*END_STATION_NAME*/

*GRAPH_JS_FINAL
"legend": {"visible": true},
"extensions" : { "com.esri.map" :
  "scalebar" :
  { "scalebarUnit": "dual",
    "attachTo": "bottom-left"
  },
  "baseMapInfo": {
    "drawBasemapControl": false,
    "showArcGISBasemaps": false,
    "customBaseMaps" : [
      {"ibiBaseLayer": "dark-gray"}
    ],
  },
  "overlayLayers":
  [{
    "ibiDataLayer": {"map-geometry" : {"map_by_field" : "Route"}}, "title" : "Chart"}
  ]
},
"introAnimation": "{\"enabled\":false}"

*END
ENDSTYLE
HEADING
"Chart Drive Route"
END
GIS_GEOCODE_ADDR uses a GIS geocoding service to obtain the geometry point for a complete address.

**Syntax:**

How to Geocode a Complete Address

```plaintext
GIS_GEOCODE_ADDR(address[, country])
```

where:

- **address**
  - Fixed length alphanumeric
  - Is the complete address to be geocoded.

- **country**
  - Fixed length alphanumeric
  - Is a country name, which is optional if the country is the United States.
**Example:** Geocoding a Complete Address

The following request creates a complete address by concatenating the street address, city, state, and ZIP code. It then uses GIS_GEOCODE_ADDR to create a GIS point for the address.

```plaintext
DEFINE FILE WF_RETAIL_LITE
GADDRESS/A200 = ADDRESS_LINE_1 || '  ' | CITY_NAME || '  ' | STATE_PROV_NAME
|| ' ' | POSTAL_CODE;
GEOCODE1/A200 = GIS_GEOCODE_ADDR(GADDRESS);
END

TABLE FILE WF_RETAIL_LITE
PRINT ADDRESS_LINE_1 AS Address GEOCODE1
BY POSTAL_CODE AS Zip
WHERE CITY_NAME EQ 'New York'
WHERE POSTAL_CODE FROM '10013' TO '10020'
ON TABLE SET PAGE NOPAGE
END
```

The output is shown in the following image.

<table>
<thead>
<tr>
<th>Zip</th>
<th>Address</th>
<th>GEOCODE1</th>
</tr>
</thead>
<tbody>
<tr>
<td>10013</td>
<td>125 Worth St</td>
<td>(&quot;spatialReference&quot;: {&quot;wkt&quot;: &quot;4265&quot;}, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: 74.00269, &quot;y&quot;: 40.71542})</td>
</tr>
<tr>
<td>10016</td>
<td>139 E 35Th St</td>
<td>(&quot;spatialReference&quot;: {&quot;wkt&quot;: &quot;4265&quot;}, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: 73.97911, &quot;y&quot;: 40.74075})</td>
</tr>
<tr>
<td>10017</td>
<td>2 United Nations Plc</td>
<td>(&quot;spatialReference&quot;: {&quot;wkt&quot;: &quot;4265&quot;}, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: 73.97115, &quot;y&quot;: 40.75111})</td>
</tr>
<tr>
<td>405 E 42N4 St</td>
<td></td>
<td>(&quot;spatialReference&quot;: {&quot;wkt&quot;: &quot;4265&quot;}, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: 73.96956, &quot;y&quot;: 40.74687})</td>
</tr>
<tr>
<td>405 E 42N4 St</td>
<td></td>
<td>(&quot;spatialReference&quot;: {&quot;wkt&quot;: &quot;4265&quot;}, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: 73.96956, &quot;y&quot;: 40.74687})</td>
</tr>
<tr>
<td>219 E 42N4 St</td>
<td></td>
<td>(&quot;spatialReference&quot;: {&quot;wkt&quot;: &quot;4265&quot;}, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: 73.97333, &quot;y&quot;: 40.75901})</td>
</tr>
<tr>
<td>330 Madison Ave</td>
<td></td>
<td>(&quot;spatialReference&quot;: {&quot;wkt&quot;: &quot;4265&quot;}, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: 73.97906, &quot;y&quot;: 40.75336})</td>
</tr>
<tr>
<td>10018</td>
<td>119 W 40Th St Fl 10</td>
<td>(&quot;spatialReference&quot;: {&quot;wkt&quot;: &quot;4265&quot;}, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: 73.98599, &quot;y&quot;: 40.75398})</td>
</tr>
<tr>
<td>10019</td>
<td>11 West 40Th Street</td>
<td>(&quot;spatialReference&quot;: {&quot;wkt&quot;: &quot;4265&quot;}, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: 73.98235, &quot;y&quot;: 40.75245})</td>
</tr>
<tr>
<td>1301 Ave Of The Americas</td>
<td></td>
<td>(&quot;spatialReference&quot;: {&quot;wkt&quot;: &quot;4265&quot;}, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: 73.97776, &quot;y&quot;: 40.76044})</td>
</tr>
<tr>
<td>1345 Avenue Of The Americas</td>
<td></td>
<td>(&quot;spatialReference&quot;: {&quot;wkt&quot;: &quot;4265&quot;}, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: 73.97843, &quot;y&quot;: 40.76064})</td>
</tr>
<tr>
<td>745 Th Ave</td>
<td></td>
<td>(&quot;spatialReference&quot;: {&quot;wkt&quot;: &quot;4265&quot;}, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: 73.98340, &quot;y&quot;: 40.76077})</td>
</tr>
<tr>
<td>10020</td>
<td>1221 Avenue Of The Americas</td>
<td>(&quot;spatialReference&quot;: {&quot;wkt&quot;: &quot;4265&quot;}, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: 73.98129, &quot;y&quot;: 40.75874})</td>
</tr>
<tr>
<td>1271 Avenue Of The Americas</td>
<td></td>
<td>(&quot;spatialReference&quot;: {&quot;wkt&quot;: &quot;4265&quot;}, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: 73.98018, &quot;y&quot;: 40.76025})</td>
</tr>
</tbody>
</table>

**GIS_GEOCODE_ADDR_CITY:** Geocoding an Address Line, City, and State

GIS_GEOCODE_ADDR_CITY uses a GIS geocoding service to obtain the geometry point for an address line, city, state, and optional country. The returned value is a fixed length alphanumeric format, large enough to hold the JSON describing the geographic location (for example, A200).

**Syntax:** How to Geocode an Address Line, City, and State

```plaintext
GIS_GEOCODE_ADDR_CITY(street_addr, city, state [, country])
```

Functions Reference 519
where:

street_addr
Fixed length alphanumeric
Is the street address to be geocoded.

city
Fixed length alphanumeric
Is the city name associated with the street address.

state
Fixed length alphanumeric
Is the state name associated with the street address.

country
fixed length alphanumeric
Is a country name, which is optional if the country is the United States.

Example: Geocoding a Street Address, City, and State

The following request geocodes a street address using GIS_GEOCODE_ADDR_CITY.

DEFINE FILE WF_RETAIL_LITE
GEOCODE1/A200 = GIS_GEOCODE_ADDR_CITY(ADDRESS_LINE_1, CITY_NAME ,
STATE_PROV_NAME);
END
TABLE FILE WF_RETAIL_LITE
PRINT ADDRESS_LINE_1 AS Address GEOCODE1
BY POSTAL_CODE AS Zip
WHERE CITY_NAME EQ 'New York'
WHERE POSTAL_CODE FROM '10013' TO '10020'
ON TABLE SET PAGE NOPAGE
END
GIS_GEOCODE_ADDR_POSTAL: Geocoding an Address Line and Postal Code

GIS_GEOCODE_ADDR_POSTAL uses a GIS geocoding service to obtain the geometry point for an address line, postal code and optional country. The returned value is a fixed length alphanumeric format, large enough to hold the JSON describing the geographic location (for example, A200).

**Syntax:**

How to Geocode an Address Line and Postal Code

GIS_GEOCODE_ADDR_POSTAL( street_addr, postal_code [, country] )

where:

- **street_addr**
  
  fixed length alphanumeric

  Is the street address to be geocoded.

- **postal_code**
  
  fixed length alphanumeric

  Is the postal code associated with the street address.

- **country**
  
  fixed length alphanumeric

  Is a country name, which is optional if the country is the United States.

The output is shown in the following image.
Example:  Geocoding a Street Address and Postal Code

The following request geocodes a street address using GIS_GEOCODE_ADDR_POSTAL.

```
DEFINE FILE WF_RETAIL_LITE
GEOCODE1/A200 = GIS_GEOCODE_ADDR_POSTAL(ADDRESS_LINE_1, POSTAL_CODE);
END
TABLE FILE WF_RETAIL_LITE
PRINT ADDRESS_LINE_1 AS Address GEOCODE1
BY POSTAL_CODE AS Zip
WHERE CITY_NAME EQ 'New York'
WHERE POSTAL_CODE FROM '10013' TO '10020'
ON TABLE SET PAGE NOPAGE
END
```

The output is shown in the following image.

<table>
<thead>
<tr>
<th>Zip</th>
<th>Address</th>
<th>GEOCODE1</th>
</tr>
</thead>
<tbody>
<tr>
<td>10015</td>
<td>125 Worth St</td>
<td>{'spatialReference': {'wkid': 4326}, 'geometryType': 'esriGeometryPoint', 'geometry': {'x': 74.00269, 'y': 40.71543}}</td>
</tr>
<tr>
<td>10016</td>
<td>139 E 35Th St</td>
<td>{'spatialReference': {'wkid': 4326}, 'geometryType': 'esriGeometryPoint', 'geometry': {'x': 73.97911, 'y': 40.74705}}</td>
</tr>
<tr>
<td>10017</td>
<td>2 United Nations Plz</td>
<td>{'spatialReference': {'wkid': 4326}, 'geometryType': 'esriGeometryPoint', 'geometry': {'x': 73.97115, 'y': 40.75111}}</td>
</tr>
<tr>
<td>405</td>
<td>42Nd St</td>
<td>{'spatialReference': {'wkid': 4326}, 'geometryType': 'esriGeometryPoint', 'geometry': {'x': 73.96956, 'y': 40.74867}}</td>
</tr>
<tr>
<td>405</td>
<td>42Nd St</td>
<td>{'spatialReference': {'wkid': 4326}, 'geometryType': 'esriGeometryPoint', 'geometry': {'x': 73.96956, 'y': 40.74867}}</td>
</tr>
<tr>
<td>219</td>
<td>42Nd St</td>
<td>{'spatialReference': {'wkid': 4326}, 'geometryType': 'esriGeometryPoint', 'geometry': {'x': 73.97333, 'y': 40.75030}}</td>
</tr>
<tr>
<td></td>
<td>330 Madison Ave</td>
<td>{'spatialReference': {'wkid': 4326}, 'geometryType': 'esriGeometryPoint', 'geometry': {'x': 73.97906, 'y': 40.73316}}</td>
</tr>
<tr>
<td></td>
<td>119 W 40Th St</td>
<td>{'spatialReference': {'wkid': 4326}, 'geometryType': 'esriGeometryPoint', 'geometry': {'x': 73.98599, 'y': 40.73988}}</td>
</tr>
<tr>
<td></td>
<td>11 West 40Th St</td>
<td>{'spatialReference': {'wkid': 4326}, 'geometryType': 'esriGeometryPoint', 'geometry': {'x': 73.98235, 'y': 40.75245}}</td>
</tr>
<tr>
<td></td>
<td>31 West 52Nd St</td>
<td>{'spatialReference': {'wkid': 4326}, 'geometryType': 'esriGeometryPoint', 'geometry': {'x': 73.97776, 'y': 40.76044}}</td>
</tr>
<tr>
<td></td>
<td>1301 Ave Of The Americas</td>
<td>{'spatialReference': {'wkid': 4326}, 'geometryType': 'esriGeometryPoint', 'geometry': {'x': 73.97945, 'y': 40.78125}}</td>
</tr>
<tr>
<td></td>
<td>1345 Avenue Of The Americas</td>
<td>{'spatialReference': {'wkid': 4326}, 'geometryType': 'esriGeometryPoint', 'geometry': {'x': 73.97806, 'y': 40.76309}}</td>
</tr>
<tr>
<td></td>
<td>745 7Th Ave</td>
<td>{'spatialReference': {'wkid': 4326}, 'geometryType': 'esriGeometryPoint', 'geometry': {'x': 73.98340, 'y': 40.76077}}</td>
</tr>
<tr>
<td></td>
<td>1231 Avenue Of The Americas</td>
<td>{'spatialReference': {'wkid': 4326}, 'geometryType': 'esriGeometryPoint', 'geometry': {'x': 73.98129, 'y': 40.75874}}</td>
</tr>
<tr>
<td></td>
<td>1271 Avenue Of The Americas</td>
<td>{'spatialReference': {'wkid': 4326}, 'geometryType': 'esriGeometryPoint', 'geometry': {'x': 73.98618, 'y': 40.4076025}}</td>
</tr>
</tbody>
</table>

**GIS_GEOMETRY: Building a JSON Geometry Object**

The GIS_GEOMETRY function builds a JSON Geometry object given a geometry type, WKID, and a geometry.

**Syntax:**  How to Build a JSON Geometry Object

```
GIS_GEOMETRY(geotype, wkid, geometry)
```

where:

`geotype`  
Alphanumeric

Is a geometry type, for example, 'esriGeometryPolygon', 'esriGeometryPolyline', 'esriGeometryMultipoint', 'EsriGeometryPoint', 'EsriGeometryExtent'..
**wkid**

Alphanumeric

Is a valid spatial reference ID. WKID is an abbreviation for Well-Known ID, which identifies a projected or geographic coordinate system.

**geometry**

TX

A geometry in JSON.

The output is returned as TX.

**Example:** Building a JSON Geometry Object

The following request builds a polygon geometry of the area encompassing ZIP code 10036 in Manhattan. The input geometry object is stored in a text (.ftm) file that is cross-referenced in the esri-citibike Master File. The field containing the geometry object is GEOMETRY.

```sql
DEFINE FILE esri/esri-citibike
WKID/A10  = '4326';
MASTER_GEOMETRY/TX256 (GEOGRAPHIC_ROLE=GEOMETRY_AREA) =
    GIS_GEOMETRY( 'esriGeometryPolygon', WKID , GEOMETRY );
END
TABLE FILE esri/esri-citibike
PRINT
    START_STATION_NAME AS Station
    START_STATION_LATITUDE AS Latitude
    START_STATION_LONGITUDE AS Longitude
    MASTER_GEOMETRY AS 'JSON Geometry Object'
WHERE START_STATION_ID EQ 479
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
type=report, grid=off, size=10,$
ENDSTYLE
END
```
Example:

Charting a Geometry Object

The following request uses GIS_GEOMETRY to build a geometry object and chart it on an Esri map.

**Example Code:**

```sql
DEFINE FILE esri-citibike
WKID/A10 = '4326';
MASTER_GEOMETRY/TX256 (GEOGRAPHIC_ROLE=GEOMETRY_AREA) = GIS_GEOMETRY( 'esriGeometryPolygon', WKID , GEOMETRY );
END

GRAPH FILE ESRI-CITIBIKE
PRINT
START_STATION_NAME
END_STATION_NAME
ON TABLE PCHOLD FORMAT JSCART
ON TABLE SET LOOKGRAPH CHOROPLETH
ON TABLE SET EMBEDHEADING ON
ON TABLE SET AUTOFIT ON
ON TABLE SET STYLE *
TYPE=REPORT, TITLETEXT='Map', PAGESIZE=E, CHART-LOOK=com.esri.map,
TYPE=DATA, COLUMN=N1, /*START_STATION_NAME*/
BUCKET=tooltip,
TYPE=DATA, COLUMN=N2, /*END_STATION_NAME*/
```

**Output Description:**

The output is shown in the following image.
The output is shown in the following image.
GIS_IN_POLYGON: Determining if a Point is in a Complex Polygon

Given a point and a polygon definition, the GIS_IN_POLYGON function returns the value 1 (TRUE) if the point is in the polygon or 0 (FALSE) if the point is not in the polygon. The value is returned in integer format.

Syntax: How to Determine if a Point is in a Complex Polygon

GIS_IN_POLYGON(point, polygon_definition)

where:

point
Alphanumeric or text
Is the geometry point.

polygon_definition
Text
Is the geometry area (polygon) definition.

Example: Determining if a Point is in a Polygon

The following example determines if a station is inside ZIP code 10036. GIS_IN_POLYGON returns 1 for a point inside the polygon definition and 0 for a point outside. The polygon definition being passed is the same one used in the example for the GIS_GEOMETRY function described previously and defines the polygon for ZIP code 10036 in Manhattan in New York City. The value 1 is translated to Yes and 0 to No for display on the output.

DEFINE FILE esri/esri-citibike
WKID/A10 = '4326';
MASTER_GEOMETRY/TX256 (GEOGRAPHIC_ROLE=GEOMETRY_AREA) =
  GIS_GEOMETRY('esriGeometryPolygon', WKID, GEOMETRY);
START_STATION_POINT/A200=GIS_POINT(WKID, START_STATION_LONGITUDE,
START_STATION_LATITUDE);
STATION_IN_POLYGON/I4=GIS_IN_POLYGON(START_STATION_POINT, MASTER_GEOMETRY);
IN_POLYGON/A5 = IF STATION_IN_POLYGON EQ 1 THEN 'Yes' ELSE 'No';
END
TABLE FILE esri/esri-citibike
PRINT
START_STATION_NAME AS Station
IN_POLYGON AS 'Station in zip, code 10036?'
BY START_STATION_ID AS 'Station ID'
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
type=report, grid=off, size=10,$
type=data, column=in_polygon, style=bold, color=red, when = in_polygon eq 'Yes',$
ENDSTYLE
END
GIS_LINE: Building a JSON Line

Given two geometry points or lines, GIS_LINE builds a JSON line. The output is returned in text format.

Syntax: How to Build a JSON Line

```
GIS_LINE(geometry1, geometry2)
```

where:

- `geometry1`  
  Alphanumeric or text  
  Is the first point or line for defining the beginning of the new line.
Alphanumeric or text

Is the second point or line for the concatenation of the new line.

**Example:**  Building a JSON Line

The following request prints start stations and end stations and builds a JSON line between them.

```plaintext
DEFINE FILE ESRI/ESRI-CITIBIKE
STARTPOINT/A200 = GIS_POINT('4326', START_STATION_LONGITUDE,
START_STATION_LATITUDE);
ENDPOINT/A200 = GIS_POINT('4326', END_STATION_LONGITUDE,
END_STATION_LATITUDE);
CONNECTION_LINE/TX80 (GEOGRAPHIC_ROLE=GEOMETRY_LINE) =
   GIS_LINE(STARTPOINT, ENDPOINT);
END
TABLE FILE ESRI/ESRI-CITIBIKE
PRINT END_STATION_NAME AS End CONNECTION_LINE AS 'Connecting Line'
   BY START_STATION_NAME AS Start
WHERE START_STATION_NAME LE 'D'
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
TYPE=REPORT, GRID=OFF,$
ENDSTYLE
END
```
Example: Charting Geometry Lines

The following request generates geometry lines and charts them on an Esri map.

```sql
DEFINE FILE ESRI-CITIBIKE
CONNECTION_LINE/TX80 (GEOGRAPHIC_ROLE=GEOMETRY_LINE)
=GIS_LINE(START_STATION_POINT, END_STATION_POINT);
DISTANCE/P33.11 TITLE 'Distance'=GIS_DISTANCE(START_STATION_POINT, END_STATION_POINT);
END
```
GRAPH FILE ESRI-CITIBIKE
PRINT
   START_STATION_NAME
   END_STATION_NAME
   DISTANCE
ON TABLE PCHOLD FORMAT JSCHART
ON TABLE SET LOOKGRAPH BUBBLEMAP
ON TABLE SET EMBEDHEADING ON
ON TABLE SET AUTOFIT ON
ON TABLE SET STYLE *
   TYPE=REPORT, TITLETEXT='Map', PAGESIZE=E, CHART-LOOK=com.esri.map, $  
   TYPE=DATA, COLUMN=N1, /*START_STATION_NAME*/
      BUCKET=tooltip, $  
   TYPE=DATA, COLUMN=N2, /*END_STATION_NAME*/
      BUCKET=tooltip, $  
   TYPE=DATA, COLUMN=N3, /*DISTANCE*/
      BUCKET=tooltip, $  
*GRAPH_JS_FINAL
"legend": {"visible": true},
"extensions" : { "com.esri.map" :
   { "scalebar" :
      {
        "scalebarUnit": "dual",
        "attachTo" : "bottom-left"
      },
      "baseMapInfo": {
        "drawBasemapControl" : false,
        "showArcGISBasemaps" : false,
        "customBaseMaps" : [ {
          "ibiBaseLayer" : "dark-gray"
        } ]
      },
      "overlayLayers":{
        { "ibiDataLayer": {"map-geometry" : {"map_by_field" : "CONNECTION_LINE"}},
        "title" : "Chart"}
      },
      "introAnimation": "{\"enabled\":false}"
   }
}
*END
ENDSTYLE
HEADING
   "Chart Geometry Lines"
END
The output is shown in the following image.

![GIS_POINT: Building a Geometry Point](image)

**GIS_POINT: Building a Geometry Point**

Given a WKID (Well-Known ID) spatial reference, longitude, and latitude, the GIS_POINT function builds a JSON point defining a Geometry object with the provided WKID, longitude, and latitude. The function is optimized for those SQL engines that can build a JSON geometry object.

The field to which the point is returned should have fixed length alphanumeric format, large enough to hold the JSON describing the point (for example, A200).

**Syntax:**  

How to Build a Geometry Point  

\[
\text{GIS\_POINT}(\text{wkid, longitude, latitude})
\]

where:

- **wkid**
  - Fixed length alphanumeric
  - Is a spatial reference code (WKID). WKID is an abbreviation for Well-Known ID, which identifies a projected or geographic coordinate system.

- **longitude**
  - D20.8
  - Is the longitude for the point.
**Example:** Building a Geometry Point

The following request uses the spatial reference code 4326 (decimal degrees) and state capital longitudes and latitudes to build a geometry point.

```plaintext
DEFINE FILE WF_RETAIL_LITE
GPOINT/A200 = GIS_POINT('4326', STATE_PROV_CAPITAL_LONGITUDE,
STATE_PROV_CAPITAL_LATITUDE);
END

TABLE FILE WF_RETAIL_LITE
SUM FST.STATE_PROV_CAPITAL_LONGITUDE AS Longitude
FST.STATE_PROV_CAPITAL_LATITUDE AS Latitude
FST.GPOINT AS Point
BY STATE_PROV_CAPITAL_NAME AS Capital
WHERE COUNTRY_NAME EQ 'United States'
WHERE STATE_PROV_CAPITAL_NAME LT 'C'
ON TABLE SET PAGE NOPAGE
END
```

The output is shown in the following image.

<table>
<thead>
<tr>
<th>Capital</th>
<th>Longitude</th>
<th>Latitude</th>
<th>Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albany</td>
<td>-73.7600000000</td>
<td>42.6600000000</td>
<td>[&quot;spatialReference&quot;: &quot;wkid&quot;: 4326&quot;, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -73.7600000000, &quot;y&quot;: 42.6600000000}]</td>
</tr>
<tr>
<td>Annapolis</td>
<td>-76.4900000000</td>
<td>38.9500000000</td>
<td>[&quot;spatialReference&quot;: &quot;wkid&quot;: 4326&quot;, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -76.4900000000, &quot;y&quot;: 38.9500000000}]</td>
</tr>
<tr>
<td>Atlanta</td>
<td>-84.2700000000</td>
<td>33.9400000000</td>
<td>[&quot;spatialReference&quot;: &quot;wkid&quot;: 4326&quot;, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -84.2700000000, &quot;y&quot;: 33.9400000000}]</td>
</tr>
<tr>
<td>Augusta</td>
<td>-89.7700000000</td>
<td>44.3200000000</td>
<td>[&quot;spatialReference&quot;: &quot;wkid&quot;: 4326&quot;, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -89.7700000000, &quot;y&quot;: 44.3200000000}]</td>
</tr>
<tr>
<td>Austin</td>
<td>-97.7500000000</td>
<td>30.4000000000</td>
<td>[&quot;spatialReference&quot;: &quot;wkid&quot;: 4326&quot;, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -97.7500000000, &quot;y&quot;: 30.4000000000}]</td>
</tr>
<tr>
<td>Baton Rouge</td>
<td>-91.1700000000</td>
<td>30.3800000000</td>
<td>[&quot;spatialReference&quot;: &quot;wkid&quot;: 4326&quot;, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -91.1700000000, &quot;y&quot;: 30.3800000000}]</td>
</tr>
<tr>
<td>Bismarck</td>
<td>-100.7700000000</td>
<td>46.8200000000</td>
<td>[&quot;spatialReference&quot;: &quot;wkid&quot;: 4326&quot;, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -100.7700000000, &quot;y&quot;: 46.8200000000}]</td>
</tr>
<tr>
<td>Boise</td>
<td>-116.1600000000</td>
<td>43.6000000000</td>
<td>[&quot;spatialReference&quot;: &quot;wkid&quot;: 4326&quot;, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -116.1600000000, &quot;y&quot;: 43.6000000000}]</td>
</tr>
<tr>
<td>Boston</td>
<td>-71.1000000000</td>
<td>42.3500000000</td>
<td>[&quot;spatialReference&quot;: &quot;wkid&quot;: 4326&quot;, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -71.1000000000, &quot;y&quot;: 42.3500000000}]</td>
</tr>
</tbody>
</table>
**Example:** Charting Geometry Points

The following request generates geometry points using GIS_POINT charts them on an Esri map.

```plaintext
DEFINE FILE WF_RETAIL
GPOINT/A200 = GIS_POINT('4326', STATE_PROV_CAPITAL_LONGITUDE,
STATE_PROV_CAPITAL_LATITUDE);
END

GRAPH FILE WF_RETAIL
PRINT
STATE_PROV_NAME
WHERE STATE_PROV_CAPITAL_LONGITUDE NE MISSING
ON TABLE PCHOLD FORMAT JSCHART
ON TABLE SET LOOKGRAPH BUBBLEMAP
ON TABLE SET EMBEDHEADING ON
ON TABLE SET AUTOFIT ON
ON TABLE SET STYLE *
  TYPE=REPORT, TITLETEXT='Map', PAGESIZE=E, CHART-LOOK=com.esri.map, $
  TYPE=DATA, COLUMN=N1,
  BUCKET=tooltip, $

  "bubbleMarker": {"maxSize": "10%"},
  "legend": {"visible": true},
  "extensions" : { "com.esri.map" :
    { "scalebar" :
      { "scalebarUnit": "dual",
        "attachTo" : "bottom-left"
      },
      "baseMapInfo": {
        "drawBasemapControl" : false,
        "showArcGISBasemaps" : false,
        "customBaseMaps" : [
          {"ibiBaseLayer" : "gray"}
        ]
      },
      "overlayLayers":
        {{
          "ibiDataLayer": {"map-geometry" : {"map_by_field" : "GPOINT"}},
          "title" : "Report"}
        },
      "introAnimation": "{\"enabled\":false}"
    }
  }

*END
ENDSTYLE
HEADING
"Chart Geometry Points"
END
```
GIS_REVERSE_COORDINATE: Returning a Geographic Component

Given longitude and latitude values and the name of a geographic component, GIS_REVERSE_COORDINATE returns the specified geographic component values associated with those coordinates.

Syntax: How to Return a Geographic Component

GIS_REVERSE_COORDINATE(longitude, latitude, component)

where:

longitude
   Numeric
   Is the longitude of the component to return.

latitude
   Numeric
   Is the latitude of the component to return.
component

Keyword

Is one of the following components:

- MATCH_ADDRESS, which returns the matching address.
- METROAREA, which returns the metro area name.
- REGION, which returns the region name.
- SUBREGION, which returns the subregion name.
- CITY, which returns the city name.
- POSTAL, which returns the postal code.

The value is returned as text and can be assigned to a field with text or alphanumerical (fixed or variable length) format.

Example: Returning Geographic Components Associated With Coordinates

The following request uses city longitude and city latitude to return the matching address, postal code, region, and subregion.

```
TABLE FILE WF_RETAIL_GEOGRAPHY
SUM FST.CITY_LONGITUDE AS Longitude FST.CITY_LATITUDE AS Latitude
COMPUTE
MatchingAddress/A250 = GIS_REVERSE_COORDINATE(CITY_LONGITUDE, 
CITY_LATITUDE, MATCH_ADDRESS);
PostalCode/A250 = GIS_REVERSE_COORDINATE(CITY_LONGITUDE, 
CITY_LATITUDE, POSTAL);
Region/A250 = GIS_REVERSE_COORDINATE(CITY_LONGITUDE, 
CITY_LATITUDE, REGION);
Subregion/A250 = GIS_REVERSE_COORDINATE(CITY_LONGITUDE, 
CITY_LATITUDE, SUBREGION);

BY CITY_NAME AS City
WHERE COUNTRY_NAME EQ 'United States'
WHERE TOTAL PostalCode NE ' '
WHERE RECORDLIMIT EQ 20
ON TABLE SET PAGE NOLEAD
END
```
GIS_REVERSE_COORDINATE returns the REGION, given a city longitude and city latitude.

GIS_REVERSE_COORDINATE(CITY_LONGITUDE, CITY_LATITUDE, REGION)

For Annapolis, the result is Maryland.
For Baton Rouge, the result is Louisiana.

GIS_SERVICE_AREA: Calculating a Geometry Area Around a Given Point

The GIS_SERVICE_AREA function uses a GIS service to calculate the geometry area with access boundaries within the given time or distance from the provided geometry point. The output is returned in text format.

Syntax:

GIS_SERVICE_AREA(geo_point, distance, travel_mode)

where:

geo_point
Alphanumeric
Is the starting geometry point.

distance
Alphanumeric
Is the travel limitation in either time or distance units.
travel_mode

Alphanumeric

Is a valid travel mode as defined in gis_serv_area.mas in the Catalog directory under the server installation directory. The accepted travel modes are:

- 'Miles'. This is the default value.
- 'TravelTime'.
- 'TruckTravelTime'.
- 'WalkTime'.
- 'Kilometers'.

**Example:** Calculating a Service Area Around a Geometry Point

The following request calculates the geometry area that is a five-minute walk around a station.

```plaintext
DEFINE FILE esri/esri-citibike
WKID/A10='4326';
START_STATION_POINT/A200=GIS_POINT(WKID, START_STATION_LONGITUDE,
START_STATION_LATITUDE);
DISTANCE/A10='5';
TRAVEL_MODE/A10='WalkTime';
STATION_SERVICE_AREA/TX80 (GEOGRAPHIC_ROLE=GEOMETRY_AREA)=
GIS_SERVICE_AREA(START_STATION_POINT, DISTANCE, TRAVEL_MODE);
END
TABLE FILE esri/esri-citibike
PRINT
   START_STATION_ID AS 'Station ID'
   START_STATION_NAME AS 'Station Name'
   STATION_SERVICE_AREA AS '5-Minute Walk Service Area Around Station'
WHERE START_STATION_ID EQ 479 OR 512;
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
TYPE=REPORT, GRID=OFF, SIZE=12,$
ENDSTYLE
END
```
Example: Charting a Geometry Service Area Around a Point

The following request generates service areas that are 5-minute walking distances from start station geometry points and charts them on an Esri map.

```
DEFINE FILE esri-citibike
WKID/A10='4326';
START_STATION_POINT/A200=GIS_POINT(WKID, START_STATION_LONGITUDE, START_STATION_LATITUDE);
DISTANCE/A10='5';
TRAVEL_MODE/A10='WalkTime';
STATION_SERVICE_AREA/TX80 (GEOGRAPHIC_ROLE=GEOMETRY_AREA)=GIS_SERVICE_AREA(START_STATION_POINT, DISTANCE, TRAVEL_MODE);
END
```
GRAPH FILE ESRI-CITIBIKE
PRINT
  START_STATION_NAME
  END_STATION_NAME
  DISTANCE
ON TABLE PCHOLD FORMAT JSCHART
ON TABLE SET LOOKGRAPH CHOROPLETH
ON TABLE SET EMBEDHEADING ON
ON TABLE SET AUTOFIT ON
ON TABLE SET STYLE *
  TYPE=REPORT, TITLETEXT='Map', PAGESIZE=E, CHART-LOOK=com.esri.map, $
  TYPE=DATA, COLUMN=N1, /*START_STATION_NAME*/
    BUCKET=tooltip, $
  TYPE=DATA, COLUMN=N2, /*END_STATION_NAME*/
    BUCKET=tooltip, $
  TYPE=DATA, COLUMN=N3, /*DISTANCE*/
    BUCKET=tooltip, $

*GRAPH_JS_FINAL
"legend": {"visible": true},
"extensions" : { "com.esri.map" :
  "scalebar" :
    { "scalebarUnit": "dual",
      "attachTo" : "bottom-left"
    },
  "baseMapInfo": {
    "drawBasemapControl" : false,
    "showArcGISBasemaps" : false,
    "customBaseMaps" : [ {
      "ibiBaseLayer" : "dark-gray"
    } ]
  },
  "overlayLayers": {
    "ibiDataLayer": {"map-geometry" : {"map_by_field" : "STATION_SERVICE_AREA"}}, "title" : "Chart"}
  },
"introAnimation": "{\"enabled\":false}"
}

*END
ENDSTYLE
HEADING
  "Chart Geometry Service Area"
END
The output is shown in the following image.

GIS_SERV_AREA_XY: Calculating a Service Area Around a Given Coordinate

The GIS_SERV_AREA_XY function uses a GIS service to calculate the geometry area with access boundaries within the given time or distance from the provided coordinate. The output is returned in text format.

**Syntax:** How to Calculate a Geometry Area Around a Coordinate

GIS_SERV_AREA_XY(longitude, latitude, distance, travel_mode[, wkid])

where:

**longitude**
Alphanumeric
Is the longitude of the starting point.

**latitude**
Alphanumeric
Is the latitude of the starting point.

**distance**
Integer
Is the travel limitation in either time or distance units.
travel_mode
Alphanumeric

Is a valid travel mode as defined in gis_serv_area.mas in the Catalog directory under the server installation directory. The accepted travel modes are;

- 'Miles'. This is the default value.
- 'TravelTime'.
- 'TruckTravelTime'.
- 'WalkTime'.
- 'Kilometers'.

wkid
Alphanumeric

Is the spatial reference ID for the coordinate. WKID is an abbreviation for Well-Known ID, which identifies a projected or geographic coordinate system. The default value is '4326', which represents decimal degrees.

**Example:** Calculating a Service Area Around a Coordinate

The following request calculates the geometry area that is a five-minute walk around a station, using the longitude and latitude that specify the station location.

```
DEFINE FILE esri/esri-citibike
DISTANCE/I4=5;
WKID/A10='4326';
TRAVEL_MODE/A10='WalkTime';
STATION_SERVICE_AREA/TX80 (GEOGRAPHIC_ROLE=GEOMETRY_AREA)=
   GIS_SERV_AREA_XY(START_STATION_LONGITUDE, START_STATION_LATITUDE,
   DISTANCE, TRAVEL_MODE, WKID);
END
TABLE FILE esri/esri-citibike
PRINT
   START_STATION_ID AS 'Station ID'
   START_STATION_NAME AS 'Station Name'
   STATION_SERVICE_AREA
   AS '5-Minute Walk Service Area Around Station Coordinate'
WHERE START_STATION_ID EQ 479 OR 512;
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
   TYPE=REPORT, GRID=OFF, SIZE=12,$
ENDSTYLE
END
```
Example:  Charting a Geometry Service Area Around a Coordinate

The following request generates service areas that are 5-minute walking distances from start station coordinates and charts them on an Esri map.

```
DEFINE FILE esri-citibike
WKID/A10='4326';
DISTANCE/A10='5';
TRAVEL_MODE/A10='WalkTime';
STATION_SERVICE_AREA/TX80 (GEOGRAPHIC_ROLE=GEOMETRY_AREA)=
  GIS_SERV_AREA_XY(START_STATION_LONGITUDE, START_STATION_LATITUDE, DISTANCE, TRAVEL_MODE, WKID);
END
```
GRAPH FILE ESRI-CITIBIKE
PRINT
    START_STATION_NAME
END_STATION_NAME
DISTANCE
ON TABLE PCHOLD FORMAT JSCHART
ON TABLE SET LOOKGRAPH CHOROPLETH
ON TABLE SET EMBEDHEADING ON
ON TABLE SET AUTOFIT ON
ON TABLE SET STYLE *
    TYPE=REPORT, TITLETEXT='Map', PAGESIZE=E, CHART-LOOK=com.esri.map, $
    TYPE=DATA, COLUMN=N1, /*START_STATION_NAME*/
        BUCKET=tooltip, $
    TYPE=DATA, COLUMN=N2, /*END_STATION_NAME*/
        BUCKET=tooltip, $
    TYPE=DATA, COLUMN=N3, /*DISTANCE*/
        BUCKET=tooltip, $
*GRAPH_JS_FINAL
    "legend": {"visible": true},
    "extensions" : { "com.esri.map" :
        { "scalebar" :
            { "scalebarUnit": "dual",
                "attachTo" : "bottom-left"
            },
            "baseMapInfo": {
                "drawBasemapControl" : false,
                "showArcGISBasemaps" : false,
                "customBaseMaps" : [
                    {"ibiBaseLayer" : "dark-gray"}
                ]
            },
            "overlayLayers": [
                {"ibiDataLayer": {"map-geometry" : {"map_by_field" : "STATION_SERVICE_AREA"}}, "title" : "Chart"}]
        },
        "introAnimation": "{\"enabled\":false}"
    }
*END
ENDSTYLE
HEADING
    "Chart Geometry Service Area"
END
GIS_SERV_AREA_XY: Calculating a Service Area Around a Given Coordinate

The output is shown in the following image.
SQL character functions manipulate alphanumeric fields and character strings.

In this chapter:
- CHAR_LENGTH: Finding the Length of a Character String
- CONCAT: Concatenating Two Character Strings
- DIGITS: Converting a Numeric Value to a Character String
- EDIT: Editing a Value According to a Format (SQL)
- LCASE: Converting a Character String to Lowercase
- LTRIM: Removing Leading Spaces
- POSITION: Finding the Position of a Substring
- RTRIM: Removing Trailing Spaces
- SUBSTR: Extracting a Substring From a String Value (SQL)
- TRIM: Removing Leading or Trailing Characters (SQL)
- UCASE: Converting a Character String to Uppercase
- VARGRAPHIC: Converting to Double-byte Character Data

CHAR_LENGTH: Finding 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 (variable length character). For example, if a column described as GLOSS VARCHAR(10) contains

'bryllig'
'slythy'
'toves'

then CHAR_LENGTH(GLOSS) would return
If the column is described as CHAR (non-variable length character), the same number is returned for all rows. In this case, CHAR_LENGTH(GLOSS) would return 10.

To avoid counting trailing blanks use CHAR_LENGTH(TRIM (TRAILING FROM GLOSS)). See TRIM: Removing Leading or Trailing Characters (SQL) on page 553 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**

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**

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 for DB2, ORACLE, and MS SQL Server. It does not work for flat file sources.

**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,

\[
\text{DIGITS}(-444.321)
\]

returns 0000444321.

EDIT: Editing a Value According to a Format (SQL)

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

\[
\text{EDIT}(\text{arg}, \text{mask})
\]

where:

\text{arg}

Numeric or character string

Is the value to be edited.

\text{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,

\[
\text{EDIT}('FRED', '9$$')
\]

returns F.

This example,

\[
\text{EDIT}('123456789', '999-99-9999')
\]

returns 123-45-6789.
LCASE: Converting a Character String to Lowercase

The LCASE function converts a character string value to lowercase. That is, capital letters are replaced by their corresponding lowercase values.

LOWER and LOWERCASE are identical to LCASE.

Syntax: How to Convert a Character String to Lowercase

\[
\text{LCASE}(\text{arg})
\]

where:

\text{arg}

character string

Is the value to be converted to lowercase.

This function returns a varying character string. The length is the same as the input argument.

Example: Converting a Character String to Lowercase

LCASE converts a character string to lowercase. This example,

\[
\text{LCASE}(\text{‘XYZ’})
\]

returns xyz.

LTRIM: Removing Leading Spaces

The LTRIM function removes leading spaces from a character string.

Syntax: How to Remove Leading Spaces

\[
\text{LTRIM}(\text{arg})
\]

where:

\text{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

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**

The RTRIM function removes trailing spaces from a character string.

**Syntax:**

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)**

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)

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 '.
UCASE: Converting a Character String to Uppercase

This example,

\[
\text{TRIM(TRAILING FROM ' ABC ')} \\
\text{TRIM(BOTH 'X' FROM 'XXYYXXX')} = ('YYY') \\
\]

returns 'ABC'

This example,

\[
\text{TRIM(BOTH 'X' FROM 'XXYYXXX')} \\
\]

returns YYY.

UCASE: Converting 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

\[
\text{UCASE} (\text{arg}) \\
\]

where:

\[
\text{arg} \\
\]

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, \[
\text{UCASE ('abc')} \\
\]

returns ABC.

VARGRAPHIC: Converting to Double-byte Character Data

The VARGRAPHIC function converts the input value to double-byte character data

Syntax:

How to Convert to the Double-byte Character Format

\[
\text{VARGRAPHIC arg} \\
\]
where:

\[ \text{arg} \]

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.
SQL Date and Time Functions

In this chapter:

- CURRENT_DATE: Obtaining the Date
- CURRENT_TIME: Obtaining the Time
- CURRENT_TIMESTAMP: Obtaining the Timestamp (Date/Time)
- DAY: Obtaining the Day of the Month From a Date/Timestamp
- DAYS: Obtaining the Number of Days Since January 1, 0001
- EXTRACT: Obtaining a Datetime Field From Date/Time/Timestamp
- HOUR: Obtaining the Hour From Time/Timestamp
- MICROSECOND: Obtaining Microseconds From Time/Timestamp
- MILLISECOND: Obtaining Milliseconds From Time/Timestamp
- MINUTE: Obtaining the Minute From Time/Timestamp
- MONTH: Obtaining the Month From Date/Timestamp
- SECOND: Obtaining the Second Field From Time/Timestamp
- QUARTER: Returning the Quarter of the Year
- WEEKDAY: Returning the Day of the Week
- YEAR: Obtaining the Year From a Date or Timestamp

CURRENT_DATE: Obtaining the Date

The CURRENT_DATE function returns the current date of the operating system 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

The CURRENT_TIME function returns the current time of the operating system 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)

The CURRENT_TIMESTAMP function returns the current timestamp of the operating system (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**
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, 0001

The DAYS function returns 1 more than the number of days from January 1, 0001 to the provided date value.

Syntax: How to Obtain the Number of Days Since January 1, 1900

\[ \text{DAYS}(\text{arg}) \]

where:

\( \text{arg} \)

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 one more than the number of days since January 1, 1900.

\[ \text{DAYS('2000-01-01')} \]

returns 730120.

EXTRACT: Obtaining a Datetime Field From Date/Time/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

\[ \text{EXTRACT}(\text{field FROM arg}) \]

where:

\( \text{arg} \)

Date, time, or timestamp

Is the input argument.

\( \text{field} \)

Is the datetime field of interest. Possible values are YEAR, QUARTER, MONTH, DAY, WEEKDAY, HOUR, MINUTE, SECOND, MILLISECOND and MICROSECOND.

This function returns an integer value.
**Note:**

- YEAR, QUARTER, MONTH, DAY, and WEEKDAY 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**

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

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

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

```sql
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,

```sql
MILLISECOND('11:22:33.456')
```

returns 456.

This example,

```sql
MILLISECOND('2001-01-22 10:11:12')
```

returns 0.

MINUTE: Obtaining the Minute From Time/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

```sql
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

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

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

\[
\text{SECOND}(\text{arg})
\]

where:

\[
\text{arg}
\]

- 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,

\[
\text{SECOND('11:22:33')}
\]

returns 33.

This example,

\[
\text{SECOND('2001-01-22 12:24:36')}
\]

returns 36.

### QUARTER: Returning the Quarter of the Year

Given a date or date-time value, QUARTER returns an integer (from 1 to 4) that represents the quarter within which that date falls.

### Syntax: How to Return the Quarter of the Year

\[
\text{QUARTER}(\text{arg})
\]

where:

\[
\text{arg}
\]

- Date or date-time
- Is the input date or date-time value.

### Example: Returning the Quarter of the Year

QUARTER returns the quarter of the year for each date of birth:

\[
\text{QUARTER(DATE_OF_BIRTH)}
\]
For 1993/03/27, the result is 1.

**WEEKDAY: Returning the Day of the Week**

Given a date or date-time value, WEEKDAY returns an integer from 1 (Monday) to 7 (Sunday) representing the day of the week for that date.

**Syntax:**

```
WEEKDAY(arg)
```

where:

- **arg**
  - Date or date-time
  - Is the input date or date-time value.

**Example:**

**Returning the Day of the Week**

WEEKDAY returns the day of the week for each birth date, where 1 represents Monday and 7 represents Sunday:

```
WEEKDAY(DATE_OF_BIRTH)
```

For 1993/03/27, the result is 6 (Saturday).

**YEAR: Obtaining the Year From a Date or Timestamp**

The YEAR function returns the year field from a date or timestamp value.

**Syntax:**

```
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,

\[
\text{YEAR('1976-07-04')}
\]

returns 1976.

This example,

\[
\text{YEAR('2001-01-22 10:00:00')}
\]

SQL data type conversion functions convert fields from one data type to another.

**In this chapter:**

- CAST: Converting to a Specific Data Type
- CHAR: Converting to a Character String
- CHAR: Converting to a Standard Date-Time Format
- DATE: Converting to a Date
- DECIMAL: Converting to Decimal Format
- FLOAT: Converting to Floating Point Format
- INT: Converting to an Integer
- SMALLINT: Converting to a Small Integer
- TIME: Converting to a Time
- TIMESTAMP: Converting to a Timestamp

### CAST: Converting 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(expression AS data_type[(length)])
```

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.
**length**

Is an optional parameter of character data types.

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**

There are two versions of the CHAR function, one for converting an argument to a character string, and one for converting a date, time, or timestamp value to a standard format. The version that takes one argument converts its argument to a character string. For information about using CHAR to convert a date, time, or timestamp value to a standard format, see *CHAR: Converting to a Standard Date-Time Format* on page 571.

**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.
CHAR: Converting to a Standard Date-Time Format

There are two versions of the CHAR function, one for converting an argument to a character string, and one for converting a date, time, or timestamp value to a standard format. The version that takes two arguments converts a date, time, or timestamp value to one of the standard date-time formats. For information about using CHAR to convert a single argument to a character string, see CHAR: Converting to a Character String on page 570.

Syntax: How to Convert a Date, Time, or Timestamp Value to a Standard Format

\[
\text{CHAR} (\text{datetime}, \text{fmt})
\]

where:

\text{datetime}

Date

Is the date, time, or timestamp value to be converted.

\text{fmt}

Can be one of the following formats:

<table>
<thead>
<tr>
<th>Name of Standard</th>
<th>Date Format</th>
<th>Time Format</th>
<th>Timestamp Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO</td>
<td>yyyy-mm-dd</td>
<td>hh.mm.ss</td>
<td>yyyy-mm-dd hh:mm:ss.xxxxxx</td>
</tr>
<tr>
<td>USA</td>
<td>mm/dd/yyyy</td>
<td>hh.mm AM/PM</td>
<td>yyyy-mm-dd-hh.mm.ss.xxxxxx</td>
</tr>
<tr>
<td>EUR</td>
<td>dd.mm.yyyy</td>
<td>hh.mm.ss</td>
<td>yyyy-mm-dd-hh.mm.ss.xxxxxx</td>
</tr>
<tr>
<td>JIS</td>
<td>yyyy-mm-dd</td>
<td>hh:mm:ss</td>
<td>yyyy-mm-dd-hh.mm.ss.xxxxxx</td>
</tr>
</tbody>
</table>

This function returns a character string whose length is of sufficient size to hold the value.

Example: Converting Date and Time Values to Standard Formats

CHAR converts a date, time, or timestamp value to a standard format. The following examples use the constants CURRENT DATE, CURRENT TIME, and CURRENT TIMESTAMP. Assume the current date is November 17, 2011:

\[
\text{CHAR (CURRENT DATE, USA) returns 11/17/2011}
\]

\[
\text{CHAR (CURRENT DATE, ISO) returns 2011-11-17}
\]

\[
\text{CHAR (CURRENT TIME, USA) returns 03:45 PM}
\]
DATE: Converting 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 value is returned.

Syntax: How to Convert to a Date

DATE(arg)

where:

arg

can be a 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

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

The FLOAT function converts a number to floating-point format.

### Syntax: How to Convert to the Floating Point Format

FLOAT(arg)

Where:

- **arg**
  - Numeric
**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**

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**

The SMALLINT function converts a number to a small integer. Generally, a small integer occupies only two 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,

\[ \text{SMALLINT}(3.5) \]

returns 3.

**TIME: Converting 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 value is returned.

**Syntax:** How to Convert to a Time

\[ \text{TIME}(\arg) \]

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,

\[ \text{TIME('2004-03-15 01:02:03.444')} \]
returns 010203444.

**TIMESTAMP: Converting 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**

```sql
TIMESTAMP(arg)
```

where:

```sql
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,

```sql
TIMESTAMP('2004-06-24')
```

returns 20040624000000.

This example,

```sql
TIMESTAMP('11:22:33')
```

returns 20010101112233, if the current date is January 1, 2001.
SQL numeric functions perform calculations on numeric constants and fields.

In this chapter:

- **ABS**: Returning an Absolute Value (SQL)
- **CEIL**: Returning the Smallest Integer Greater Than or Equal to a Value
- **FLOOR**: Returning the Largest Integer Less Than or Equal to a Value (SQL)
- **GREATEST**: Returning the Largest Value
- **LEAST**: Returning the Smallest Value
- **LOG**: Returning a Logarithm (SQL)
- **EXP**: Returning e Raised to a Power
- **MOD**: Returning the Remainder of a Division
- **POWER**: Raising a Value to a Power (SQL)
- **SQRT**: Returning a Square Root (SQL)

### ABS: Returning an Absolute Value (SQL)

The ABS function returns the absolute value of a number.

**Syntax:** How to Return an Absolute Value

\[ \text{ABS}(\text{arg}) \]

where:

\[ \text{arg} \]

Numeric

Is the input value.

This function returns the value as the same data type 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,

\[ \text{ABS}(-5.5) \]

returns 5.5.

**CEIL: Returning the Smallest Integer Greater Than or Equal to a Value**

CEIL returns the smallest integer value not less than the argument. CEILING is a synonym for CEIL.

**Syntax:** How to Return the Smallest Integer Greater Than or Equal to a Value

\[ \text{CEIL}(n) \]

where:

- \( n \) Numeric or Alphanumeric
  
  Is the value less than or equal to the returned integer. For exact-value numeric arguments, the return value has an exact-value numeric type. For alphanumeric or floating-point arguments, the return value has a floating-point type.

**Example:** Returning an Integer Greater Than or Equal to a Value

CEIL returns an integer greater than or equal to the argument.

\[ \text{CEIL}(N) \]

For \( N=1.23 \), the result is 2.

For \( N=-1.23 \), the result is -1.

**FLOOR: Returning the Largest Integer Less Than or Equal to a Value (SQL)**

FLOOR returns the largest integer value not greater than a value.

**Syntax:** How to Return the Largest Integer Less Than or Equal to a Value

\[ \text{FLOOR}(n) \]
where:

\( n \)

Numeric or Alphanumeric

Is the value greater than or equal to the returned integer. For exact-value numeric arguments, the return value has an exact-value numeric type. For alphanumeric or floating-point arguments, the return value has a floating-point type.

**Example: Returning an Integer Less Than or Equal to a Value**

FLOOR returns an integer less than or equal to the argument.

\( \text{FLOOR}(N) \)

For \( N=1.23 \), the result is 1.

For \( N=-1.23 \), the result is -2.

**GREATEST: Returning the Largest Value**

With two or more arguments, GREATEST returns the largest (maximum-valued) argument. The arguments are compared using the following rules:

- If any argument is NULL, the result is NULL. No comparison is needed.
- If the return value is used in an INTEGER context, or all arguments are integer-valued, they are compared as integers.
- If the return value is used in a floating-point context, or all arguments are floating-point-valued, they are compared as floating-point values.
- If the arguments comprise a mix of numbers and strings, they are compared as numbers.
- If any argument is a character string, the arguments are compared as character strings. In all other cases, the arguments are compared as binary strings.

**Syntax:** How to Return the Largest Value

\[
\text{GREATEST}(\text{value1, value2, ... , valuenn})
\]

where:

\[
\text{value1, value2, ... , valuenn}
\]

Numeric or alphanumeric

Are the values to be compared.
Example: Returning the Largest Value

GREATEST returns the largest argument.

\[ \text{GREATEST}(X, Y, Z) \]

For \( X=2, \ Y=0, \) and \( Z=-1 \), the result is 2.

For \( X='B', \ Y='A', \) and \( Z='C' \), the result is 'C'.

LEAST: Returning the Smallest Value

With two or more arguments, LEAST returns the smallest (minimum-valued) argument. The arguments are compared using the following rules:

- If any argument is NULL, the result is NULL. No comparison is needed.
- If the return value is used in an INTEGER context, or all arguments are integer-valued, they are compared as integers.
- If the return value is used in a floating-point context, or all arguments are floating-point-valued, they are compared as floating-point values.
- If the arguments comprise a mix of numbers and strings, they are compared as numbers.
- If any argument is a character string, the arguments are compared as character strings. In all other cases, the arguments are compared as binary strings.

Syntax: How to Return the Smallest Value

\[ \text{LEAST}(\text{value}_1, \text{value}_2, \ldots, \text{value}_n) \]

where:

\[ \text{value}_1, \text{value}_2, \ldots, \text{value}_n \]

Numeric or alphanumeric

Are the values to be compared.

Example: Returning the Smallest Value

LEAST returns the smallest argument.

\[ \text{LEAST}(X, Y, Z) \]

For \( X=2, \ Y=0, \) and \( Z=-1 \), the result is -1.

For \( X='B', \ Y='A', \) and \( Z='C' \), the result is 'A'.
LOG: Returning a Logarithm (SQL)

The LOG function returns the natural logarithm of the input value.

Syntax: How to Return a Logarithm

\[
\text{LOG}(\text{arg})
\]

where:

\[
\text{arg}
\]

Numeric

Is the input value.

This function returns double precision numbers with three decimal places.

Example: Returning a Logarithm

LOG returns the natural logarithm of a value. This example,

\[
\text{LOG}(4)
\]

returns 1.386.

EXP: Returning e Raised to a Power

The EXP function returns the mathematical constant e raised to a power.

Syntax: How to Return e Raised to a Power

\[
\text{EXP}(\text{arg})
\]

where:

\[
\text{arg}
\]

Numeric

Is the value of the power to which to raise the mathematical constant e.

Example: Returning e Raised to a Power

EXP returns the mathematical constant e to a power. This example,

\[
\text{EXP}(4)
\]

returns 54.598.
MOD: Returning the Remainder of a Division

The SQL function MOD returns the remainder of the first argument divided by the second argument.

Syntax: How to Return the Remainder of a Division

\[ \text{MOD} \left( n, m \right) \]

where:

\[ n \]
Numeric
Is the dividend (number to be divided).

\[ m \]
Numeric
Is the divisor (number to divide by). If the divisor is zero (0), MOD returns NULL.

Example: Returning the Remainder of a Division

MOD returns the remainder of \( n \) divided by \( m \).

\[ \text{MOD} \left( N, M \right) \]

For \( N=16 \) and \( M=5 \), the result is 1.
For \( N=34.5 \) and \( M=3 \), the result is 1.5.

POWER: Raising a Value to a Power (SQL)

The POWER function returns the value calculated by raising the first argument to the power specified by the second argument.

Syntax: How to Return a Value Raised to a Power

\[ \text{POWER} \left( \text{arg1}, \text{arg2} \right) \]

where:

\[ \text{arg1} \]
Numeric
Is the value to be raised to the power specified by \( \text{arg2} \).
\[ arg2 \]

Numeric

Is the value of the power to which to raise \( arg1 \).

**Example:** Returning a Value Raised to a Power

POWER returns the value calculated by raising the first argument to the value specified by the second argument. This example,

\[ \text{EXP}(2, 4) \]

returns 16.000.

**SQRT: Returning a Square Root (SQL)**

The SQRT function returns the square root of the input value.

**Syntax:** How to Return a Square Root

\[ \text{sqrt}(\text{arg}) \]

where:

\[ \text{arg} \]

Numeric

Is the input value.

This function returns double precision numbers with three decimal places.

**Example:** Returning a Square Root

SQRT returns the square root of a value. This example,

\[ \text{SQRT}(4) \]

returns 2.000.
The SQL functions described in this chapter perform a variety of conversions, tests, and manipulations.

In this chapter:
- COUNTBY: Incrementing Column Values Row by Row
- DB_EXPR: Inserting an SQL Expression Into a Request (SQL)
- 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

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, ...

**DB_EXPR: Inserting an SQL Expression Into a Request (SQL)**

The DB_EXPR function inserts a native SQL expression exactly as entered into the native SQL generated for a FOCUS or SQL language request.

The DB_EXPR function can be used in a DEFINE command, a DEFINE in a Master File, a WHERE clause, a FILTER FILE command, a filter in a Master File, or in an SQL statement. It can be used in a COMPUTE command if the request is an aggregate request (uses the SUM, WRITE, or ADD command) and has a single display command. The expression must return a single value.

**Syntax:** How to Insert an SQL Expression Into a Request With DB_EXPR

```sql
DB_EXPR(native_SQL_expression)
```

where:

- **native_SQL_expression**

  Is a partial native SQL string that is valid to insert into the SQL generated by the request. The SQL string must have double quotation marks (" ) around each field reference, unless the function is used in a DEFINE with a WITH phrase.

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

- The expression must return a single value.
- Any request that includes one or more DB_EXPR functions must be for a synonym that has a relational SUFFIX.
- Field references in the native SQL expression must be within the current synonym context.
- The native SQL expression must be coded inline. SQL read from a file is not supported.
**Example:** Inserting the DB2 BIGINT and CHAR Functions Into a TABLE Request

The following TABLE request against the WF_RETAIL data source uses the DB_EXPR function in the COMPUTE command to call two DB2 functions. It calls the BIGINT function to convert the squared revenue to a BIGINT data type, and then uses the CHAR function to convert that value to alphanumeric.

```sql
TABLE FILE WF_RETAIL
SUM REVENUE NOPRINT
AND COMPUTE BIGREV/A31 = DB_EXPR(CHAR(BIGINT("REVENUE" * "REVENUE") ) ) ;
AS 'Alpha Square Revenue'
BY REGION
ON TABLE SET PAGE NOPAGE
END
```

WF_RETAIL is a sample data source you can create by right-clicking an application on the Reporting Server Web Console and selecting New and then Samples from the context menu.

The trace shows that the expression from the DB_EXPR function was inserted into the DB2 SELECT statement:

```sql
SELECT
T11."REGION",
SUM(T1."Revenue"),
((CHAR(BIGINT( SUM(T1."Revenue") * SUM(T1."Revenue") ) ) ))
FROM
wrd_fact_sales T1,
wrd_dim_customer T5,
wrd_dim_geography T11
WHERE
(T5."ID_CUSTOMER" = T1."ID_CUSTOMER") AND
(T11."ID_GEOGRAPHY" = T5."ID_GEOGRAPHY")
GROUP BY
T11."REGION"
ORDER BY
T11."REGION"
FOR FETCH ONLY;
END
```
The output is:

<table>
<thead>
<tr>
<th>Region</th>
<th>Alpha Square Revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Central</td>
<td>459024717717929</td>
</tr>
<tr>
<td>MidEast</td>
<td>61720506151994</td>
</tr>
<tr>
<td>NorthEast</td>
<td>247772056471221</td>
</tr>
<tr>
<td>NorthWest</td>
<td>42335175855351</td>
</tr>
<tr>
<td>SouthEast</td>
<td>205820846242532</td>
</tr>
<tr>
<td>SouthWest</td>
<td>9449541537794</td>
</tr>
<tr>
<td>West</td>
<td>164356565757257</td>
</tr>
</tbody>
</table>

**HEX: Converting 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(character)
```

where:

`character`

Is the input value.

This function returns an alphanumeric value.

**Example:** Converting a Value to Hex

This example,

```
HEX('n')
```

returns 6E.

**IF: Testing 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

\[
\text{IF}(\text{test}, \text{val1}, \text{val2})
\]

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.

\[
\text{IF(COUNTRY = 'ENGLAND', 'LONDON', 'PARIS') =}
\]

- '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.

\[
\text{IF(COUNTRY = 'ENGLAND', 'LONDON',}
\]

- \[
\text{IF(COUNTRY = 'FRANCE', 'PARIS', 'ROME')) =}
\]

- 'LONDON' if COUNTRY is 'ENGLAND'
- 'PARIS' if COUNTRY = 'FRANCE'
- 'ROME' otherwise.

**LENGTH:** Obtaining the Physical Length of a Data Item

The LENGTH function returns the actual length in memory of a data item.

**Syntax:** How to Obtain the Physical Length of a Data Item

\[
\text{LENGTH}(\text{arg})
\]

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,

```sql
LENGTH('abcdef')
```

returns 6.

This example,

```sql
LENGTH(3)
```

returns 4.

**VALUE: Coalescing Data Values**

**Note:** The SQL function VALUE is not supported. Instead, use the SQL operator COALESCE. For more information see [COALESCE: Coalescing Data Values](#) on page 593.
SQL Operators

SQL operators are used to evaluate expressions.

In this chapter:

- CASE: SQL Case Operator
- COALESCE: Coalescing Data Values
- NULLIF: NULLIF Operator

CASE: 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 \textit{test-expr} is compared to \textit{value-expr-1}, ..., \textit{value-expr-n} in turn:

- If any of these match, the value of the result is the corresponding \textit{result-expr}.
- If there are no matches and the ELSE clause is present, the result is \textit{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 \textit{cond-1}, ..., \textit{cond-n} are evaluated in turn.

- If any of these are true, the value of the result is the corresponding \textit{result-expr}.
- If no conditions are true and the ELSE clause is present, the result is \textit{else-expr}.
- If no conditions are true and the ELSE clause is not present, the result is NULL.

Format 1

\begin{verbatim}
CASE test-expr
  WHEN value-expr-1 THEN result-expr-1
  . . .
  WHEN value-expr-n THEN result-expr-n
  \[ ELSE else-expr \]
END
\end{verbatim}
**Format 2**

```sql
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,

```sql
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

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

```sql
COALESCE(arg1, arg2, [ ... argn ])
```

where:

```sql
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,

```sql
COALESCE('A', 'B')
```

return A.

This example,

```sql
COALESCE(NULL, 'B')
```

return B.

This example,

```sql
COALESCE(NULL, NULL)
```
return NULL.

**NULLIF: NULLIF Operator**

The NULLIF operator returns NULL if its two arguments are equal. Otherwise, the first argument is returned.

**Syntax:**

`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

ABS function 448, 577
alphanumeric strings 410
    converting 410
analytic functions 31
    INCREASE 63
    PCT_INCREASE 67
    PREVIOUS 70
    RUNNING_AVE 72
    RUNNING_MAX 75
    RUNNING_MIN 78
    RUNNING_SUM 81
ARGLEN function 122, 123, 185, 188
ASIS function 123, 124
assigning date-time values 358
    COMPUTE command 358, 360
    DEFINE command 358
    IF criteria 358, 361
    WHERE criteria 358, 360
ATODBL function 410
AYMD function 331

B

bit strings 127, 128
BITSON function 125, 126
BITVAL function 127, 128
BUSDAYS parameter 280

business days 280
    BUSDAYS parameter 280
BYTVAL function 129, 131

C

CASE operator 591
CAST function 569
CEILING function 439
CHAR function 395, 570, 571
CHAR_LENGTH function 86, 545
character functions, simplified 85
    CHAR_LENGTH 86
    CONCAT 87
    DIGITS 90
    LAST_NONBLANK 95
    LOWER 96
    LPAD 98
    LTRIM 99
    PATTERNS 101
    POSITION 102
    REGEX 104
    REPLACE 106
    RPAD 108
    RTRIM 110
    SPLIT 111
    SUBSTRING 112
    TOKEN 114
    TRIM 116
    UPPER 118
character functions

ARGLEN 122, 123, 185, 188
ASIS 123, 124
BITSON 125, 126
BITVAL 127, 128
BYTVAL 129, 131
CHKFMT 131–133, 135, 136
CHKNUM 135
CTRAN 137–139
CTRFLD 143, 144
DCTRAN 203
DSTRIP 206, 207
EDIT 144, 145
GETTOK 146, 147
LCWORD 148–150
LCWORD2 149, 150, 152
LCWORD3 151
LJUST 152
LOCASE 153, 154
OVRLAY 155, 156
PARAG 158–160
POSIT 163, 164
RJUST 166, 167
SOUNDEX 167–169
SPELLNM 169, 170
SQL 545
SQUEEZ 171, 172
STRIP 172–174, 176
SUBSTR 176–178, 195
TRIM 178–180

character functions

TRIMV 197
UPCASE 181, 182
variable length 189
XMLDECOD 184
XMLENCOD 186

character strings 122, 152
bits 125, 127
centering 143, 144
comparing 167
converting case 153, 181
Dialogue Manager 123
dividing 158
extracting characters 144
extracting substrings 146, 176, 178, 195
finding substrings 163
format 131
justifying 152, 166
measuring length 122
overlaying 155
reducing spaces 171
removing occurrences 178
right-justifying 166
spelling out numbers 169
translating characters 129, 137

CHECKMD5 function 220
CHECKSUM function 221
CHGDAT function 333, 334
CHKFMT function 131–133, 135, 136
CHKNUM function 135
CHKPCK function 449
CLSDREC 493, 500
COALESCE function 222
COALESCE operator 593
COMPACTFORMAT function 396
components 355
COMPUTE command
assigning date-time values 360
CONCAT function 87, 546
conversion functions, simplified 395
CHAR 395
CTRLCHAR 398
HETYPE 402
TO_INTEGER 407
TO_NUMBER 408
conversion functions, simplified
PHONETIC 405
converting formats 410
COUNTBY function 585
cross-referenced data sources 252
CTRAN function 137–139
CTRFLD function 143, 144
CTRLCHAR function 398
CURRENT_DATE function 557
CURRENT_TIME function 558
CURRENT_TIMESTAMP function 558

D

DA functions 336
DADDMY function 336
DADMY function 336
DAMDY function 336
DAMYD function 336
data source functions 219
FIND 240–244
LAST 250–252
LOOKUP 252–257
data sources 219
cross-referenced 252, 257
decoding values 236
retrieving values 250–252, 257
values 219
verifying values 240–245
data type conversion functions 569
date and time functions 278, 350
arguments and 355
AYMD 331
CHGDAT 333, 334
DA 336
DADDMY 336
DADYM 336
DAMDY 336
DAMYD 336
DATEADD 287
DATECVT 291
DATEDIF 293
DATEDIF 293
DATEMOV 297
DATETRAN 304
DAYDM 336
DAYMD 336, 337
date and time functions 278, 350
  DOWK 339
  DOWKL 339
  DTDMY 341
  DTDYM 341
  DTMDY 341
  DTMYD 341
  DTYMD 341
  GREGDT 342
  HADD 361, 363
  HCNVRT 364–366
  HDATE 366, 367
  HDIFF 367–369
  HDTTM 369–371
  HGETC 371–374
  HGETZ 373
  HHMMSS 375, 376
  HHMS 376
  HINPUT 378, 379
  HMIDNT 379–381
  HNAME 381–383
  HPART 383, 384
  HSETPT 385, 386
  HTIME 387, 388
  JULDAT 344
  legacy 329
  SQL 557
  standard 279
  TIMETOTS 388, 389

date and time functions 278, 350
  TODAY 328
  YM 346
  YMD 339
date formats 355
  formatted-string format 356, 357
  international 304
  numeric string format 356
  translated-string format 357
DATE function 572
date functions, simplified 261
  DT_CURRENT_DATE 262
  DT_CURRENT_DATETIME 262
  DT_CURRENT_TIME 263
  DTPART 271
  DTRUNC 273
date functions
  work days 280
date-time format
  ISO standard input values 358
date-time values
  adding 331
  assigning 358
  converting 387, 388
  converting formats 333, 336, 341, 344, 364, 366, 369
  elapsed time 346
  finding day of week 339
  finding difference 293, 338, 367
  incrementing 361
date-time values
  moving dates 297
  retrieving components 383
  retrieving time 375, 376
  returning dates 328
  setting time 379
  storing 371, 373
  subtracting 331
DATEADD function 287
DATECVT function 291
DATEDIF function 293
DATEFORMAT parameter 351
DATEMOV function 297
DATETRAN function 304, 312
DAY function 559
DAYDM function 336
DAYMD function 336, 337
DAYS function 560
DB_EXPR function 225, 586
DB_LOOKUP function 233
    COMPUTE command 233
    DEFINE 233
    MODIFY 233
    TABLE COMPUTE 233
DCTRAN function 203
DECIMAL function 572
DECODE function 236–239
  decoding functions 219, 236–239
  decoding values 236
    in a function 236–238
DEDIT function 204
DIGITS function 90, 547
DMOD function 451–453
DMY function 338
double exponential smoothing 40
  FORECAST_DOUBLEEXP 40
double-byte characters 203, 206
DOWK function 339
DOWKL function 339
DSTRIP function 206, 207
DSUBSTR function 207
DSUBSTR function 207
DT_CURRENT_DATE function 262
DT_CURRENT_DATETIME function 262
DT_CURRENT_TIME function 263
DTADD function 264
DTDIFF function 267
TDMY function 341
DTDYM function 341
DTMDY function 341
DTMYD function 341
DTPART function 271
DTRUNC function 273
DTSTRICT parameter 353, 354
DTYDM function 341
DTYMD function 341
EDIT function 144, 145, 413, 414, 548
ENCRYPT function 487
environment variables 495
    assigning values 496
    retrieving values 495
error messages 494
EXP function 453, 454, 581
EXPN function 454
exponential moving average 37
    FORECAST_EXPAVE 37
EXTRACT function 560

F

FEXERR function 494, 495
FGETENV function 495
FIND function 240–245
FIQTR function 322
FIYR function 320
FIYYQ function 325
FLOAT function 573
FLOOR function 442
FMOD function 451–453
FORECAST_DOUBLEXP
    double exponential smoothing 40
FORECAST_EXPAVE
    exponential moving average 37
FORECAST_LINEAR
    linear regression equation 47
FORECAST_MOVAVE
    simple moving average 31

FORECAST_SEASONAL
    triple exponential smoothing 42
format conversion functions
    ATODBL 410
    EDIT 413, 414
    FPRINT 414
    FTOA 420, 421
    HEXBYT 421–423
    ITONUM 425
    ITOPACK 426, 427
    ITOZ 428, 429
    PCKOUT 430, 431
    PTOA 431–433
    TSTOPACK 433
    UFMT 435, 436
format conversions 410
    packed numbers 430
    to alphanumeric 420, 431
    to characters 421
    to double-precision 425
    to hexadecimal 435
    to packed decimal 426
    to zoned format 428
formats 410
    alphanumeric 413, 414
    converting 410
formatted-string format 356, 357
FPRINT function 400, 414
FPUTENV function 496, 497
FTOA function 420, 421
function types
  data source 219
  decoding 219
  numeric 447
  system 493
functions 233
  analytic 31
  character 545
  data type conversion 569
  date and time 278, 329, 350, 557
  FIND 240–245
  FIQTR 322
  FIYR 320
  FIYYQ 325
  numeric 577
  SLEEP 503
  SQL 545, 557, 569, 577, 585
  STRREP 174
  variable length character 189

geography functions
  GIS_LINE 527
  GIS_POINT 531
  GIS_SERV_AREA_XY 540
  GIS_SERVICE_AREA 536
  GET_TOKEN function 92
  GETENV function 487
  GETTOK function 146, 147
  GETUSER function 497–499
  GIS_DISTANCE function 512
  GIS_DRIVE_ROUTE function 514
  GIS_GEOCODE_ADDR function 518
  GIS_GEOCODE_ADDR_CITY function 519
  GIS_GEOCODE_ADDR_POSTAL function 521
  GIS_GEOMETRY function 522
  GIS_IN_POLYGON function 526
  GIS_LINE function 527
  GIS_POINT function 531
  GIS_REVERSECOORDINATE function 534
  GIS_SERV_AREA_XY function 540
  GIS_SERVICE_AREA function 536
  GREGDT function 342, 345

G
  geography functions 507
  geography functions
    GIS_DISTANCE 512
    GIS_DRIVE_ROUTE 514
    GIS_GEOCODE_ADDR 518
    GIS_GEOCODE_ADDR_CITY 519
    GIS_GEOCODE_ADDR_POSTAL 521
    GIS_GEOMETRY 522
    GIS_IN_POLYGON 526

H
  HADD function 361, 363
  hash value 220, 221
  HCNVRT function 364–366
  HDATETIME function 366, 367
  HDIFF function 367–369
  HDTTM function 369–371
HEX function 588
HEXBYT function 421–423
HEXTYPE function 402
HGETC function 371–374
HGETZ function 373
HHMSS function 375, 376
HHMS function 376
HINPUT function 378, 379
HMIDNT function 379–381
HNAME function 381–383
holidays 280, 282
holiday files 280, 282
HOUR function 561
HPART function 383, 384
HSETPT function 385, 386
HTIME function 387, 388
HTMTOTS function 388
HYYWD function 390

ISO standard date-time formats 358
ITONUM function 425
ITOPACK function 426, 427
ITOZ function 428, 429

J
JOBNAME function 498
JULDAT function 344

K
KKFCUT function 213

L
lag values 59
LAST function 250–252
LAST_NONBLANK function 95
LCASE function 549
LCWORD function 148–150
LCWORD2 function 149, 150, 152
LCWORD3 function 151
LEADZERO parameter 286
legacy date functions
    DMY 338
    legacy dates 330
    legacy versions 330
    MDY 338
    YMD 338

LENGTH function 589
linear regression equation 47
FORECAST_LINEAR 47
LJUST function 152
LOCAS function
variable length 191
LOCASE function 153, 154
LOG function 456, 457, 581
LOOKUP function 252–257
extended function 257
LOWER function 96, 549
LOWERCASE function 549
LPAD function 98
LTRIM function 99, 549

M
Maintain data source functions 241, 242
MAX function 457
MD5 hash value 220
MDY function 338
MICROSECOND function 562
MILLISECOND function 563
MIN function 457, 458
MINUTE function 563
MODIFY data source functions 243–245
MONTH function 564

N
NORMSDST function 459–463
NORMSINV function 459, 461–463
NULLIF function 258
NULLIF operator 594
numbers 448
absolute value 448
calculating remainders 451
generating random 463, 466
greatest integer 455
logarithms 456
maximum 457
minimum 457
raising to a power 453
square root 468
standard normal deviation 459, 460, 462
validating packed fields 449
numeric functions 447, 577
ABS 448
CHKPCK 449
DMOD 451–453
EXP 453, 454
FMOD 451–453
IMOD 451–453
INT 455, 456
LOG 456, 457
MAX 457
MIN 457, 458
NORMSDST 459–463
NORMSINV 459, 461–463
PRDNOR 463–465
PRDUNI 463, 464
RDNORM 466, 467
RDUNIF 466, 467
numeric functions 447, 577
  SQRT 468
numeric string format 356
numeric values 447

O
  OUTLIER function 476
  OVRLAY function 155, 156

P
  packed numbers, writing to an output file 437
  PARAG function 158–160
  PARTITION_AGGR 51
  PARTITION_REF 59
  PATTERN function 160
  PATTERNS function 101
  PCKOUT function 430, 431
  PCT_INCREASE function 67
  PHONETIC function 405
  POSIT function 163, 164
  POSITION function 102, 550
  POWER function 582
  PRDNOR function 463–465
  PRDUNI function 463, 464
  PREVIOUS function 70
prior values 59
process IDs 498
  PTOA function 431–433
  PUTDDREC 500
  PUTENV function 488

Q
  QUARTER function 565

R
  RDNORM function 466, 467
  RDUNIF function 466, 467
  REGEX function 104
  REPLACE function 106
retrieving environment variable values 495
  REVERSE function 164
  RJUST function 166, 167
rolling calculations 51
  RPAD function 108
  RTRIM function 110, 551
  RUNNING_AVE function 72
  RUNNING_MAX function 75
  RUNNING_MIN function 78
  RUNNING_SUM function 81

S
  SECOND function 564
  SET parameters 280
    BUSDAYS 280
    DTSTRICT 353, 354
    HDAY 280, 282
    LEADZERO 286
  SFTDEL function 214
  SFTINS function 216
simple moving average 31
  FORECAST_MOVAVE 31
simplified character functions 85
simplified conversion functions 395
simplified date functions 261
simplified geography functions
  GIS_REVERSE_COORDINATE 534
simplified system functions 485
single-byte characters 203, 206
SLACK function 489
SLEEP function 503
SMALLINT function 574
SOUNDEX function 167–169
SPELLNM function 169, 170
SPLIT function 111
SQL functions 545, 557, 569, 577, 585
SQL operators 591
SQRT function 468, 583
SQUEEZ function 171, 172
standard date and time functions 279
standard normal deviation 459, 460, 462
statistical functions 469
string replacement 174, 175
STRIP function 172–174, 176
STRREP function 174, 175
SUBSTR function 176–178, 195, 551
  variable length 195
SUBSTRING function 112, 551
substrings 144
  extracting 144, 146, 176, 178, 195
  finding 163
  overlaying character strings 155
system functions 493
  FEXERR 494, 495
  FGETENV 495
  FPUTENV 496, 497
  GETUSER 497–499
  JOBNAME 498
  SYSVAR 504
SYSVAR function 504

T
time formats 357
TIME function 575
TIMESTAMP function 576
TIMETOTS function 388, 389
TO_INTEGER function 407
TO_NUMBER function 408
TODAY function 328
TOKEN function 114
TRIM function 178–180, 553
TRIM_ function 116
TRIMV function 197
triple exponential smoothing 42
  FORECAST_SEASONAL 42
TSTOPACK function 433

U
UCASE function 554
UFMT function 435, 436
UPCASE function 181, 182
UPPER function 118, 554
UPPERCASE function 554
user IDs 497

V
VALUE function 590
values 236
  decoding 236
  verifying 240–245
variable length character functions 189

W
WEEKDAY function 566
WEEKFIRST parameter 351
WHERE criteria 360
  assigning date-time values 360
work days 280
  holidays 280, 282

X
XMLDECOD function 184
XMLENCOD function 186
XTPACK function 437

Y
YEAR function 566
YM function 346
YMD function 338, 339
Feedback

Customer success is our top priority. Connect with us today!

Information Builders Technical Content Management team is comprised of many talented individuals who work together to design and deliver quality technical documentation products. Your feedback supports our ongoing efforts!

You can also preview new innovations to get an early look at new content products and services. Your participation helps us create great experiences for every customer.

To send us feedback or make a connection, contact Sarah Buccellato, Technical Editor, Technical Content Management at Sarah_Buccellato@ibi.com.

To request permission to repurpose copyrighted material, please contact Frances Gambino, Vice President, Technical Content Management at Frances_Gambino@ibi.com.