## Contents

**Preface** .................................................................................................................. 13
  - Conventions ........................................................................................................... 15
  - Related Publications ............................................................................................... 16
  - Customer Support .................................................................................................... 16
  - Information You Should Have .................................................................................. 17
  - User Feedback ........................................................................................................ 18
  - iWay Software Training and Professional Services ................................................ 18

**1. Functions Overview** ......................................................................................... 19
  - Function Arguments .............................................................................................. 19
  - Function Categories .............................................................................................. 20
  - Character Chart for ASCII and EBCDIC .............................................................. 20

**2. Simplified Analytic Functions** ........................................................................ 29
  - FORECAST_MOVAVE: Using a Simple Moving Average ...................................... 29
  - FORECAST_EXPAVE: Using Single Exponential Smoothing ............................... 35
  - FORECAST_DOUBLEXP: Using Double Exponential Smoothing ......................... 38
  - FORECAST_SEASONAL: Using Triple Exponential Smoothing ............................. 40
  - FORECAST_LINEAR: Using a Linear Regression Equation .................................. 44
  - PARTITION_AGGR: Creating Rolling Calculations ............................................. 48
  - PARTITION_REF: Using Prior Field Values in Calculations ................................. 58

**3. Simplified Character Functions** ....................................................................... 63
  - CHAR_LENGTH: Returning the Length in Characters of a String ......................... 64
  - CONCAT: Concatenating Strings After Removing Trailing Blanks From the First .... 64
  - DIGITS: Converting a Number to a Character String ........................................... 65
  - LAST_NONBLANK: Retrieving the Last Field Value That is Neither Blank nor Missing .... 66
  - LOWER: Returning a String With All Letters Lowercase .................................... 67
  - LPAD: Left-Padding a Character String ............................................................... 68
  - LTRIM: Removing Blanks From the Left End of a String ...................................... 69
  - PATTERNS: Returning a Pattern That Represents the Structure of the Input String ...... 70
  - POSITION: Returning the First Position of a Substring in a Source String ............. 70
  - REGEX: Matching a String to a Regular Expression ........................................... 71
  - REPLACE: Replacing a String ................................................................................ 72
4. Character Functions .......................................................... 81
    ARGLEN: Measuring the Length of a String ........................................... 82
    ASIS: Distinguishing Between Space and Zero ........................................... 82
    BITSON: Determining If a Bit Is On or Off ........................................... 84
    BITVAL: Evaluating a Bit String as an Integer ........................................ 85
    BYTVAL: Translating a Character to Decimal ........................................ 86
    CHKFMT: Checking the Format of a String ........................................ 87
    CHKNUM: Checking a String for Numeric Format ....................................... 89
    CTRAN: Translating One Character to Another ........................................ 89
    CTRFLD: Centering a Character String ................................................. 91
    EDIT: Extracting or Adding Characters ............................................. 92
    GETTOK: Extracting a Substring (Token) ........................................... 93
    LCWORD: Converting a String to Mixed-Case ........................................ 94
    LCWORD2: Converting a String to Mixed-Case ....................................... 95
    LCWORD3: Converting a String to Mixed-Case ....................................... 96
    LJUST: Left-Justifying a String ...................................................... 97
    LOCASE: Converting Text to Lowercase ............................................. 97
    OVRLAY: Overlaying a Character String ............................................. 98
    PARAG: Dividing Text Into Smaller Lines ........................................... 99
    PATTERN: Generating a Pattern From a String ...................................... 101
    POSIT: Finding the Beginning of a Substring ...................................... 102
    REVERSE: Reversing the Characters in a String .................................... 103
    RJUST: Right-Justifying a Character String ......................................... 104
    SOUNDEX: Comparing Character Strings Phonetically ............................. 104
    SPELLNM: Spelling Out a Dollar Amount ........................................... 105
SQUEEZ: Reducing Multiple Spaces to a Single Space ................................... 106
STRIP: Removing a Character From a String ............................................ 107
STRREP: Replacing Character Strings ................................................. 108
SUBSTR: Extracting a Substring ...................................................... 109
TRIM: Removing Leading and Trailing Occurrences ...................................... 111
UPCASE: Converting Text to Uppercase ................................................ 112
XMLDECOD: Decoding XML-Encoded Characters ........................................ 113
XMLENCOD: XML-Encoding Characters ................................................ 114

5. Variable Length Character Functions ......................................... 117
Overview .......................................................................... 117
LENV: Returning the Length of an Alphanumeric Field .................................... 118
LOCASV: Creating a Variable Length Lowercase String ................................... 118
POSITV: Finding the Beginning of a Variable Length Substring ............................. 119
SUBSTV: Extracting a Variable Length Substring ........................................ 121
TRIMV: Removing Characters From a String ............................................ 122
UPCASV: Creating a Variable Length Uppercase String ................................... 123

6. Character Functions for DBCS Code Pages .................................... 125
DCTRAN: Translating A Single-Byte or Double-Byte Character to Another .................... 125
DEDIT: Extracting or Adding Characters ................................................ 126
DSTRIP: Removing a Single-Byte or Double-Byte Character From a String ................... 128
DSUBSTR: Extracting a Substring .................................................................. 129
JPTRANS: Converting Japanese Specific Characters ..................................... 130
KKFCUT: Truncating a String .................................................................... 135
SFTDEL: Deleting the Shift Code From DBCS Data ...................................... 136
SFTINS: Inserting the Shift Code Into DBCS Data ....................................... 137

7. Data Source and Decoding Functions ......................................... 139
CHECKMD5: Computing an MD5 Hash Check Value ..................................... 139
CHECKSUM: Computing a Hash Sum .................................................... 140
DB_EXPR: Inserting an SQL Expression Into a Request .................................. 140
DB_INFILE: Testing Values Against a File or an SQL Subquery ......................... 142
DB_LOOKUP: Retrieving Data Source Values ........................................... 147
DECODE: Decoding Values ................................................................... 149
8. Simplified Date and Date-Time Functions .............................................. 155

DT_CURRENT_DATE: Returning the Current Date ........................................ 156
DT_CURRENT_DATETIME: Returning the Current Date and Time ...................... 156
DT_CURRENT_TIME: Returning the Current Time ........................................ 157
DTADD: Incrementing a Date or Date-Time Component .................................. 157
DTDIFF: Returning the Number of Component Boundaries Between Date or Date-Time Values 159
DTIME: Extracting Time Components From a Date-Time Value ........................... 160
DTPART: Returning a Date or Date-Time Component in Integer Format .............. 161
DTRUNC: Returning the Start of a Date Period for a Given Date ....................... 162

9. Date Functions ..................................................................................... 165

Overview of Date Functions .......................................................................... 166
Using Standard Date Functions ..................................................................... 166
  Specifying Work Days. .............................................................................. 167
 Specifying Business Days. .......................................................................... 167
  Specifying Holidays. ................................................................................. 168
  Enabling Leading Zeros For Date and Time Functions in Dialogue Manager ... 172
DATEADD: Adding or Subtracting a Date Unit to or From a Date ...................... 173
DATECVT: Converting the Format of a Date ................................................ 175
DATEDIF: Finding the Difference Between Two Dates .................................... 177
DATEMOV: Moving a Date to a Significant Point ......................................... 179
DATETRAN: Formatting Dates in International Formats .................................. 183
FIYR: Obtaining the Financial Year ............................................................ 199
FIQTR: Obtaining the Financial Quarter ....................................................... 201
FIYYQ: Converting a Calendar Date to a Financial Date .................................. 203
TODAY: Returning the Current Date ............................................................ 204
Using Legacy Date Functions ........................................................................ 205
  Using Old Versions of Legacy Date Functions .......................................... 206
AYMD: Adding or Subtracting Days .............................................................. 206
CHGDAT: Changing How a Date String Displays .......................................... 207
10. Date-Time Functions ................................................................. 217
   Using Date-Time Functions ........................................................ 218
      Date-Time Parameters. ............................................................. 218
         Specifying the Order of Date Components. ............................. 218
         Specifying the First Day of the Week for Use in Date-Time Functions. 219
         Controlling Processing of Date-Time Values ................................ 221
      Supplying Arguments for Date-Time Functions. ............................ 221
   HADD: Incrementing a Date-Time Value ........................................... 223
   HCNVRT: Converting a Date-Time Value to Alphanumeric Format .......... 225
   HDATE: Converting the Date Portion of a Date-Time Value to a Date Format 226
   HDIFF: Finding the Number of Units Between Two Date-Time Values .......... 226
   HDTTM: Converting a Date Value to a Date-Time Value ........................ 228
   HGETC: Storing the Current Local Date and Time in a Date-Time Field .... 229
   HGETZ: Storing the Current Coordinated Universal Time in a Date-Time Field 230
   HHMMSS: Retrieving the Current Time .............................................. 231
   HHMS: Converting a Date-Time Value to a Time Value ......................... 232
   HINPUT: Converting an Alphanumeric String to a Date-Time Value ........... 233
   HMDNT: Setting the Time Portion of a Date-Time Value to Midnight .......... 234
   HNAME: Retrieving a Date-Time Component in Alphanumeric Format ........... 235
   HPART: Retrieving a Date-Time Component as a Numeric Value ............... 235
   HSETPT: Inserting a Component Into a Date-Time Value ....................... 236
   HTIME: Converting the Time Portion of a Date-Time Value to a Number ........ 237
   HTMTOTS: Converting a Time to a Timestamp ..................................... 238
   HYYWD: Returning the Year and Week Number From a Date-Time Value ....... 239
11. Simplified Conversion Functions ................................................. 241
EXPN: Evaluating a Number in Scientific Notation ........................................... 271
INT: Finding the Greatest Integer ........................................................................... 272
LOG: Calculating the Natural Logarithm ................................................................. 273
MAX and MIN: Finding the Maximum or Minimum Value ........................................ 273
NORMSDST and NORMSINV: Calculating Normal Distributions ............................ 274
NORMSDST: Calculating Standard Cumulative Normal Distribution ....................... 274
NORMSINV: Calculating Inverse Cumulative Normal Distribution ......................... 276
PRDNOR and PRDUNI: Generating Reproducible Random Numbers ....................... 277
RDNORM and RDUNIF: Generating Random Numbers ............................................. 278
SQRT: Calculating the Square Root ........................................................................... 279

15. Simplified Statistical Functions ........................................................................... 281
Specify the Partition Size for Simplified Statistical Functions ................................. 281
CORRELATION: Calculating the Degree of Correlation Between Two Sets of Data .... 282
KMEANS_CLUSTER: Partitioning Observations Into Clusters Based on the Nearest Mean Value ................................................................. 282
MULTIREGRESS: Creating a Multivariate Linear Regression Column....................... 285
RSERVE: Running an R Script .................................................................................. 287
STDDEV: Calculating the Standard Deviation for a Set of Data Values ..................... 292

16. Simplified System Functions .............................................................................. 295
EDAPRINT: Inserting a Custom Message in the EDAPRINT Log File ...................... 295
ENCRYPT: Encrypting a Password ............................................................................ 296
GETENV: Retrieving the Value of an Environment Variable .................................... 297
PUTENV: Assigning a Value to an Environment Variable ....................................... 297

17. System Functions ............................................................................................. 299
CLSDREC: Closing All Files Opened by the PUTDDREC Function ......................... 299
FEXERR: Retrieving an Error Message ..................................................................... 300
FGETENV: Retrieving the Value of an Environment Variable .................................. 301
FPUTENV: Assigning a Value to an Environment Variable ...................................... 301
GETUSER: Retrieving a User ID ............................................................................... 303
JOBNAME: Retrieving the Current Process Identification String .......................... 303
PUTDDREC: Writing a Character String as a Record in a Sequential File .............. 305
SLEEP: Suspending Execution for a Given Number of Seconds ............................. 306
18. **Simplified Geography Functions** ........................................... 309

Sample Geography Files ............................................................. 310
GIS_DISTANCE: Calculating the Distance Between Geometry Points ................. 314
GIS_DRIVE_ROUTE: Calculating the Driving Directions Between Geometry Points ...... 316
GIS_GEOCODE_ADDR: Geocoding a Complete Address .................................. 318
GIS_GEOCODE_ADDR_CITY: Geocoding an Address Line, City, and State ............... 319
GIS_GEOCODE_ADDR_POSTAL: Geocoding an Address Line and Postal Code ........... 321
GIS_GEOMETRY: Building a JSON Geometry Object ...................................... 322
GIS_IN_POLYGON: Determining if a Point is in a Complex Polygon ..................... 324
GIS_LINE: Building a JSON Line ...................................................... 326
GIS_POINT: Building a Geometry Point ................................................. 328
GIS_SERVICE_AREA: Calculating a Geometry Area Around a Given Point ............. 330
GIS_SERV_AREA_XY: Calculating a Service Area Around a Given Coordinate .......... 332

19. **SQL Character Functions** ................................................... 335

CHAR_LENGTH: Finding the Length of a Character String .................................. 335
CONCAT: Concatenating Two Character Strings ........................................... 336
DIGITS: Converting a Numeric Value to a Character String ................................ 337
EDIT: Editing a Value According to a Format (SQL) ....................................... 338
LCASE: Converting a Character String to Lowercase ..................................... 339
LTRIM: Removing Leading Spaces ..................................................... 339
POSITION: Finding the Position of a Substring ........................................... 340
RTRIM: Removing Trailing Spaces ..................................................... 341
SUBSTR: Extracting a Substring From a String Value (SQL) ................................ 341
TRIM: Removing Leading or Trailing Characters (SQL) ..................................... 343
UCASE: Converting a Character String to Uppercase ..................................... 344
VARGRAPHIC: Converting to Double-byte Character Data .................................. 344

20. **SQL Date and Time Functions** ............................................. 347

CURRENT_DATE: Obtaining the Date ................................................. 347
CURRENT_TIME: Obtaining the Time .................................................. 348
CURRENT_TIMESTAMP: Obtaining the Timestamp (Date/Time) ............................ 348
DAY: Obtaining the Day of the Month From a Date/Timestamp ............................ 349
DAYS: Obtaining the Number of Days Since January 1, 0001 ........................................ 350
EXTRACT: Obtaining a Datetime Field From Date/Time/Timestamp .............................. 350
HOUR: Obtaining the Hour From Time/Timestamp .................................................... 350
MICROSECOND: Obtaining Microseconds From Time/Timestamp ................................ 351
MILLISECOND: Obtaining Milliseconds From Time/Timestamp ..................................... 352
MINUTE: Obtaining the Minute From Time/Timestamp .............................................. 352
MONTH: Obtaining the Month From Date/Timestamp ................................................ 353
SECOND: Obtaining the Second Field From Time/Timestamp .................................. 353
QUARTER: Returning the Quarter of the Year ....................................................... 354
WEEKDAY: Returning the Day of the Week .......................................................... 354
YEAR: Obtaining the Year From a Date or Timestamp ............................................. 355

21. SQL Data Type Conversion Functions .......................................................... 359
   CAST: Converting to a Specific Data Type ....................................................... 356
   CHAR: Converting to a Character String ....................................................... 356
   CHAR: Converting to a Standard Date-Time Format ......................................... 357
   DATE: Converting to a Date ................................................................. 357
   DECIMAL: Converting to Decimal Format ..................................................... 358
   FLOAT: Converting to Floating Point Format ............................................... 359
   INT: Converting to an Integer ................................................................. 360
   SMALLINT: Converting to a Small Integer ................................................... 360
   TIME: Converting to a Time ................................................................. 361
   TIMESTAMP: Converting to a Timestamp .................................................... 362

22. SQL Numeric Functions ................................................................................. 366
   ABS: Returning an Absolute Value (SQL) ..................................................... 367
   CEIL: Returning the Smallest Integer Greater Than or Equal to a Value ................. 367
   FLOOR: Returning the Largest Integer Less Than or Equal to a Value (SQL) ......... 367
   GREATEST: Returning the Largest Value ..................................................... 368
   LEAST: Returning the Smallest Value ......................................................... 368
   LOG: Returning a Logarithm (SQL) ............................................................. 369
   EXP: Returning e Raised to a Power ........................................................... 370
   MOD: Returning the Remainder of a Division ................................................. 370
   POWER: Raising a Value to a Power (SQL) ................................................... 370
SQRT: Returning a Square Root (SQL) ................................................. 373

23. SQL Miscellaneous Functions ...................................................... 375
   COUNTBY: Incrementing Column Values Row by Row ........................................ 375
   DB_EXPR: Inserting an SQL Expression Into a Request (SQL) ..................................... 376
   HEX: Converting to Hexadecimal .............................................................................. 377
   IF: Testing a Condition .............................................................................................. 378
   LENGTH: Obtaining the Physical Length of a Data Item ........................................ 379
   VALUE: Coalescing Data Values .................................................................................. 379

24. SQL Operators .............................................................................. 381
   CASE: SQL Case Operator ......................................................................................... 381
   COALESCE: Coalescing Data Values ............................................................................ 383
   NULLIF: NULLIF Operator ............................................................................................. 384
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</td>
<td>Functions Overview</td>
</tr>
<tr>
<td>2</td>
<td>Simplified Analytic Functions</td>
</tr>
<tr>
<td>3</td>
<td>Simplified Character Functions</td>
</tr>
<tr>
<td>4</td>
<td>Character Functions</td>
</tr>
<tr>
<td>5</td>
<td>Variable Length Character Functions</td>
</tr>
<tr>
<td>6</td>
<td>Character Functions for DBCS Code Pages</td>
</tr>
<tr>
<td>7</td>
<td>Data Source and Decoding Functions</td>
</tr>
<tr>
<td>8</td>
<td>Simplified Date and Date-Time Functions</td>
</tr>
<tr>
<td>9</td>
<td>Date Functions</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></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>
</tbody>
</table>
### Chapter/Appendix

<table>
<thead>
<tr>
<th>Chapter/Appendix</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>Convention</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>...</td>
<td>Indicates that you can enter a parameter multiple times. Type only the parameter, not the ellipsis (...).</td>
</tr>
<tr>
<td>.</td>
<td>Indicates that there are (or could be) intervening or additional commands.</td>
</tr>
</tbody>
</table>

**Related Publications**

Visit our Technical Content Library, [http://documentation.informationbuilders.com](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](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](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](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](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](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></td>
</tr>
<tr>
<td>41</td>
<td>)</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>,</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>:</td>
<td>colon</td>
</tr>
<tr>
<td>59</td>
<td>;</td>
<td>semicolon</td>
</tr>
<tr>
<td>Decimal</td>
<td>ASCII</td>
<td>EBCDIC</td>
</tr>
<tr>
<td>---------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>60</td>
<td>&lt;</td>
<td>less-than sign</td>
</tr>
<tr>
<td>61</td>
<td>=</td>
<td>equal sign</td>
</tr>
<tr>
<td>62</td>
<td>&gt;</td>
<td>greater-than sign</td>
</tr>
<tr>
<td>63</td>
<td>?</td>
<td>question mark</td>
</tr>
<tr>
<td>64</td>
<td>@</td>
<td>at sign</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>67</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>68</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>69</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>70</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>71</td>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>72</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>73</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>74</td>
<td>J</td>
<td>J</td>
</tr>
<tr>
<td>75</td>
<td>K</td>
<td>K</td>
</tr>
<tr>
<td>76</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>77</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>78</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>79</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>80</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>81</td>
<td>Q</td>
<td>Q</td>
</tr>
<tr>
<td>Decimal</td>
<td>ASCII</td>
<td>EBCDIC</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>--------</td>
</tr>
<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>`</td>
</tr>
<tr>
<td>193</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>194</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>195</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>196</td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>197</td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>198</td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>199</td>
<td></td>
<td>G</td>
</tr>
<tr>
<td>200</td>
<td></td>
<td>H</td>
</tr>
<tr>
<td>201</td>
<td></td>
<td>I</td>
</tr>
<tr>
<td>209</td>
<td></td>
<td>J</td>
</tr>
<tr>
<td>210</td>
<td></td>
<td>K</td>
</tr>
<tr>
<td>211</td>
<td></td>
<td>L</td>
</tr>
<tr>
<td>212</td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>213</td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>214</td>
<td></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>
</tbody>
</table>
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 Field Values in Calculations

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**

```
FORECAST_MOVAVE(display, infield, interval, npredict, 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.

**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, npredict is the number of *periods* to calculate. The number of *points* generated is:

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

\[ n_{\text{point}} \]

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.

```
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:
  \( \frac{801,123 + 682,340}{2} \).

<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>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>749,513.7</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>57050</td>
<td>691274</td>
<td>712,897.3</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>59229</td>
<td>720444</td>
<td>725,598.7</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>58466</td>
<td>742457</td>
<td>718,058.3</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>60771</td>
<td>747253</td>
<td>736,718.0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>54633</td>
<td>655896</td>
<td>715,202.0</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>57829</td>
<td>730317</td>
<td>711,155.3</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>57012</td>
<td>724412</td>
<td>703,541.7</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>51110</td>
<td>620264</td>
<td>691,664.3</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>58981</td>
<td>762328</td>
<td>702,334.7</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>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>671,534.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>58026</td>
<td>718514</td>
<td>686,796.3</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>53289</td>
<td>660740</td>
<td>674,018.7</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>58742</td>
<td>734705</td>
<td>704,653.0</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>60127</td>
<td>760586</td>
<td>718,677.0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>55622</td>
<td>695235</td>
<td>730,175.3</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>55787</td>
<td>683140</td>
<td>712,987.0</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>57340</td>
<td>713768</td>
<td>697,381.0</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>57459</td>
<td>710138</td>
<td>702,348.7</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>57290</td>
<td>705315</td>
<td>709,740.3</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>

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:
  \( \frac{801,123 + 682,340}{2} \).
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.

```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(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
```
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>801123</td>
<td>801,123.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>54870</td>
<td>682340</td>
<td>682,340.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>61608</td>
<td>765078</td>
<td>765,078.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>57050</td>
<td>691274</td>
<td>691,274.0</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>59229</td>
<td>720444</td>
<td>720,444.0</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>58466</td>
<td>742457</td>
<td>742,457.0</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>60771</td>
<td>747253</td>
<td>747,253.0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>54633</td>
<td>655896</td>
<td>655,896.0</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>57829</td>
<td>730317</td>
<td>730,317.0</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>57012</td>
<td>724412</td>
<td>724,412.0</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>51110</td>
<td>620264</td>
<td>620,264.0</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>58981</td>
<td>762328</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>672727</td>
<td>672,727.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>54894</td>
<td>699073</td>
<td>699,073.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>52713</td>
<td>642802</td>
<td>642,802.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>58026</td>
<td>718514</td>
<td>718,514.0</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>53289</td>
<td>660740</td>
<td>660,740.0</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>58742</td>
<td>734705</td>
<td>734,705.0</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>60127</td>
<td>760586</td>
<td>760,586.0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>55622</td>
<td>695235</td>
<td>695,235.0</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>55787</td>
<td>683140</td>
<td>683,140.0</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>57340</td>
<td>713768</td>
<td>713,768.0</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>57459</td>
<td>710138</td>
<td>710,138.0</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>57290</td>
<td>705315</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:

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

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

```
FORECAST_EXPAVE(display, infield, interval, npredict, 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:

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

**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 (EXPANE 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:

\[
\begin{align*}
    n &= 3 \text{ (number used to calculate weights)} \\
    k &= \frac{2}{1+n} = \frac{2}{4} = 0.5 \\
    \text{EXPANE} &= (\text{EXPANE} \times (1-k)) + (\text{new-DOLLARS} \times k) \\
    &= (801123 \times 0.5) + (682340 \times 0.50) \\
    &= 400561.5 + 341170 = 741731.5
\end{align*}
\]
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:**

How to Calculate a Double Exponential Smoothing Column

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

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

**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>12,957,681.0</td>
<td>12,957,681.0</td>
<td>12,957,681.0</td>
</tr>
<tr>
<td>2002/02</td>
<td>25,441,971.0</td>
<td>19,199,826.0</td>
<td>22,439,246.3</td>
</tr>
<tr>
<td>2002/03</td>
<td>39,164,321.0</td>
<td>29,182,073.5</td>
<td>34,791,885.1</td>
</tr>
<tr>
<td>2002/04</td>
<td>52,733,326.0</td>
<td>40,957,699.8</td>
<td>48,845,816.0</td>
</tr>
<tr>
<td>2002/05</td>
<td>66,765,920.0</td>
<td>53,861,809.9</td>
<td>63,860,955.9</td>
</tr>
<tr>
<td>2002/06</td>
<td>80,952,492.0</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:

$$SEASONAL(t) = k \times (datavalue(t)/I(t-L)) + (1-k) \times (SEASONAL(t-1) + b(t-1))$$

- 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:

$$b(t) = g \times (SEASONAL(t)-SEASONAL(t-1)) + (1-g) \times b(t-1)$$

- 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:

$$I(t) = p \times (datavalue(t)/SEASONAL(t)) + (1-p) \times I(t-L)$$

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:

$$b(0) = (1/L) \times ((y(L+1)-y(1))/L + (y(L+2)-y(2))/L + \ldots + (y(2L) - y(L))/L)$$

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):

$$A(j) = (y((j-1)L+1) + y((j-1)L+2) + \ldots + y(jL))/L$$

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):

$$I(n) = (y(n)/A(1) + y(L+n)/A(2) + \ldots + y((N-1)L+n)/A(N))/N$$
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:

\[
\text{forecast}(t+m) = (\text{SEASONAL}(t) + m \times b(t)) / I(t-L+\text{MOD}(m/L))
\]

where:

\( m \)

Is the number of periods ahead for the forecast.

**Syntax:**

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

```plaintext
FORECAST_SEASONAL(display, infield, interval, npredict, nperiod, npoint1, npoint2, 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.

\textit{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 \textit{periods} to calculate. The number of \textit{points} generated is:

\textit{nperiod} \times \textit{npredict}

\textit{nperiod}

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

\textit{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, \textit{k}, is calculated by the following formula:

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

\textit{npoint2}

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

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

\textit{npoint3}

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

\[ p = \frac{2}{1 + \textit{npoint3}} \]
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:

```
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>44.72</td>
<td>34.6</td>
</tr>
<tr>
<td>91/06/21</td>
<td>126.28</td>
<td>53.2</td>
</tr>
<tr>
<td>91/06/24</td>
<td>47.74</td>
<td>75.3</td>
</tr>
<tr>
<td>91/06/25</td>
<td>40.97</td>
<td>82.7</td>
</tr>
<tr>
<td>91/06/26</td>
<td>60.24</td>
<td>73.7</td>
</tr>
<tr>
<td>91/06/27</td>
<td>31.00</td>
<td>66.3</td>
</tr>
<tr>
<td>91/06/28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>91/06/29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>91/06/30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>91/07/01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>91/07/02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>91/07/03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>91/07/04</td>
<td></td>
<td></td>
</tr>
<tr>
<td>91/07/05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>91/07/06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>91/07/07</td>
<td></td>
<td></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.

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

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

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.
Syntax: 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.

---

**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 the current row. You can stop the rolling calculation at the current row 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

PARTITION_AGGR([prefix.]measure, {sortfield|TABLE}, from, to, operation)

where:

**prefix.**

Defines an aggregation operator to apply to the measure before using it in the rolling calculation. Valid operators are:

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

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

**measure**

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.

**sortfield**

Is a BY or ACROSS field that defines the boundary of the partition. Operations will not cross a boundary. In the request the BY HIGHEST phrase to sort high-to-low is supported. ACROSS COLUMNS AND is also supported, but BY ROWS OVER and FOR are not supported.

Specifying TABLE as the boundary makes the partition boundary the entire internal matrix.
For example, if the sort is BY YEAR BY MONTH, with data from both 2014 and 2015, specifying the boundary as YEAR means that January 2015 - 2 will be valued as zero (0) or MISSING, as two months prior to January 2015 would cross the YEAR boundary. However, specifying TABLE as the boundary and requesting - 2 months would return the data for November 2014.

*from*

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

- `-n`, which starts the calculation `n` rows 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).

*to*

Identifies the ending point of the rolling calculation. Valid values are:

- `C`, which ends the rolling calculation at the current row in the internal matrix.
- `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.)

*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
END
```
PARTITION_AGGR: Creating Rolling Calculations

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>$131,745.00</td>
<td>$131,745.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>$134,947.50</td>
<td>$134,947.50</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>$152,123.50</td>
<td>$152,123.50</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>$147,770.00</td>
<td>$147,770.00</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1</td>
<td>$145,615.50</td>
<td>$145,615.50</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1</td>
<td>$143,781.00</td>
<td>$143,781.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>$140,641.00</td>
<td>$140,641.00</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>$139,698.50</td>
<td>$139,698.50</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>1</td>
<td>$135,307.00</td>
<td>$135,307.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>$172,837.00</td>
<td>$172,837.00</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1</td>
<td>$87,220.50</td>
<td>$87,220.50</td>
</tr>
<tr>
<td>South America</td>
<td>1</td>
<td>1</td>
<td>$33,895.00</td>
<td>$33,895.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>$34,824.00</td>
<td>$34,824.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>$24,753.00</td>
<td>$24,753.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>$24,339.00</td>
<td>$24,339.00</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1</td>
<td>$26,707.50</td>
<td>$26,707.50</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1</td>
<td>$25,901.00</td>
<td>$25,901.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>$16,475.00</td>
<td>$16,475.00</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>$14,489.50</td>
<td>$14,489.50</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>1</td>
<td>$16,128.50</td>
<td>$16,128.50</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>$27,526.00</td>
<td>$27,526.00</td>
</tr>
</tbody>
</table>

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

Compute Ave1/D12.2M = PARTITION_AGGR(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 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>$131,745.00</td>
<td>$131,745.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>$138,150.00</td>
<td>$134,947.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>$166,097.00</td>
<td>$145,330.67</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>$147,770.00</td>
<td>$147,770.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>$143,461.00</td>
<td>$145,615.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>$144,101.00</td>
<td>$145,110.67</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>$140,641.00</td>
<td>$140,641.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>$138,756.00</td>
<td>$139,698.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>$131,858.00</td>
<td>$137,085.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>$172,837.00</td>
<td>$172,837.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>$1,604.00</td>
<td>$87,220.50</td>
</tr>
<tr>
<td>South America</td>
<td>1</td>
<td>1</td>
<td>$33,895.00</td>
<td>$33,895.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>$35,753.00</td>
<td>$34,824.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>$13,753.00</td>
<td>$27,800.33</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>$24,339.00</td>
<td>$24,339.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>$29,076.00</td>
<td>$26,707.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>$22,726.00</td>
<td>$25,380.33</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>$16,475.00</td>
<td>$16,475.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>$12,504.00</td>
<td>$14,489.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>$19,753.00</td>
<td>$16,244.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>$27,526.00</td>
<td>$27,526.00</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 until a break in the business region sort field.

<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>$131,745.00</td>
<td>$131,745.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>$138,150.00</td>
<td>$134,947.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>$166,097.00</td>
<td>$145,330.67</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>$147,770.00</td>
<td>$145,940.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>$143,461.00</td>
<td>$145,444.60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>$144,101.00</td>
<td>$145,220.67</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>$140,641.00</td>
<td>$144,566.43</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>$138,756.00</td>
<td>$143,840.13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>$131,858.00</td>
<td>$142,508.78</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>$172,837.00</td>
<td>$145,541.60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>$1,604.00</td>
<td>$132,456.36</td>
</tr>
<tr>
<td>South America</td>
<td>1</td>
<td>1</td>
<td>$33,895.00</td>
<td>$124,242.92</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>$35,753.00</td>
<td>$117,436.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>$13,753.00</td>
<td>$110,030.07</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>$24,339.00</td>
<td>$104,317.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>$29,076.00</td>
<td>$99,614.75</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>$22,726.00</td>
<td>$95,091.88</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>$16,475.00</td>
<td>$90,724.28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>$12,504.00</td>
<td>$86,607.42</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>$19,753.00</td>
<td>$83,264.70</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>$27,526.00</td>
<td>$80,610.48</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.

```sql
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
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>$435,992.00</td>
<td>$435,992.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>$435,992.00</td>
<td>$435,992.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td>$435,992.00</td>
<td>$435,992.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>$435,332.00</td>
<td>$435,332.00</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td>$435,332.00</td>
<td>$435,332.00</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td>$435,332.00</td>
<td>$435,332.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>$411,255.00</td>
<td>$411,255.00</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td>$411,255.00</td>
<td>$411,255.00</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
<td>$411,255.00</td>
<td>$411,255.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>$174,441.00</td>
<td>$174,441.00</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
<td>$174,441.00</td>
<td>$174,441.00</td>
</tr>
<tr>
<td>South America</td>
<td>1</td>
<td>1</td>
<td>$83,401.00</td>
<td>$83,401.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>$83,401.00</td>
<td>$83,401.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td>$83,401.00</td>
<td>$83,401.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>$76,141.00</td>
<td>$76,141.00</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td>$76,141.00</td>
<td>$76,141.00</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td>$76,141.00</td>
<td>$76,141.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>$48,732.00</td>
<td>$48,732.00</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td>$48,732.00</td>
<td>$48,732.00</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
<td>$48,732.00</td>
<td>$48,732.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>$27,526.00</td>
<td>$27,526.00</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 with quarter using PARTITION_AGG.

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, $435,992.00/1514 =$287.97), while PARTITION_AGG divides the total cost for the quarter by the number of report rows in the quarter (for example, $435,992.00/3 = $145,330.67).

<table>
<thead>
<tr>
<th>Region</th>
<th>Sale Quarter</th>
<th>Sale Month</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>$131,745.00</td>
<td>483</td>
<td>$287.97</td>
<td>$145,330.67</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>$138,150.00</td>
<td>485</td>
<td>$287.97</td>
<td>$145,330.67</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>$166,097.00</td>
<td>546</td>
<td>$287.97</td>
<td>$145,330.67</td>
</tr>
<tr>
<td></td>
<td>*TOTAL TIME_QTR 1</td>
<td></td>
<td>$435,992.00</td>
<td>1514</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>$147,770.00</td>
<td>497</td>
<td>$298.79</td>
<td>$145,110.67</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5</td>
<td>$143,461.00</td>
<td>502</td>
<td>$298.79</td>
<td>$145,110.67</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>6</td>
<td>$144,101.00</td>
<td>548</td>
<td>$298.79</td>
<td>$145,110.67</td>
</tr>
<tr>
<td></td>
<td>*TOTAL TIME_QTR 2</td>
<td></td>
<td>$435,332.00</td>
<td>1457</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>$140,641.00</td>
<td>451</td>
<td>$299.97</td>
<td>$137,085.00</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8</td>
<td>$138,756.00</td>
<td>451</td>
<td>$299.97</td>
<td>$137,085.00</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>9</td>
<td>$131,858.00</td>
<td>451</td>
<td>$299.97</td>
<td>$137,085.00</td>
</tr>
<tr>
<td></td>
<td>*TOTAL TIME_QTR 3</td>
<td></td>
<td>$411,255.00</td>
<td>1371</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>$172,837.00</td>
<td>569</td>
<td>$302.32</td>
<td>$87,220.50</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>11</td>
<td>$1,604.00</td>
<td>8</td>
<td>$302.32</td>
<td>$87,220.50</td>
</tr>
<tr>
<td></td>
<td>*TOTAL TIME_QTR 4</td>
<td></td>
<td>$174,441.00</td>
<td>577</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td></td>
<td>$1,457,020.00</td>
<td>4919</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 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 to go in order to retrieve a prior value, and a sort break within which the prior value calculation will be contained.

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

\[
\text{PARTITION}_{\text{REF}}([prefix.]\text{field}, \{\text{sortfield|TABLE}\}, -\text{offset})
\]

where:

prefix

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

- AVE. Average
- MAX. Maximum
- MIN. Minimum
- CNT. Count
- SUM. Sum

field

Is the field whose prior value is to be retrieved.

\{sortfield|TABLE\}

Is the sort break within which to go back to retrieve the value. TABLE means retrieve the value without regard to sort breaks. Operations will not cross a partition boundary.

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.

- offset

Is the integer number of records back to go 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.

```sql
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
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>164</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Headphones</td>
<td>335</td>
<td>164</td>
</tr>
<tr>
<td></td>
<td>Universal Remote Controls</td>
<td>264</td>
<td>335</td>
</tr>
<tr>
<td>Camcorder</td>
<td>Handheld</td>
<td>430</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Professional</td>
<td>21</td>
<td>430</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>324</td>
<td>21</td>
</tr>
<tr>
<td>Computers</td>
<td>Smartphone</td>
<td>342</td>
<td>0</td>
</tr>
<tr>
<td>Media Player</td>
<td>Blu Ray</td>
<td>1,102</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>DVD Players</td>
<td>31</td>
<td>1102</td>
</tr>
<tr>
<td></td>
<td>Streaming</td>
<td>85</td>
<td>31</td>
</tr>
<tr>
<td>Stereo Systems</td>
<td>Home Theater Systems</td>
<td>652</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Receivers</td>
<td>220</td>
<td>652</td>
</tr>
<tr>
<td></td>
<td>Speaker Kits</td>
<td>404</td>
<td>220</td>
</tr>
<tr>
<td></td>
<td>iPod Docking Station</td>
<td>520</td>
<td>404</td>
</tr>
<tr>
<td>Televisions</td>
<td>Flat Panel TV</td>
<td>144</td>
<td>0</td>
</tr>
<tr>
<td>Video Production</td>
<td>Video Editing</td>
<td>317</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 PRODUCT_CATEGORY sort field.

```
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 PRODUCT_CATEGORY
BY PRODUCT_SUBCATEG
ON TABLE SET PAGE NOPAGE
END
```
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>Partition Ave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessories</td>
<td>Charger</td>
<td>$8,573.00</td>
<td>$26.96</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Headphones</td>
<td>$199,325.00</td>
<td>$292.69</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Universal Remote Controls</td>
<td>$134,979.00</td>
<td>$278.31</td>
<td>$26.96</td>
</tr>
<tr>
<td>Camcorder</td>
<td>Handheld</td>
<td>$90,607.00</td>
<td>$114.11</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Professional</td>
<td>$159,048.00</td>
<td>$3,614.73</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>$203,550.00</td>
<td>$351.55</td>
<td>$114.11</td>
</tr>
<tr>
<td>Computers</td>
<td>Smartphone</td>
<td>$109,281.00</td>
<td>$182.13</td>
<td>$0.00</td>
</tr>
<tr>
<td>Media Player</td>
<td>Blu Ray</td>
<td>$754,249.00</td>
<td>$374.32</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>DVD Players</td>
<td>$15,096.00</td>
<td>$215.66</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Streaming</td>
<td>$10,248.00</td>
<td>$62.49</td>
<td>$374.32</td>
</tr>
<tr>
<td>Stereo Systems</td>
<td>Home Theater Systems</td>
<td>$245,581.00</td>
<td>$199.98</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Receivers</td>
<td>$159,790.00</td>
<td>$369.88</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Speaker Kits</td>
<td>$343,113.00</td>
<td>$458.09</td>
<td>$199.98</td>
</tr>
<tr>
<td></td>
<td>iPod Docking Station</td>
<td>$108,558.00</td>
<td>$116.85</td>
<td>$369.88</td>
</tr>
<tr>
<td>Televisions</td>
<td>Flat Panel TV</td>
<td>$227,820.00</td>
<td>$853.26</td>
<td>$0.00</td>
</tr>
<tr>
<td>Video Production</td>
<td>Video Editing</td>
<td>$180,540.00</td>
<td>$279.91</td>
<td>$0.00</td>
</tr>
</tbody>
</table>
Replacing the function call with the following syntax changes the partition boundary to TABLE.

```
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>$8,573.00</td>
<td>$26.96</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Headphones</td>
<td>$199,325.00</td>
<td>$292.69</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Universal Remote Controls</td>
<td>$134,979.00</td>
<td>$278.31</td>
<td>$26.96</td>
</tr>
<tr>
<td>Camcorder</td>
<td>Handheld</td>
<td>$90,607.00</td>
<td>$114.11</td>
<td>$292.69</td>
</tr>
<tr>
<td></td>
<td>Professional</td>
<td>$159,048.00</td>
<td>$3,614.73</td>
<td>$278.31</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>$203,550.00</td>
<td>$351.55</td>
<td>$114.11</td>
</tr>
<tr>
<td>Computers</td>
<td>Smartphone</td>
<td>$109,281.00</td>
<td>$182.13</td>
<td>$3,614.73</td>
</tr>
<tr>
<td>Media Player</td>
<td>Blu Ray</td>
<td>$754,249.00</td>
<td>$374.32</td>
<td>$351.55</td>
</tr>
<tr>
<td></td>
<td>DVD Players</td>
<td>$15,096.00</td>
<td>$215.66</td>
<td>$182.13</td>
</tr>
<tr>
<td></td>
<td>Streaming</td>
<td>$10,248.00</td>
<td>$62.49</td>
<td>$374.32</td>
</tr>
<tr>
<td>Stereo Systems</td>
<td>Home Theater Systems</td>
<td>$245,581.00</td>
<td>$199.98</td>
<td>$215.66</td>
</tr>
<tr>
<td></td>
<td>Receivers</td>
<td>$159,790.00</td>
<td>$369.88</td>
<td>$62.49</td>
</tr>
<tr>
<td></td>
<td>Speaker Kits</td>
<td>$343,113.00</td>
<td>$458.09</td>
<td>$199.98</td>
</tr>
<tr>
<td></td>
<td>iPod Docking Station</td>
<td>$108,558.00</td>
<td>$116.85</td>
<td>$369.88</td>
</tr>
<tr>
<td>Televisions</td>
<td>Flat Panel TV</td>
<td>$227,820.00</td>
<td>$853.26</td>
<td>$458.09</td>
</tr>
<tr>
<td>Video Production</td>
<td>Video Editing</td>
<td>$180,540.00</td>
<td>$279.91</td>
<td>$116.85</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.
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).

**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
- **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
- **PATTERNS**: Returning a Pattern That Represents the Structure of the Input 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

```
CHAR_LENGTH(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

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

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

DIGITS converts the integer expression ID_PRODUCT+1 to a six-character string:

DIGITS(ID_PRODUCT, 6)

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

Functions Reference 65
**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.

**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.

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.

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>A</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

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

where:

\[
\text{string}
\]

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

\[
\text{LOWER(} \text{LAST\_NAME} \text{)}
\]

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

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

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

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

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**

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

where:

\[\text{string}\]

Alphanumeric

Is a string whose pattern will be returned.

**Example:**

**Returning a Pattern Representing an Input String**

PATTERNS returns the pattern representing the field ADDRESS_LINE_1.

\[
\text{PATTERNS}(\text{ADDRESS\_LINE\_1})
\]

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

\[
\text{POSITION(pattern, string)}
\]

where:

- \text{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.
- \text{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

\[
\text{POSITION(determines the position of the first capital letter I in LAST\_NAME.}
\]

\[
\text{POSITION(’I’, LAST\_NAME)}
\]

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.

Syntax: How to Match a String to a Regular Expression

\[
\text{REGEX(string, regular\_expression)}
\]

where:

- \text{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
- | represents the Or operator
- \ is the Escape Special Character
- () contains a character sequence

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

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

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.'

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

For South Africa, the result is S. Africa.

Example: Replacing All Instances of a String

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

RPAD right-pads the PRODUCTCATEGORY column with @ symbols:

```sql
RPAD(PRODUCTCATEGORY, 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

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.
SUBSTRING: Extracting a Substring From a Source 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

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**
  Alphanumeric
  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

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

\[ \text{SUBSTRING(LAST\_NAME, I\_IN\_NAME, I\_IN\_NAME+2)} \]

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 character and specified by a token number reflecting the position of the token in the string.

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

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

where:

- \( \text{string} \)
  - Fixed length alphanumeric
  - Is the character string from which to extract the token.

- \( \text{delimiter} \)
  - Fixed length alphanumeric
  - Is a single character delimiter.

- \( \text{number} \)
  - 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 a blank:

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

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

\[ \text{TRIM}_\text{(where, pattern, string)} \]

where:

- \text{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.

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

- \text{string}
  - Alphanumeric
    - Is the string to be trimmed.

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

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

TRIM_(LEADING, 'B', DIRECTOR)

For BROOKS R., the result is ROOKS R.

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

UPPER(string)

where:

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

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.
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

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.

Syntax: How to Measure the Length of a Character String

ARGLEN(length, source_string, output)

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

Example: Measuring the Length of a Character String

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

ARGLEN(15, LAST_NAME, 'I3')

For SMITH, the result is 5.
For BLACKWOOD, the result is 9.

ASIS: Distinguishing Between Space and Zero

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.
  
    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.

```plaintext
-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.

```plaintext
-SET &VAR1 = ' ';  
-SET &VAR2 = 0;  
-IF &VAR2 EQ 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 '0' THEN 'TRUE' ELSE 'FALSE';
  -SET &DMVAR4=IF &DMVAR1 EQ '.' 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('0') EQ '0' 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

The BITSON function evaluates an individual bit within a character string to determine whether it is on or off. If the bit is on, BITSON returns a value of 1. If the bit is off, it returns a value of 0. This function is useful in interpreting multi-punch data, where each punch conveys an item of information.
Syntax: How to Determine If a Bit Is On or Off

BITSON(bitnumber, source_string, output)

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. 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.

Example: Evaluating a Bit in a Field

BITSON evaluates the 24th bit of LAST_NAME:

BITSON(24, LAST_NAME, 'I1')

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

BITVAL: Evaluating a Bit String as an Integer

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

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

```plaintext
BITVAL(source_string, startbit, number, output)
```

where:

- **source_string**
  - Alphanumeric
  - Is the character string to be evaluated.

- **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

**Example:** Evaluating a Bit String

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

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

For SMITH, the result is 332.

For JONES, the result is 365.

**BYTVAL: Translating a Character to Decimal**

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

BYTVAL(character, output)

where:

*character*
   Alphanumeric

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

*output*
   Integer

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

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.

**CHKFMT: Checking the Format of a String**

The CHKFMT function checks a character string for incorrect characters or character types. It compares each character string to a second string, called a mask, 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

\[
\text{CHKFMT}(\text{numchar}, \text{source\_string}, \text{'}\text{mask}\text{'}\text{, output})
\]

where:

\text{numchar}

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

\text{string}

Alphanumeric
Is the character string to be checked.

\text{'}\text{mask}\text{'}

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:

- \text{A} is any letter between A and Z (uppercase or lowercase).
- \text{9} is any digit between 0–9.
- \text{X} is any letter between A–Z or any digit between 0-9.
- \text{$} 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.

\text{output}

Integer

Example: Checking the Format of a Field

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

\[
\text{CHKFMT}(9, \text{EMP\_ID, '119999999', 'I3'})
\]

For 071382660, the result is 1.
For 119265415, the result is 0.
For 23764317, the result is 2.

**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.

- **output**
  - Numeric

**Example:**

Checking a String for Numeric Format

CHKNUM examines STR1 for numeric format.

```plaintext
CHKNUM(8, str1, 'I1')
```

For 12345E01, the result is 1.
For ABCDEFG, the result is 0.

**CTRAN: Translating One Character to Another**

The CTRAN function translates a character within a character string to another character based on its decimal value. This function is especially useful for changing replacement characters to unavailable characters, or to characters that are difficult to input or unavailable on your keyboard. It can also be used for inputting characters that are difficult to enter when responding to a Dialogue Manager-PROMPT command, such as a comma or apostrophe. It eliminates the need to enclose entries in single quotation marks (').
To use CTRAN, you 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 20.

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

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

where:

- **length**
  
  Integer
  
  Is the number of characters in the source string.

- **source_string**
  
  Alphanumeric
  
  Is the character string to be translated.

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

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

- **output**
  
  Alphanumeric
Example: Translating Spaces to Underscores on an ASCII Platform

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

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

For RUTHERFORD NJ 07073, the result is RUTHERFORD_NJ_07073_.

For NEW YORK NY 10039, the result is NEW_YORK_NY_10039___.

CTRFLD: Centering a Character String

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

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

Syntax: How to Center a Character String

CTRFLD(source_string, length, output)

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. This argument must be greater than 0. A length less than 0 can cause unpredictable results.

output
  Alphanumeric

Example: Centering a Field

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

CTRFLD(LAST_NAME, 12, 'A12')
EDIT: Extracting or Adding Characters

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

EDIT compares the characters in a mask to the characters in a source string. When it encounters a 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 250.

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. The length of the mask, excluding characters other than 9 and $, determines the length of the output field.

Example: Extracting Characters

EDIT extracts the first initials from the FNAME column.

```plaintext
EDIT(FNAME, '9$$$$$$$$$')
```
For GREGORY, the result is G.
For STEVEN, the result is S.

**GETTOK: Extracting a Substring (Token)**

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 99.

**Syntax:**

How to Extract a Substring (Token)

```plaintext
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.

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
   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 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
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

\[ \text{LCWORD}(\text{length}, \text{source\_string}, \text{output}) \]

where:

\text{length}  
Integer  
Is the number of characters in \text{source\_string} and \text{output}.

\text{string}  
Alphanumeric  
Is the character string to be converted.

\text{output}  
Alphanumeric

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

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

\[ \text{LCWORD}(15, \text{LAST\_NAME}, \text{'A15'}) \]

For STEVENS, the result is Stevens.
For SMITH, the result is Smith.

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

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

`LCWORD2(length, string, 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
  - The length must be greater than or equal to length.

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

`LCWORD2` converts the string O'CONNOR’s to mixed-case:

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

`LCWORD3(length, string, 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
    The length must be greater than or equal to length.

Example: Converting a Character String to Mixed-Case Using LCWORD3
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
    LJUST left-justifies a character string.

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.
    source_string
        Alphanumeric
        Is the character string to be justified.
    output
        Alphanumeric

Example: Left-Justifying a String
    LJUST left-justifies FNAME and stores the result in a column with the format A25:
    LJUST(15, FNAME, 'A25')

LOCASE: Converting Text to Lowercase
    The LOCASE function converts alphanumeric text to lowercase.
**Syntax:** How to Convert Text to Lowercase

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

where:

- **length**
  - Integer
  - Is the number of characters in `source\_string` and `output`. The length must be greater than 0.

- **source\_string**
  - Alphanumeric
  - Is the character string to convert.

- **output**
  - Alphanumeric

**Example:** Converting a String to Lowercase

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

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

For SMITH, the result is smith.

For JONES, the result is jones.

**OVRLAY: Overlaying a Character String**

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

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

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

where:

- **source\_string**
  - Alphanumeric
  - Is the base character string.
**stringlen**
Integer
Is the number of characters in `source_string` and `output`. 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`. 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.

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

**Example: Replacing Characters in a Character String**

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

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

For EMP_ID of 326179357 with CURR_JOBCODE of B04, the result is 26179B04.
For EMP_ID of 818692173 with CURR_JOBCODE of A17, the result is 818692A17.

**PARAG: Dividing Text Into Smaller Lines**
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 93). 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**

\[
\text{PARAG}(\text{length}, \text{source_string}, \text{'delimiter'}, \text{max_token_size}, \text{output})
\]

where:

- **length**
  - Integer
  - Is the number of characters in `source_string` and `output`.

- **source_string**
  - Alphanumeric
  - Is a string to divide into tokens.

- **delimiter**
  - Alphanumeric
  - Is the delimiter enclosed in single quotation marks. Choose a character that does not appear in the text.

- **max_token_size**
  - Integer
  - Is the upper limit for the size of each token.
Example: Dividing Text Into Smaller Lines

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

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

For 147-15 NORTHERN BLD, the result is 147-15,NORTHERN,BLD.
For 13 LINDEN AVE., the result is 13 LINDEN,AVE.

PATTERN: Generating a Pattern From 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.
Example: Producing a Pattern From Alphanumeric Data

PATTERN generates a pattern for each instance of TESTFLD. The result is stored in a column with the format A14:

\[ \text{PATTERN (14, TESTFLD, 'A14')} \]

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

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.

Syntax: How to Find the Beginning of a Substring

\[ \text{POSIT(source_string, length, substring, sublength, output)} \]

where:

source_string
   Alphanumeric
   Is the string to parse.

length
   Integer
   Is the number of characters in the source string. If this argument is less than or equal to 0, the function returns a 0.

substring
   Alphanumeric
   Is the substring whose position you want to find.
**sublength**

Integer

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

**output**

Integer

**Example:** Finding the Position of a Letter

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

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

For STEVENS, the result is 0.

For SMITH, the result is 3.

For IRVING, the result is 1.

**REVERSE: Reversing the Characters in a String**

The REVERSE function reverses the characters in a string.

**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*.

*source_string*

Alphanumeric

Is the character string to reverse.

*output*

Alphanumeric
**Example:** Reversing the Characters in a String

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

REVERSE(15, PRODCAT, 'A15')

For VCRs, the result is sRCV.
For DVD, the result is DVD.

**RJUST: Right-Justifying a Character String**

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

**Syntax:** How to Right-Justify a Character String

RJUST(length, source_string, output)

where:

- **length** Integer
- **source_string** Alphanumeric
- **output** Alphanumeric

**Example:** Right-Justifying a String

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**

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.
Syntax: How to Compare Character Strings Phonetically

```
SOUNDEX(length, source_string, output)
```

where:

- **length**
  - Alphanumeric
  - Is the number of characters in `source_string`. 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.

- **output**
  - Alphanumeric

Example: Comparing Character Strings Phonetically

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

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

SPELLNM: Spelling Out a Dollar Amount

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`. 
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>
<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

**Example:** Spelling Out a Dollar Amount

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

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

For $13,200.00, the result is THIRTEEN THOUSAND TWO HUNDRED DOLLARS AND NO CENTS.  
For $18,480.00, the result is EIGHTEEN THOUSAND FOUR HUNDRED EIGHTY DOLLARS AND NO CENTS.

**SQUEEZ: Reducing Multiple Spaces to a Single Space**

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:

\[
\text{length} \\
\text{Integer} \\
\text{Is the number of characters in source\_string and output.}
\]

\[
\text{source\_string} \\
\text{Alphanumeric} \\
\text{Is the character string to squeeze.}
\]

\[
\text{output} \\
\text{Alphanumeric}
\]

**Example:** Reducing Multiple Spaces to a Single Space

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

\[
\text{SQUEEZ}(30, \text{NAME}, 'A30')
\]

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**

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

\[
\text{STRIP}(\text{length}, \text{source\_string}, \text{char}, \text{output})
\]

where:

\[
\text{length} \\
\text{Integer} \\
\text{Is the number of characters in source\_string and output.}
\]
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

**Example:**  Removing Occurrences of a Character From a String

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

\[ \text{STRIP}(17, \text{DIRECTOR}, '.', 'A17') \]

For ZEMECKIS R., the result is ZEMECKIS R.

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

**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

\[ \text{STRREP} (\text{inlength}, \text{instring}, \text{searchlength}, \text{searchstring}, \text{replength}, \text{replstring}, \text{outlength}, \text{output}) \]

where:

\[ \text{inlength} \]
  Numeric
  Is the number of characters in the source string.

\[ \text{instring} \]
  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

Reference: Usage Note for STRREP Function
The maximum string length is 4095.

Example: Replacing Commas and Dollar Signs
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
The SUBSTR function extracts a substring based on where it begins and its length in the source string.
**Syntax:**

How to Extract a Substring

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

where:

`length`
- Integer
- Is the number of characters in `source_string`.

`source_string`
- Alphanumeric
- Is the string from which to extract a substring.

`start`
- Integer
- Is the starting position of the substring in the source string. If `start` is less than 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
**Example:** Extracting a String

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

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

For BANNING, the result is BAN.

For MCKNIGHT, the result is MCK.

**TRIM: Removing Leading and Trailing Occurrences**

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

**Syntax:** How to Remove Leading and Trailing Occurrences

```
TRIM(trim_where, source_string, length, pattern, sublength, output)
```

where:

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

- **source_string**
  - Alphanumeric
  - Is the string to trim.

- **string_length**
  - Integer
  - Is the number of characters in the source string.

- **pattern**
  - Alphanumeric
  - Is the character string pattern to remove.
**Example:** Removing Leading Occurrences

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

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

For BROOKS R., the result is OOKS R.

For ABRAHAMS J., the result is ABRAHAMS J.

**UPCASE: Converting Text to Uppercase**

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.

**Syntax:** How to Convert Text to Uppercase

```
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.
**output**

Alphanumeric of type AnV or An

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 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.

**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

**Example:** Decoding XML-Encoded Characters

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

Example: XML-Encoding Characters

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 alphanumeric 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.

**LENV: Returning the Length of an Alphanumeric Field**

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

`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

**Example:** Finding the Length of an AnV Field

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

`LENV(TITLEV, 'I2')`

For ALICE IN WONDERLAND, the result is 19.
For SLEEPING BEAUTY, the result is 15.

**LOCASV: Creating a Variable Length Lowercase String**

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. 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
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

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

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. 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. 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.

sub_limit
Integer

Is the limit for the length of the substring.

output
Integer

Example:

Finding the Starting Position of a Variable Length Pattern

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

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

For “SMURFS, THE”, the result is 7.

For “SHAGGY DOG, THE”, the result is 11.
**SUBSTV: Extracting a Variable Length Substring**

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

```plaintext
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. 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
  - 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):

    ```plaintext
    If end > p_length or end < start, then outlen = 0. Otherwise, outlen = end - start + 1.
    ```
**Example:** Extracting a Variable Length Substring

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

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

For SMURFS, the result is SMU.

For SHAGGY DOG, the result is SHA.

**TRIMV: Removing Characters From a String**

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

```
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. 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. 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.

**plength_limit**
Integer

Is the limit for the length of the pattern.

**output**
Alphanumeric of type An or AnV

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 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**

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.

**source_string**
Alphanumeric of type An or AnV

is the string to convert to uppercase. 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.
UPCASEV: Creating a Variable Length Uppercase String

**output**
Alphanumeric of type An or AnV

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

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

```
UPCASEV(15, LAST_NAME_MIXED, 'A15V')
```

For Banning, the result is BANNING.
For McKnight, the result is MCKNIGHT.
Character Functions for DBCS Code Pages

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

```
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

**Example:** Using DCTRAN to Translate Double-Byte Characters

In the following:

\[ \text{DCTRAN}(8, 'A\text{\u65f6}A\text{\u6c44}本\text{\u65f6}語', 177, 70, A8) \]

For `A\text{\u65f6}A\text{\u6c44}本\text{\u65f6}語`, the result is `AFA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:

`inlength`

Integer

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

`source_string`

Alphanumeric

Is the string to edit.

`mask_length`

Integer

Is the number of characters in mask.

`mask`

Alphanumeric

Is the string of mask characters.

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

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

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

`output`

Alphanumeric

**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$9$-かきくけこ', 'A30')
The result is あいうえお-かきくけこ.
```
DSTRIP: Removing a Single-Byte or Double-Byte Character From a String

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$9-ABCDE','A20')
The result is aiueo-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 `outfield`.

- **source_string**
  - Alphanumeric
  - Is the string from which the character will be removed.

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

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

- **output**
  - Alphanumeric
Example:  Removing a Double-Byte Character From a String

In the following:

\[ \text{DSUBSTR}(9,'A日A本B語', '日', 9) \]

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

\[ \text{DSUBSTR}(\text{inlength}, \text{source_string}, \text{start}, \text{end}, \text{sublength}, \text{output}) \]

where:

\text{inlength}

Integer

Is the length of the source string in bytes. 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.

\text{source_string}

Alphanumeric

Is the string from which the substring will be extracted.

\text{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 \text{end}, the function returns spaces.

\text{end}

Integer

Is the ending position (in number of characters) of the substring. If this argument is less than \text{start} or greater than \text{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

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 いう
```

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>
</tbody>
</table>
### Conversion Type

<table>
<thead>
<tr>
<th>Conversion Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>'ZHNHKANA'</td>
<td>Converts Katakana from Zenkaku to Hankaku.</td>
</tr>
<tr>
<td>'ZHNNSPACE'</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>

**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

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

For a b c, the result is A B C.
JPTRANS: Converting Japanese Specific Characters

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 「べースボール。」, the result is 「べースボール。」

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.

JPTRANS('ZNHNNSIGN', 20, Symbol_DBCS_Field, 'A20')

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

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

For 「べースボール。」, the result is 「べースボール。」

JPTRANS('ZNHNNSPACE', 20, Zenkaku_Katakana_Field, 'A20')

For ア イ ウ, the result is ア イ ウ

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

For あいう, the result is ア イ ウ
JPTRANS('KATAHIRA', 20, Zenkaku_Katakana_Field, 'A20')

For アイウ, the result is あいう

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)
Fullwidth Backslash \ (U+FF3C) -> Backslash \ (U+005C)
Fullwidth Left Corner Bracket (U+300C) -> Halfwidth Left Corner Bracket (U+FF62)
Fullwidth Right Corner Bracket (U+300D) -> Halfwidth Right Corner Bracket (U+FF63)
Fullwidth Middle Dot · (U+30FB) -> Halfwidth Katakana Middle Dot · (U+FF65)

HNZNKANA and ZNHNKANA focus on the conversion of Katakana
They convert not only letters, but also punctuation symbols on the following list:
Fullwidth Ideographic Comma (U+3001) <-> Halfwidth Ideographic Comma (U+FF64)
Fullwidth Ideographic Full Stop (U+3002) <-> Halfwidth Ideographic Full Stop (U+FF61)
Fullwidth Left Corner Bracket (U+300C) <-> Halfwidth Left Corner 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('ZNHNSIGN', 20, Symbol_DBCS_Field, 'A20'), 'A20')

HNZNSPACE and ZNHNSPACE focus on the conversion of a space (blank character).
Currently only conversion between U+0020 and U+3000 is supported.

**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. 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.

**output**

Alphanumeric

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:

```
COUNTRY_CUT/A4 = KKFCUT(10, COUNTRY, '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.

- **length**
  
  Integer
  
  Is the length of the source string in bytes. 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

**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):

\[
\text{COUNTRY\_DEL/A10} = \text{SFTDEL} (\text{COUNTRY}, 10, 'A10');
\]
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>ｂｂＡメ לג</td>
</tr>
<tr>
<td>日本</td>
<td>イ</td>
</tr>
<tr>
<td>イタリア</td>
<td>ｂｂイメセ</td>
</tr>
<tr>
<td>ドイツ</td>
<td>ｂｂ</td>
</tr>
<tr>
<td>フランス</td>
<td>ｂｂ[ ] לג</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*

\[
\text{SFTINS}(\text{source\_string, length, output})
\]

where:

- **source\_string**
  Alphanumeric
  
  Is the string into which the shift code will be inserted.

- **length**
  Integer
  
  Is the length of the source string in bytes. 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

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):

\[
\text{COUNTRY_INS/A10} = \text{SFTINS} (\text{COUNTRY_DEL, 10, 'A10'})
\]

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] [f] [A] [M] [N]</td>
<td>イギリス</td>
</tr>
<tr>
<td>日本</td>
<td>[i] [k]</td>
<td>日本</td>
</tr>
<tr>
<td>イタリア</td>
<td>[b] [j] [M] [a]</td>
<td>イタリア</td>
</tr>
<tr>
<td>ドイツ</td>
<td>[f] [b]</td>
<td>ドイツ</td>
</tr>
<tr>
<td>フランス</td>
<td>[f] [f] [f] [N]</td>
<td>フランス</td>
</tr>
</tbody>
</table>
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.

In this chapter:

- CHECKMD5: Computing an MD5 Hash Check Value
- CHECKSUM: Computing a Hash Sum
- 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
- LAST: Retrieving the Preceding Value
- LOOKUP: Retrieving a Value From a Cross-referenced Data Source

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

```
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

CHECKMD5 calculates a fixed length MD5 hash check value, and HETYPE 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

CHECKSUM calculates a checksum hash value.

CHECKSUM(PRODUCT_CATEGORY)

For Accessories, the result is -830549649.

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

\[
\text{DB_EXPR(native\_SQL\_expression)}
\]

where:

- \text{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_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
```
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
```

**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 FOCUS
request, such as in a DEFINE or a WHERE phrase.

The target file can be any data source that FOCUS can read. Depending on the data sources
accessed and the components in the request, either FOCUS or an RDBMS will process the
comparison of values.

If FOCUS 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 FOCUS 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.

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 FOCUS 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 FOCUS processes the comparison, and the target data contains characters that FOCUS 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.

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:

```sql
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:

```sql
SEGNAME=RETAIL_SUBQUERY, CONNECTION=CON1, DATASET=RETAIL_SUBQUERY.SQL, $
```

144 Information Builders
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_RETAIL
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
```

Functions Reference 145
The `empvalues.mas` Master File describes the data in the `empvalues.ftm` file

```plaintext
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.

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:

```sql
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:

```sql
SET DB_INFILE = {DEFAULT|EXPAND_ALWAYS|EXPAND_NEVER}
```

In a TABLE request, issue the following command:

```sql
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**

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

\[
\text{DB_LOOKUP}(\text{look}_m\text{f}, \text{srcfld}_1, \text{lookfld}_1, \text{srcfld}_2, \text{lookfld}_2, \ldots, \text{returnfld});
\]

where:

\[
\text{look}_m\text{f}
\]

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 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**

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 *female*). 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.

**Syntax:** How to Supply Values in the Function

\[
\text{DECODE } \text{fieldname} (\text{code1 result1 code2 result2} ... [\text{ELSE default }]);
\]

\[
\text{DECODE } \text{fieldname} (\text{filename} ... [\text{ELSE default }]);
\]

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 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

DECODE returns the state abbreviation for PLANT.

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

For BOS, the result is MA.

For DAL, the result is TX.

For LA, the result is CA.

FIND: Verifying the Existence of a Value in a Data Source

The FIND function determines if an incoming data value is in an indexed FOCUS data source
field. The function sets a temporary field to a non-zero value if the incoming value is in the data
source field, and to 0 if it is not. A value greater than zero confirms the presence of the data
value, not the number of instances in the data source field.
You can also use FIND in a VALIDATE command to determine if a transaction field value exists in another FOCUS data source. If the field value is not in that data source, the function returns a value of 0, causing the validation test to fail and the request to reject the transaction.

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

**Limit:** FIND does not work on files with different DBA passwords.

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**

```
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.

- **file**
  - Is the name of the indexed FOCUS data source.

**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 an Indexed Field**

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

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

**LAST: Retrieving the Preceding Value**

The LAST function retrieves the preceding value for a field.
The effect of LAST depends on whether it appears in an extract or load transformation:

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

**Syntax:**

**How to Retrieve the Preceding Value**

```
LAST fieldname
```

where:

- `fieldname`
  - Alphanumeric or Numeric
  - Is the field name.

**Note:** LAST does not use an `output` argument.

**Example:**

Retrieving the Preceding Value

LAST retrieves the previous value of DEPARTMENT:

```
LAST DEPARTMENT
```

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

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

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

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

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

  For example, LOOKUP retrieves all the fields from the segment
  \[
  \text{RTN} = \text{LOOKUP}(\text{SEG.DATE_ATTEND});
  \]

- 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

```
LOOKUP(field);
```

where:

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

LOOKUP (DATE_ENROLL)
Simplified Date and Date-Time Functions

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. Literal date-time values can be used with the DT function.

All arguments can be either literals, field names, or amper variables.

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:**

```plaintext
How to Return the Current Date

DT_CURRENT_DATE()
```

**Example:**

**Returning the Current Date**

DT_CURRENT_DATE returns the current date.

```plaintext
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:**

```plaintext
How to Return the Current Date and Time

DT_CURRENT_DATETIME(component)
```

where:

- `component`
  
  Is one of the following time precisions.

  - SECOND.
  - MILISECOND.
  - 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**

DT_CURRENT_DATETIME returns the current date and time to microsecond precision.

```plaintext
DT_CURRENT_DATETIME(MICROSECOND)
```
For September 8, 2016 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

```
DT_CURRENT_TIME(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 Time

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

```
DTADD(date, component, increment)
```
DTADD: Incrementing a Date or Date-Time Component

where:

date
Date or date-time
Is the date or date-time value to be incremented.

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

DTADD adds three days to the employee date of birth:

```
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, DDIFF 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:

\[\text{end\_date}\]
- Date or date-time
  - Is the ending 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.

\[\text{start\_date}\]
- Date or date-time
  - Is the starting 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.

\[\text{component}\]
- 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).
DTIME: Extracting Time Components From a Date-Time Value

- HOUR (0-23).
- MINUTE (0-59).
- SECOND (0-59).

*Example: Returning the Number of Years Between Two Dates*

DTDIFF calculates employee age when hired:

\[ \text{DTDIFF(START_DATE, DATE_OF_BIRTH, YEAR)} \]

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: How to Extract a Time Component From a Date-Time Value**

\[ \text{DTIME(datetime, component)} \]

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.

- **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

DTIME extracts the TIME component from the data-time value 2018/01/17 05:45:22.777888.

\[ \text{DTIME(DT(2018/01/17 05:45:22.777888), TIME)} \]

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:** How to Return a Date or Date-Time Component in Integer Format

\[ \text{DTPART(date, component)} \]

where:

- \textit{date}
  - Date or date-time
  - Is the date in standard date or date-time format.

- \textit{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

DTPART extracts the quarter from the employee start date:

DTPART(START_DATE, 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: How to Return the First or Last Date of a Date Period

DTRUNC(date_or_timestamp, date_period)

where:

date_or_timestamp
Date or date-time
  Is the date or timestamp of interest.

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**

DTRUNC returns the first date of the quarter given the date of birth:

```
DTRUNC(DATE_OF_BIRTH, QUARTER)
```

For 1993/03/27, the result is 1993/03/01.

**Example:** **Using the Start of Week Parameter for DTRUNC**

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 Last Day of a Week**

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
- F IQTR: 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
- J ULDAT: 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.

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 167.

- 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 172.

For detailed information on each standard date function, see:

- DATEADD: Adding or Subtracting a Date Unit to or From a Date on page 173
- DATECVT: Converting the Format of a Date on page 175
- DATEDIF: Finding the Difference Between Two Dates on page 177
- DATEMOV: Moving a Date to a Significant Point on page 179
- DATETRAN: Formatting Dates in International Formats on page 183
- FIYR: Obtaining the Financial Year on page 199
- FIQTR: Obtaining the Financial Quarter on page 201
- FIYYQ: Converting a Calendar Date to a Financial Date on page 203
- TODAY: Returning the Current Date on page 204

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

```plaintext
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:

```plaintext
SET BUSDAYS = S_TW_FS
```

**Syntax:** How to View the Current Setting of Business Days

```plaintext
? 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 170.

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 168.
2. Save the file.

If you are not using the default naming convention, see How to FILEDEF or DYNAM the Holiday File on page 170. If you are using the default naming convention, use the following instructions:

In Windows and UNIX: The file must be HDAYxxxx.ERR

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
```

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(HDAYxxxx) 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 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 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 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 = _MTWTF_
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:

```
-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:

```
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

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.
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.

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.

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

```sql
DATEADD(date, 'component', increment)
```

where:

```sql
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.

**DATECVT: Converting the Format of a Date**

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.
**Syntax:** How to Convert a Date Format

```
DATECVT(date, 'in_format', output)
```

where:

**date**

Date

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

**in_format**

Alphanumeric

Is the format of the date 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 `in_format` functions as an offset from the base date of a YYMD field (12/31/1900).

**output**

Alphanumeric

Is the output 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.

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

**Note:** DATECVT does not use an output format; it uses the format of the argument `output_format` for the result.
DATEDIF: Finding the Difference Between Two Dates

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 168 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.

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.
Syntax: How to Find the Difference Between Two Dates

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:
MYDIFF/I8 = NEW_YYM_DT/YYM - OLD_YYM_DT;

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

**DATEMOV: Moving a Date to a Significant Point**

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

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
```

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:

```
DEFINE FILE GGSales
DT/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, 09/01/1996</td>
<td>FRI, 08/30/1996</td>
<td>MON, 09/02/1996</td>
</tr>
<tr>
<td>SAT, 03/01/1997</td>
<td>FRI, 02/28/1997</td>
<td>MON, 03/03/1997</td>
</tr>
<tr>
<td>TUE, 04/01/1997</td>
<td>TUE, 04/01/1997</td>
<td>WED, 04/02/1997</td>
</tr>
<tr>
<td>THU, 05/01/1997</td>
<td>THU, 05/01/1997</td>
<td>FRI, 05/02/1997</td>
</tr>
<tr>
<td>SUN, 06/01/1997</td>
<td>FRI, 05/30/1997</td>
<td>MON, 06/02/1997</td>
</tr>
<tr>
<td>MON, 09/01/1997</td>
<td>MON, 09/01/1997</td>
<td>TUE, 09/02/1997</td>
</tr>
<tr>
<td>WED, 10/01/1997</td>
<td>WED, 10/01/1997</td>
<td>THU, 10/02/1997</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:

```
DEFINE FUNCTION BOWK(THEDATE/MDYY, WEEKSTART/A10)
DAYOFWEEK/W=THEDATE;
DAYNO/I1=IF DAYOFWEEK EQ 7 THEN 0 ELSE DAYOFWEEK;
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>

**DATETRAN: Formatting Dates in International Formats**

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>
<tr>
<td>' (DMDY) '</td>
<td>Day component followed by month followed by two-digit year.</td>
</tr>
<tr>
<td>' (MDYY) '</td>
<td>Month component followed by day followed by four-digit year.</td>
</tr>
<tr>
<td>' (MDY) '</td>
<td>Month component followed by day followed by two-digit year.</td>
</tr>
<tr>
<td>' (MD) '</td>
<td>Month component followed by day (derived from three-component date by ignoring year component).</td>
</tr>
<tr>
<td>' (DM) '</td>
<td>Day component followed by month (derived from three-component date by ignoring year component).</td>
</tr>
</tbody>
</table>
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>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>Format Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</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>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>Format Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</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>
<tr>
<td>XP</td>
<td>Includes an abbreviated day-of-the-week name at the end of the displayed date, all uppercase, followed by a period.</td>
</tr>
<tr>
<td>x</td>
<td>Includes an abbreviated day-of-the-week name at the end of the displayed date with no punctuation. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).</td>
</tr>
<tr>
<td>XR</td>
<td>Includes a full day-of-the-week name at the end of the displayed date. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).</td>
</tr>
<tr>
<td>Format Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>xp</strong></td>
<td>Includes an abbreviated day-of-the-week name at the end of the displayed date followed by a period. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).</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><strong>B</strong></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>/</td>
<td>Uses a slash (/) as the component delimiter.</td>
</tr>
<tr>
<td></td>
<td>Omits component delimiters.</td>
</tr>
<tr>
<td><strong>K</strong></td>
<td>Uses appropriate Asian characters as component delimiters.</td>
</tr>
<tr>
<td><strong>c</strong></td>
<td>Places a comma (,) after the month name (following T, Tp, TR, t, tp, or tr). Places a comma and blank after the day name (following W, Wp, WR, w, wp, or wr). Places a comma and blank before the day name (following X, XR, x, or xr).</td>
</tr>
<tr>
<td>Format Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</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
- 'EN' English
- 'ES' Spanish
- 'FI' Finnish
- 'FR' French
- 'EL' Greek
- 'IW' Hebrew
- 'IT' Italian
- 'JA' Japanese
'KO' Korean
'LT' Lithuanian
'NL' Dutch
'NO' Norwegian
'PO' Polish
'PT' Portuguese
'RU' Russian
'SV' Swedish
'TH' Thai
'TR' Turkish
'TW' Chinese (Traditional)
'ZH' Chinese (Simplified)

**outlen**

Numeric

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

**output**

Alphanumeric

**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 DTLNGlng 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.

- 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:

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

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

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

END

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

```
FORMAT wr

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

Tuesday         Monday
martes           lunes
mardi           lundi
Dienstag        Montag
```

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

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

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

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

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

```
FORMAT Btdo
Blank-delimited (B)
Abbreviated month name, default case (t)
Zero-suppress day number, end with suffix (do)
English

Jan 4th 2005       Mar 2nd 2005
```

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

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

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

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

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

```
FORMAT Btdp
Blank-delimited (B)
Abbreviated month name, default case (t)
Zero-suppress day number, end with period (dp)
German
```


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 ;
DATEYMYD/YYMDW=TRANS1 ;
DATEYMYD2/YYMDW=TRANS2 ;

OUT4A/A30 = DATETRAN(DATEYMYD, '(DMYY)', '(Bwrtr)', 'FR', 30, 'A30');
OUT4B/A30 = DATETRAN(DATEYMYD2, '(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 SET PAGE-NUM OFF
END
```
The output is:

![Table displaying formatted dates in Spanish]

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:

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

dateW/w=TRANS1  
DATEW2/W=TRANS2  
DATEYYMD/YYMDW=TRANS1    
DATEYYMD2/YYMDW=TRANS2   

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

TABLE FILE VIDEOTRK
HEADING
"FORMAT Bwrctrde"
"
"Blank-delimited (B)"
"Full day of week name at beginning of date, default case (wr)"
"Comma after day name (c)"
"Full month name, default case (tr)"
"Zero-suppress day number (d)"
"de between day and month and between month and year (e)"
"Spanish"
"
SUM OUT5A AS '' OUT5B AS '' TRANSDATE NOPRINT
ON TABLE SET PAGE-NUM OFF
END
```
The following request prints a date in Japanese characters with a full month name at the beginning, in the default case and with zero suppression:

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

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

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

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

```
FORMAT Ktrd

Japanese characters (K in conjunction with the language code JA)
Full month name at beginning of date, default case (tr)
Zero-suppress day number (d)
Japanese

2005年1月4日  2005年3月2日
```

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

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

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

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

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

**Syntax:**

*How to Obtain the Financial Year*

```plaintext
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

FIYR obtains the financial year for PERIOD, which has format YYM:

\[ \text{FIYR(PERIOD, 'M', 4, 1, 'FYE', 'YY')}; \]

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.

**Syntax:** How to Obtain the Financial Quarter

\[ \text{FIQTR(inputdate, lowcomponent, startmonth, startday, yrnumbering, output)} \]

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 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

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.

Syntax:

How to Convert a Calendar Date to a Financial Date

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.

**output**

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**

FIYYQ returns the financial date in format YQ that corresponds to START_DATE (format YYMD);

\[
FIYYQ(\text{START\_DATE}, \text{'D'}, 10, 1, \text{'FYE'}, \text{'YQ'})
\]

For 1997/10/01, the result is 98 Q1.

For 1996/07/30, the result is 96 Q4.

**TODAY: Returning the Current Date**

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.

**Syntax:** How to Retrieve the Current Date

```
TODAY(output)
```

where:

**output**

Alphanumeric, at least A8

The following apply:

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

**Example:** Retrieving the Current Date

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 alphanumerical format.

For detailed information on each legacy date function, see:

- **AYM:** Adding or Subtracting Months
- **AYMD:** Adding or Subtracting Days on page 206
- **CHGDAT:** Changing How a Date String Displays on page 207
- **DA Functions:** Converting a Legacy Date to an Integer on page 210
- **DMY, MDY, YMD:** Calculating the Difference Between Two Dates on page 211
- **DOWK and DOWKL:** Finding the Day of the Week on page 212
- **DT Functions:** Converting an Integer to a Date on page 213
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. The old versions of these functions may not work correctly with dates after December 31, 1999. However, in some cases you may want to use the old version of a function, for example, if you do not use year 2000 dates. You can "turn off" the current version with the DATEFNS parameter.

**Syntax:** How to Activate Old Legacy Date Functions

```
SET DATEFNS = {ON|OFF}
```

where:

- **ON**
  
  Activates the function that supports dates for the year 2000 and later. ON is the default value.

- **OFF**
  
  Deactivates a function that supports dates for the year 2000 and later.

**AYMD: Adding or Subtracting Days**

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

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

```
AYMD(indate, days, output)
```

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).
**days**

Integer

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

**output**

I6, I6YMD, I8, or I8YYMD

Is the same format as \textit{indate}.

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

**Example:**  Adding Days to a Date

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

\[
\text{AYMD}(\text{HIRE\_DATE}, \text{35}, \text{"I6YMD"})
\]

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**

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>
</tbody>
</table>
### Syntax: How to Change the Date Display String

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.

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

Example: Converting the Date Display From YMD to MDYYX

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

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

DA Functions: Converting a Legacy Date to an Integer

The DA functions convert a legacy date to the number of days between it and a base date. 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 213.

There are six DA functions; each one accepts a date in a different format.
Syntax: How to Convert a Date to an Integer

\[ \text{function}(\text{indate}, \text{output}) \]

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. 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

Example: Converting Dates and Calculating the Difference Between Them

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

\[ \text{DAYMD(DAT_INC, 'I8')} - \text{DAYMD(HIRE_DATE, 'I8')} \]

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

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

\[ \text{function}( \text{from\_date}, \text{to\_date} ) \]

where:

\text{function}

Is one of the following:

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

\text{from\_date}

I, P, or A format with date display options.

Is the beginning legacy date.

\text{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.

Example: Calculating the Number of Days Between Two Dates

YMD calculates the number of days between the dates in HIRE\_DATE and DAT\_INC.

\text{YMD}(\text{HIRE\_DATE}, \text{DAT\_INC})

DOWK and DOWKL: Finding the Day of the Week

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

Syntax: How to Find the Day of the Week

\{\text{DOWK}\mid\text{DOWKL}\}(\text{indate}, \text{output})

where:

\text{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.
Example: Finding the Day of the Week

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

\[
\text{DOWK(HIRE\_DATE, 'A4')}
\]

For 80/06/02, the result is MON.
For 82/08/01, the result is SUN.

DT Functions: Converting an Integer to a Date

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

Syntax: How to Convert an Integer to a Date

\[
\text{function(number, output)}
\]

where:

- **function**
  - Is one of the following:
    - \text{DTDMY} converts a number to a day-month-year date.
    - \text{DTDYM} converts a number to a day-year-month date.
    - \text{DTMDY} converts a number to a month-day-year date.
    - \text{DTMYD} converts a number to a month-year-day date.
    - \text{DTYDM} converts a number to a year-day-month date.
    - \text{DTYMD} converts a number to a year-month-day date.

- **number**
  - 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.
Example: Converting an Integer to a Date

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

\[
\text{DTMDY(NEWF, 'I8MDYY')}\]

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

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.

Syntax: How to Convert From Julian to Gregorian Format

\[
\text{GREGDT(indate, output)}
\]

where:

\[
\text{indate}
\]

I5 or I7
Is the Julian date. If the date is invalid, the function returns a 0 (zero).

\[
\text{output}
\]

I6, I8, I6YMD, or I8YYMD

Example: Converting From Julian to Gregorian Format

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**

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.

**Syntax: How to Convert From Gregorian to Julian Format**

```plaintext
JULDAT(indate, output)
```

where:

- **indate**  
  I6, I8, I6YMD, I8YYMD  
  Is the legacy date to convert.

- **output**  
  I5 or I7

**Example: Converting From Gregorian to Julian Format**

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

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

For 82213, the result is 19820801.
For 82004, the result is 19820104.

**YM: Calculating Elapsed Months**

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

\[ \text{YM}(\text{fromdate}, \text{todate}, \text{output}) \]

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

**Tip:** If fromdate or todate is in integer year-month-day format (I6YMD or I8YMD), 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

YM calculates the difference between HIRE_MONTH and MONTH_INC and stores the results in a column with the format I3.

\[ \text{YM}(\text{HIRE\_MONTH}, \text{MONTH\_INC}, 'I3') \]
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: 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. 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

\[
\text{SET DATEFORMAT} = \text{option}
\]

where:

\text{option}

Can be one of the following: MDY, DMY, YMD, or MYD. MDY is the default value for the U.S. English format.

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 219. 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 219.

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

\[
\text{SET WEEKFIRST} = \text{value}
\]

where:

\text{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, HYWD, 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>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>0-9999999</td>
</tr>
<tr>
<td>nanosecond</td>
<td>ns</td>
<td>0-9999999999</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).

HADD: Incrementing a Date-Time Value

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

**Syntax:** How to Increment a Date-Time Value

\[
\text{HADD}(\text{datetime}, '\text{component}', \text{increment}, \text{length}, \text{output})
\]

where:

- **datetime**
  - Date-time
  - Is the date-time value to be incremented.

- **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 222.

  **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.

- **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).
**Example: Incrementing a Date-Time Value**

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

\[
\text{HADD(DT}(2001/09/11 \ 08:54:34), 'MONTH', 30, 8, 'HYYMDS')
\]

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

**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/HMDYYSb = 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

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**

\[ \text{HCNVRT}(\text{datetime}, ' (\text{format})', \text{length}, \text{output}) \]

where:

- **datetime**
  - Date-time
  - Is the date-time value to be converted.

- **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. If **length** is smaller than the number of characters needed to display the alphanumeric field, the function returns a blank.

- **output**
  - Alphanumeric

**Example:**

**Converting a Date-Time Value to Alphanumeric Format**

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

\[ \text{HCNVRT(DTCUR, ' (HMDYYS)', 20, 'A20')} \]

The function returns the string '03/26/2004 14:25:58' that is assignable to an alphanumeric variable.
**HDATE: Converting the Date Portion of a Date-Time Value to a Date Format**

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

**Syntax:**

```
HDATE(datetime, output)
```

where:

- `datetime`  
  Date-time  
  Is the date-time value to be converted.

- `output`  
  Date

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

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

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

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

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

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

\[
\text{HDIFF}(\text{end}_\text{dt}, \text{start}_\text{dt}, \text{'component'}, \text{output})
\]

where:

- **end_dt**
  - Date-time
  - Is the date-time value to subtract from.

- **start_dt**
  - Date-time
  - Is the date-time value to subtract.

- **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

**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:

\[
\text{DIFDAY/I6 } = \text{ HDIFF(DTCUR, DT(2004/02/16), 'DAY', 'D6.0')}
\]

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

\[
\text{DIFSEC/19} = \text{HDIF(DTCUR, DT(2004 February 16), 'SECOND', 'D9.0')}
\]

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

**HDTTM: Converting a Date Value to a Date-Time Value**

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

**Syntax:** How to Convert a Date Value to a Date-Time Value

\[
\text{HDTTM(date, length, output)}
\]

where:

**date**

Date

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

**length**

Integer

Is the 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. 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:

\[ \text{TS/HYYMDS} = \text{HDTTM('February 16 2004', 8, TS)} \]

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

**HGETC: Storing the Current Local Date and Time in a Date-Time Field**

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

\[ \text{HGETC}(\text{length}, \text{output}) \]

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.

**Example:** Storing the Current Date and Time as a Timestamp

This example,

\[ \text{HGETC}(8, \text{'HYYMDS'}) \]

creates a timestamp representing the current date and time.
HGETZ: Storing the Current Coordinated Universal Time in a Date-Time Field

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.

**Example:** Storing the Current Universal Date and Time as a Timestamp

This example,

```
HGETZ(8, 'HYYMDS')
```

creates a timestamp representing the current date and time.
**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:

- LOCALTIME  2015/05/12 12:47:04
- UTCTIME    2015/05/12 16:47:04
- MINUTES     2015/05/12 -240
- ZONE        -4

**HHMMSS: Retrieving the Current Time**

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

**Syntax: How to Retrieve the Current Time**

```plaintext
HHMMSS(output)
```

where:

- **output**

  Alphanumeric, at least A8
**Example:** Retrieving the Current Time

This example,

\texttt{HMMSS('A10')}

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

**HHMS: Converting a Date-Time Value to a Time Value**

The HHMS function converts a date-time value to a time value.

**Syntax:** How to Convert a Date-Time Value to a Time Value

\texttt{HHMS} (\texttt{datetime, length, output})

where:

\texttt{datetime}

\texttt{Date-time}

\texttt{Is the date-time value to be converted.}

\texttt{length}

\texttt{Numeric}

\texttt{Is the length of the returned time value. Valid values are:}

\begin{itemize}
  \item \texttt{8} indicates a time value that includes milliseconds.
  \item \texttt{10} indicates a time value that includes microseconds.
  \item \texttt{12} indicates a time value that includes nanoseconds.
\end{itemize}

\texttt{output}

\texttt{Time}

**Example:** Converting a Date-Time Value to a Time value

HHMS converts the date-time field TRANSDATE to a time value with format HHIS:

\texttt{HHMS(TRANSDATE, 8, 'HHIS')}\n
For 2000/06/26 05:45, the output is 05:45:00
HINPUT: Converting an Alphanumeric String to a Date-Time Value

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.

- **source\_string**
  - Alphanumeric
  - Is the string to be converted.

- **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.

**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.
HMIDNT: Setting the Time Portion of a Date-Time Value to Midnight

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

\[
\text{HMIDNT}(\text{datetime}, \text{length}, \text{output})
\]

where:

- **datetime**
  - Date-time
  - Is the date-time value whose time is to be set to midnight.

- **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.

**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:

\[
\text{DTM/HYYMDS} = \text{HINPUT}(14, \ '20040229 13:34:00', \ 8, \ \text{DTM});
\]

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

\[
\text{DTMIDNT/HMDYYS} = \text{HMIDNT}(\text{DTM}, \ 8, \ \text{DTMIDNT});
\]
HNAME: Retrieving a Date-Time Component in Alphanumeric Format

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

Syntax: 

```
HNAME(datetime, 'component', output)
```

where:

- **datetime**
  - Date-time
  - Is the date-time value from which a component value is to be extracted.

- **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 222.

- **output**
  - Alphanumeric, at least A2
  - 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, 'HYMDS'), '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, 'HYMDS'), 'HOUR', AHOUR);
```

HPART: Retrieving a Date-Time Component as a Numeric Value

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

\[
\text{HPART}(\text{datetime}, '\text{component}', \text{output})
\]

where:

- \textit{datetime}
  - Date-time
  - Is the date-time value from which the component is to be extracted.

- \textit{component}
  - Alphanumeric
  - Is the name of the component to be retrieved enclosed in single quotation marks. For a list of valid components, see \textit{Arguments for Use With Date and Time Functions} on page 222.

- \textit{output}
  - Integer

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

Assuming that the current time obtained by \text{HGETC} in the first parameter is 14:01:39, this example returns a whole number, 14, and assigns it to I\text{HOUR}:

\[
\text{I\text{HOUR}}/\text{I2} = \text{HPART} (\text{HGETC}(8, '\text{HYYMDS}'), '\text{HOUR}', \text{I\text{HOUR}});
\]

HSETPT: Inserting a Component Into a Date-Time Value

The \text{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.
component

Alphanumeric

Is the name of the component to be inserted enclosed in single quotation marks. See Arguments for Use With Date and Time Functions on page 222 for a list of valid components.

value

Integer

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

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.

Example: Inserting a Component Into a Date-Time Value

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

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

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

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

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.

*output*

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.)

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

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

NMICRO/D12.0 = HTIME(10, HGETC(10,©HYYMDS©), NMICRO);

**HTMTOTS: 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.

**Syntax:** How to Convert a Time to a Timestamp

HTMTOTS(time, length, output)
where:

\textit{time}

Date-Time

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

\textit{length}

Integer

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

\begin{itemize}
  \item 8 for input time values including milliseconds.
  \item 10 for input time values including microseconds.
  \item 12 for input time values including nanoseconds.
\end{itemize}

\textit{output\_format}

Date-Time

Is the timestamp whose date is set to current date, and whose time is copied from \textit{time}.

\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{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

\[ \text{HYYWD}(dtvalue, \text{output}) \]

where:

\( dtvalue \)

Date-time

Is the date-time value to be edited.

\( output \)

Alphanumeric

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:

\[ \text{ISODATE}/A10 = \text{HYYWD}(\text{TRANSDATE}, \ 'A10'); \]

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.
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
- **CTRLCHAR**: Returning a Non-Printable Control Character
- **EDIT2**: Converting a Numeric, Date, or Date-Time Value to a Character String
- **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

\[
\text{CHAR}(\text{number\_code})
\]
where:

### 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

CHAR returns a carriage control character in an ASCII environment.

`CHAR(13)`

### 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.
EDIT2: Converting a Numeric, Date, or Date-Time Value to a Character String

Example: Using the CTRLCHAR Function to Insert Control Characters Into a String

CTRLCHAR returns a carriage control character in an ASCII environment.

CTRLCHAR(CR)

EDIT2: Converting a Numeric, Date, or Date-Time Value to a Character String

The EDIT2 function converts a numeric, date, or date-time value to a character string based on a specified format. The format must be valid for the data supplied. The output is returned as variable length alphanumeric.

Syntax: How to Convert a Numeric, Date, or Date_Time Value to a Character String

EDIT2(in_value, 'format')

where:

in_value
  Numeric, date, or date-time
  Is any numeric value or a date in either standard date or date-time format. If the date is given in standard date format, all of its time components are assumed to be zero.

'format'
  Is a numeric, date, or date-time format enclosed in single quotation marks.

Example: Converting a Date to a Character String

EDIT2 converts DATE1 (YYMD format) to a character string in YYMtrD format.

EDIT2(DATE1, 'YYMtrD')

For 2012/02/02, the result is 2012, February 2.

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:

\textit{in\_value}

\textit{in\_value} is an alphanumerical or integer field, constant, or expression.

\textbf{Example:} Returning a Hexadecimal View

\texttt{HEXTYPE} returns a hexadecimal view of \texttt{COUNTRY\_NAME}.

\texttt{HEXTYPE(COUNTRY\_NAME)}

For Argentina, the result is 4172656E74696E61.

\textbf{PHONETIC: Returning a Phonetic Key for a String}

\texttt{PHONETIC} calculates a phonetic key for a string, or a null value on failure. Phonetic keys are useful for grouping alphanumerical 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.

\texttt{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.

\textbf{Note:} Metaphone is not optimized in generated SQL. Therefore, if you need to optimize the request for an SQL DBMS, the \texttt{SOUNDEX} setting should be used.

Soundex is a legacy phonetic algorithm for indexing names by sound, as pronounced in English.

\textbf{Syntax:} How to Return a Phonetic Key

\texttt{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

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

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

TO_INTEGER converts the character string '56.78' to an integer.

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

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

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

TO_NUMBER converts the string '56.78' to a number with one decimal place.

\[ \text{TO_NUMBER('56.78')} \]

The result is 56.8.
TO_NUMBER: Converting a Character String to a Numeric Value
Format Conversion Functions

Format conversion functions convert fields from one format to another.

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 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

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

\[ \text{ATODBL}(\text{source\_string}, \text{length}, \text{output}) \]

where:

\[ \text{source\_string} \]

Alphanumeric
Is the string consisting of digits and, optionally, one sign and one decimal point to be converted.

**length**  
Alphanumeric

Is the length of the source string in bytes. This can be a numeric constant, or a field or variable that contains the value. If you specify a numeric constant, enclose it in single quotation marks, for example '12'.

**output**  
Double precision floating-point

**Example:** Converting an Alphanumeric Field to Double-Precision Format

ATOBL converts EMP_ID into double-precision format.

```
ATOBL(EMP_ID, '09', 'D12.2')
```

For 112847612, the result is 112,847,612.00.

For 117593129, the result is 117,593,129.00.

**EDIT: Converting the Format of a Field**

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 92.

**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.

**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.

**Syntax:** How to Convert Fields Using FPRINT

```
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.
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.

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')
```

FTOA: Converting a Number to Alphanumeric Format

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

```
FTOA(number, '(format)', output)
```

where:

- `number`
  
  Numeric F or D (single and double precision floating-point)
  
  Is the number to be converted.
**format**

Alphanumeric

Is the format of the number to be converted 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.

**output**

Alphanumeric

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.

`FTOA(GROSS, ©(D12.2)©, ©A15©)`

For $1,815.00, the result is 1,815.00.
For $2,255.00, the result is 2,255.00.

**HEXBYT: Converting a Decimal Integer to a Character**

The HEXBYT function obtains the ASCII, EBCDIC, or Unicode character equivalent of a decimal integer, depending on your configuration and operating environment. It 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.
Syntax: **How to Convert a Decimal Integer to a Character**

\[ \text{HEXBYT}(\text{decimal\_value}, \text{output}) \]

where:

- **decimal\_value**: Integer
  
  Is the decimal integer to be converted to a single character. In non-Unicde environments, a value greater than 255 is treated as the remainder of `decimal\_value` divided by 256.

- **output**: Alphanumeric

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.

\[ \text{HEXBYT}(\text{LAST\_INIT\_CODE}, 'A1') \]

On an ASCII platform, for 83, the result is S.

On ASCII platform, for 74, the result is J.

**ITONUM: Converting a Large Number to Double-Precision Format**

The ITONUM function converts a large number in a non-FOCUS data source from special long integer to double-precision format.

This is useful for some programming languages and some non-FOCUS data storage systems that use special long integers, which do not fit the regular integer format (four bytes in length) supported in the synonym, and, therefore, require conversion to double-precision format.

You must specify how many of the right-most bytes in the input field are significant. The result is an 8-byte double-precision field.

Syntax: **How to Convert a Large Binary Integer to Double-Precision Format**

\[ \text{ITYD}(\text{maxbytes}, \text{infield}, \text{output}) \]

where:

- **maxbytes**: Numeric

Information Builders
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.

\textit{infield}

A8

Is the field that contains the number. Both the USAGE and ACTUAL formats of the field must be A8.

\textit{output}

Double precision floating-point (Dn)

\textbf{Example: Converting a Large Binary Integer to Double-Precision Format}

ITONUM converts BINARYFLD to double-precision format.

\texttt{ITONUM(6, BINARYFLD, 'D14')}

\textbf{ITOPACK: Converting a Large Binary Integer to Packed-Decimal Format}

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.

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).

\textbf{Limit:} For a field defined as 'PIC 9(15) COMP' or the equivalent (15 significant digits), the maximum number that can be converted is 167,744,242,712,576.
Syntax: How to Convert a Large Binary Integer to Packed-Decimal Format

ITOPACK(maxbytes, infield, output)

where:

maxbytes

Numeric

Is the maximum number of bytes in the 8-byte 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

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.

ITOPACK(6, BINARYFLD, 'P14.4')

ITOZ: Converting a Number to Zoned Format

The ITOZ function converts a number in numeric format to zoned-decimal format. Although a request cannot process zoned numbers, it can write zoned fields to an extract file for use by an external program.
Syntax: How to Convert a Number to Zoned Format

\[ \text{ITOZ}(\text{length}, \text{in\_value}, \text{output}) \]

where:

\text{length} \\
\text{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} \\
\text{Numeric} \\
Is the number to be converted. The number is truncated to an integer before it is converted.

\text{output} \\
\text{Alphanumeric}

Example: Converting a Number to Zoned Format

\text{ITOZ} \text{converts CURR\_SAL to zoned format.}

\text{ITOZ}(8, \text{CURR\_SAL}, 'A8')

PCKOUT: Writing a Packed Number of Variable Length

The PCKOUT function writes a packed-decimal number of variable length to an extract file. When a request saves a packed 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

\[ \text{PCKOUT}(\text{in\_value}, \text{length}, \text{output}) \]

where:

\text{in\_value} \\
\text{Numeric} \\
Is the input value. It can be in packed, integer, single- or double-precision floating point format. If it is not in integer format, it is rounded to the nearest whole number.
PTOA: Converting a Packed-Decimal Number to Alphanumeric Format

Example: Writing a Packed Number of Variable Length

PCKOUT converts CURR_SAL to a five-byte packed format.

PCKOUT(CURR_SAL, 5, 'A5')

PTOA: Converting a Packed-Decimal Number to Alphanumeric Format

The PTOA function converts a 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.

Syntax: 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.

format

Alphanumeric
Is the format of the number enclosed in parentheses.

**output**

Alphanumeric

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')

**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:**  How to Convert an MSSQL or Sybase Timestamp Column to Packed Decimal

TSTOPACK(*tscol*, *output*);

where:

*tscol*

A16

Is the timestamp column to be converted.

*output*

P21
Example: Converting a Microsoft SQL Server Timestamp Column to Packed Decimal

The Master File for the TSTEST data source follows. The field TS represents the TIMESTAMP column:

```plaintext
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:

```plaintext
TNUM/P21=TSTOPACK(TS,'©P21©);
```

For 0000000000007815, the result is 30741.

For 0000000000007816, the result is 30742.

UFMT: Converting an Alphanumeric String to Hexadecimal

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: How to Convert an Alphanumeric String to Hexadecimal

```plaintext
UFMT(source_string, length, output)
```

where:

- `source_string` Alphanumeric
  - Is the alphanumeric string to convert.

- `length` Integer
  - Is the number of characters in `source_string`.

- `output` Alphanumeric
  - 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.

```
UFMT(JOBCODE, 3, 'A6')
```

For A01, the result is C1F0F1.

For A02, the result is C1F0F2.

**XTPACK: Writing a Packed Number With Up to 31 Significant Digits to an Output File**

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

**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: Writing a Packed Number With Up to 31 Significant Digits to an Output File

\[
\text{XTPACK}((\text{LONGPCK,13,2,'A13'}));
\]
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:

number

Numeric

Is the number whose ceiling will be returned. The output data type is the same as the input data type.

**Example:** Returning the Ceiling of a Number

CEILING returns the smallest integer larger than the value in GROSS_PROFIT_US:

\[
\text{CEILING(GROSS\_PROFIT\_US)}
\]

For 225.98, the output is 226.00.

For -30.01, the output is -30.00.

**EXPONENT: Raising e to a Power**

EXPONENT raises the constant e to a power.

**Syntax:** How to Raise the Constant e to a Power

\[
\text{EXPONENT(power)}
\]

where:

power

Numeric

Is the power to which to raise e. The output data type is numeric.

**Example:** Raising e to a Power

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

\[
\text{FLOOR(number)}
\]
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

FLOOR returns the smallest integer larger than the value in GROSS_PROFIT_US:

\[
\text{FLOOR(GROSS\_PROFIT\_US)}
\]

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

\[
\text{MOD(dividend, divisor)}
\]

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

MOD returns the remainder of PRICE_DOLLARS divided by DAYSDELAYED

\[
\text{MOD(PRICE\_DOLLARS, DAYSDELAYED)}
\]

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

\[
\text{POWER}(\text{base}, \text{power})
\]

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

Power returns the value COGS_US/20.00 raised to the power stored in DAYSDELAYED.

\[
\text{POWER1} = \text{POWER(COGS_US/20.00, DAYSDELAYED)}
\]

For base 12.15 and power 3, the value is 1,793.61
Chapter 14

Numeric Functions

Numeric functions perform calculations on numeric constants and fields.

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

The ABS function returns the absolute value of a number.
Syntax: How to Calculate Absolute Value

\[ \text{ABS}(\text{in\_value}) \]

where:

\text{in\_value}

Numeric

Is the value for which the absolute value is returned. If you use an expression, use parentheses as needed to ensure the correct order of evaluation.

Example: Calculating Absolute Value

ABS calculates the absolute value of DIFF.

\[ \text{ABS}(\text{DIFF}); \]

For 15, the result is 15.

For -2, the result is 2.

CHKPCK: Validating a Packed Field

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) 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

\[ \text{CHKPCK}(\text{length}, \text{in\_value}, \text{error}, \text{output}) \]

where:

\text{length}

Numeric

Is the number of bytes in the packed field. It can be between 1 and 16 bytes.
infield
Alphanumeric

Is 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

Example: Validating Packed Data

CHKPCK validates the values in PACK_SAL, and store the result in a column with the format P8CM. Values not in packed format return the error code -999. Values in packed format appear accurately.

CHKPCK(8, PACK_SAL, -999, 'P8CM')

DMOD, FMOD, and IMOD: Calculating the Remainder From a Division

The MOD functions calculate the remainder from a division. Each function returns the remainder in a different format.

The functions use the following formula.

\[ \text{remainder} = \text{dividend} - \text{INT}\left(\frac{\text{dividend}}{\text{divisor}}\right) \times \text{divisor} \]

- **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 272.
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.
  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.

EXP: Raising e to the Nth Power

The EXP function raises the value "e" (approximately 2.72) to a specified power. This function is the inverse of the LOG function, which returns 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

**Example:** Raising e to the Nth Power

EXP raises "e" to the power designated by the &POW variable, specified here as 3. The result is then rounded to the nearest integer with the .5 rounding constant. The result has the format D15.3.

```
EXP(&POW, 'D15.3') + 0.5;
```

For 3, the result is APPROXIMATELY 20.

**EXPN: Evaluating a Number in Scientific Notation**

The EXPN function evaluates a number expressed in scientific notation.

**Syntax:** How to Evaluate a Number in Scientific Notation

```
EXPN(n.nn {E|D} {+|-} p)
```

where:

*n.nn*  
Numeric  
Is a numeric constant that consists of a whole number component, followed by a decimal point, followed by a fractional component.

*E, D*  
Denotes scientific notation. E and D are interchangeable.
Indicates if \( p \) is positive or negative.

\( p \)

Integer

Is the power of 10 to which to raise \( n.nn \).

**Note:** \( \text{EXPN} \) does not use an output argument. The format of the result is floating-point double precision.

**Example:** Evaluating a Number in Scientific Notation

\( \text{EXPN} \) evaluates SCI_DATA.

\( \text{EXPN} (\text{SCI\_DATA}) \)

For 1.03E+2, the result is 103.

**INT:** Finding the Greatest Integer

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. If you supply an expression, use parentheses as needed to ensure the correct order of evaluation.

**Note:** \( \text{INT} \) does not use an output argument. The format of the result is floating-point double precision.

**Example:** Finding the Greatest Integer

\( \text{INT} \) finds the greatest integer in DED_AMT.

\( \text{INT} (\text{DED\_AMT}) \)

For $1,261.40, the result is 1261.

For $1,668.69, the result is 1668.
LOG: Calculating the Natural Logarithm

The LOG function returns the natural logarithm of a number.

Syntax: How to Calculate the Natural Logarithm

\[
\text{LOG}(\text{in\_value})
\]

where:

\[
\text{in\_value}
\]

Numeric

Is the value for which the natural logarithm is calculated. If you supply an expression, use parentheses as needed to ensure the correct order of evaluation. If \(\text{in\_value}\) 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.

\[
\text{LOG}(\text{CURR\_SAL})
\]

For $29,700.00, the result is 10.30.
For $26,862.00, the result is 10.20.

MAX and MIN: Finding the Maximum or Minimum Value

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

\[
\{\text{MAX|MIN}\}(\text{value1, value2, ...})
\]

where:

\[
\text{MAX}
\]

Returns the maximum value.

\[
\text{MIN}
\]

Returns the minimum value.
value1, value2
Numeric

Are the values for which the maximum or minimum value is returned. 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.

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 276.

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:

\[ \sqrt{\left( \frac{\sum x_i^2 - (\sum x_i)^2/n}{n - 1} \right)} \]

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**

\[
\text{NORMSDST}(value, \ 'D8')
\]

where:

- **value**
  - Is a normalized value.

- **D8**
  - Is the required format for the result. The value returned by the function is double-precision. You can assign it to a field with any valid numeric format.

**Example:**

**Using the NORMSDST Function**

NORMSDST finds the percentile for Z and stores the result in a column with the format D8.

\[
\text{NORMSDST}(Z, \ 'D8')
\]

For -.07298, the result is .47091.

For -.80273 the result is .21106.

**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 274.

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**

\[
\text{NORMSINV}(value, \ 'D8')
\]
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

**PRDNOR and PRDUNI: Generating Reproducible Random Numbers**

The PRDNOR and PRDUNI functions generate reproducible random numbers:

- PRDNOR generates reproducible double-precision random numbers normally distributed with an arithmetic mean of 0 and a standard deviation of 1.
- PRDUNI generates reproducible double-precision random numbers uniformly distributed between 0 and 1 (that is, any random number it generates has an equal probability of being anywhere between 0 and 1).

**Syntax:** How to Generate Reproducible Random Numbers

{PRDNOR|PRDUNI}(seed, output)

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.

**output**

Double-precision

---

**Example:** Generating Reproducible Random Numbers

PRDNOR assigns random numbers and stores them in a column with the format D12.2.

```
PRDNOR(40, 'D12.2')
```

---

**RDNORM and RDUNIF: Generating Random Numbers**

The RDNORM and RDUNIF functions generate random numbers:

- RDNORM generates double-precision random numbers normally distributed with an arithmetic mean of 0 and a standard deviation of 1.

- RDUNIF generates double-precision random numbers uniformly distributed between 0 and 1 (that is, any random number it generates has an equal probability of being anywhere between 0 and 1).

---

**Syntax:** How to Generate Random Numbers

```
{RDNORM|RDUNIF}(output)
```

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

---

**Example:** Generating Random Numbers

RDNORM assigns random numbers and stores them in a column with the format D12.2.
SQRT: Calculating the Square Root

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} \\
\text{Numeric}

Is the value for which the square root is calculated. If you supply an expression, use parentheses as needed to ensure the correct order of evaluation. If you supply a negative number, the result 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.
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
- RSERVE: Running an R Script
- STDDEV: Calculating the Standard Deviation for a Set of Data Values

### Specify the Partition Size for Simplified Statistical Functions

```sql
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 (0.0) and 1.0.

Reference: Calculate the Correlation Coefficient Between Two Fields

\[
\text{CORRELATION}(\text{field1}, \text{field2})
\]

where:

\text{field1}

Numeric

Is the first set of data for the correlation.

\text{field2}

Numeric

Is the second set of data for the correlation.

Example: Calculating a Correlation

CORRELATION calculates the correlation between DOLLARS and BUDDOLLARS.

\[
\text{CORRELATION(DOLLARS, BUDDOLLARS)}
\]

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: How to Partition Observations Into Clusters Based on the Nearest Mean Value

\[
\text{KMEANS_CLUSTER(number, percent, iterations, tolerance, [prefix1.]field1[, [prefix1.]field2 ...])}
\]
where:

\textit{number} \\
\textbf{Integer} \\
Is number of clusters to extract.

\textit{percent} \\
\textbf{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.

\textit{iterations} \\
\textbf{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.

\textit{tolerance} \\
\textbf{Numeric} \\
Is a weight value between zero (0) and 1.0. The value AUTO uses the internal default tolerance.

\textit{prefix1, prefix2} \\
defines an optional aggregation operator to apply to the field before using it in the calculation. Valid operators are:

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

\textbf{Note:} The operators PCT., RPCT., TOT., MDN., MDE., RNK., and DST. are not supported.

\textit{field1} \\
\textbf{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).

```plaintext
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.

\[ y = a_1 x_1 [a_2 x_2 [a_3 x_3] ...] + b \]

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, ...])
MULTIREGRESS: Creating a Multivariate Linear Regression Column

where:

\textit{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.

\textbf{Example:} \hspace{0.5cm} Creating a Multivariate Linear Regression Column

The following request uses the DOLLARS and BUDDOLLARS fields to generate a regression
column named Estimated\_Dollars.

\begin{verbatim}
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 *
INCLUDED=IBFS:/FILE/IBI\_HTML\_DIR/ibi\_themes/Warm.sty,$
\hspace{1cm} type=data, column = n1, bucket = x-axis,$
\hspace{1cm} type=data, column= dollars, bucket=y-axis,$
\hspace{1cm} type=data, column= buddollars, bucket=y-axis,$
\hspace{1cm} type=data, column= Estimated\_Dollars, bucket=y-axis,$
\hspace{0.8cm} GRAPH\_JS
\hspace{0.8cm} "series":[
\hspace{1.6cm} {"series":2, "color":"orange"}]
\hspace{0.8cm} END
\hspace{1.2cm} ENDS style
\end{verbatim}

END
The output is shown in the following image. The orange line represents the regression equation.

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

```r
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, SEGTYPE=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, SEGTYPE=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, 0.00, 17.1167, 160, 31, 43183.569
1953, 8.0393, 690, 16.7333, 160, 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, 267, 18, 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, 17.65, 247, 7, 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', $
```
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>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 xᵢ is an observation, N is the number of observations, and μ 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

\[
\text{STDDEV}(\text{field}, \text{sampling})
\]

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.

**Example:** Calculating a Standard Deviation

STDDEV calculates the standard deviation of DOLLARS.

\[
\text{STDDEV(DOLLARS,S)}
\]
The result is 6,157.711080272.
STDDEV: Calculating the Standard Deviation for a Set of Data Values
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

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).

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.

-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.

ENCRIPT: Encrypting a Password

The ENCRYPT function encrypts an alphanumeric input value using the encryption algorithm configured in the server. The result is returned as variable length alphanumeric.

Syntax: How to Encrypt a Password

ENCRYPT(password)

where:

password

Fixed length alphanumeric

Is the value to be encrypted.

Example: Encrypting a Password

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: How to Retrieve the Value of an Environment Variable

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

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

PUTENV assigns the value `FOCUS/Shell:` to the UNIX PS1 variable.

```bash
PUTENV('PS1','FOCUS/Shell:')
```

This causes UNIX to display the following prompt when the user issues the UNIX shell command SH:

`FOCUS/Shell:`
System Functions

System functions call the operating system to obtain information about the operating environment or to use a system service.

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.

Syntax: How to Close All Files Opened by the PUTDDREC Function

CLSDDREC(output)

where:

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:

```
CLSDDREC('II')
```

**FEXERR: Retrieving an Error Message**

The FEXERR function retrieves an Information Builders error message. It is especially useful in a procedure using a command that suppresses the display of output messages.

An error message consists of up to four lines of text. The first line contains the message and the remaining three contain a detailed explanation, if one exists. FEXERR retrieves the first line of the error message.

**Syntax:** How to Retrieve an Error Message

```
FEXERR(error, 'A72')
```

where:

- `error`  
  Numeric  
  Is the error number, up to 5 digits long.

- `'A72'`  
  Is the format of the output value. 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.

```
FEXERR(&ERR, 'A72')
```

The result is (FOC650) THE DISK IS NOT ACCESSED.
**FGETENV: Retrieving the Value of an Environment Variable**

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

\[\text{FGETENV}(\text{length, 'varname', outlen, output})\]

where:

\[\text{length}\]
- Integer
- Is the number of characters in the environment variable name.

\[\text{varname}\]
- Alphanumeric
- Is the name of the environment variable whose value is being retrieved.

\[\text{outlen}\]
- Integer
- Is the length of the environment variable value returned.

\[\text{output}\]
- Alphanumeric
- Is the format of the field in which the environment variable’s value is stored.

**FPUTENV: Assigning a Value to an Environment Variable**

Available Operating Systems: IBM i (formerly referred to as i5/OS), Tandem, UNIX, Windows

The FPUTENV function assigns a character string to an environment variable.

**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

\[ \text{FPUTENV}(\text{varname\_length}, \text{varname}, \text{value\_length}, \text{value}, \text{output}) \]

where:

\[ \text{varname\_length} \]

Integer

Is the maximum number of characters in the name of the environment variable.

\[ \text{varname} \]

Alphanumeric

Is the name of the environment variable. The name must be right-justified and padded with blanks to the maximum length specified by \text{varname\_length}.

\[ \text{value\_length} \]

Is the maximum length of the environment variable value.

**Note:** The sum of \text{varname\_length} and \text{value\_length} cannot exceed 64.

\[ \text{value} \]

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.

\[ \text{output} \]

Integer

Is the return code. 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:

\[-\text{SET } \&\text{RC } = \text{FPUTENV}(3, \text{'PS1'}, 12, \text{'FOCUS/Shell:'}, \text{'I4'});\]

The request displays the following prompt when the user issues the UNIX shell command SH:

FOCUS/Shell:
GETUSER: Retrieving a User ID

The GETUSER function retrieves the ID of the connected user.

**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)

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:

```
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. If the file is defined as an existing file (with the APPEND option), the new record is appended. If the file is defined as NEW and it already exists, the new record overwrites the existing file.

PUTDDREC opens the file if it is not already open. Each call to PUTDDREC can use the same file or a new one. All of the files opened by PUTDDREC remain open until the end of a request or connection. At the end of the request or connection, all files opened by PUTDDREC are automatically closed.

If PUTDDREC is called in a Dialogue Manager -SET command, the files opened by PUTDDREC are not closed automatically until the end of a request or connection. In this case, you can close the files and free the memory used to store information about open file by calling the CLSDDREC function.

**Syntax:**

How to Write a Character String as a Record in a Sequential File

```plaintext
PUTDDREC(ddname, dd_len, record_string, record_len, output)
```

where:

- **ddname**
  - Alphanumeric
  - Is the logical name assigned to the sequential file in a FILEDEF command.

- **dd_len**
  - Numeric
  - Is the number of characters in the logical name.

- **record_string**
  - Alphanumeric
  - Is the character string to be added as the new record in the sequential file.

- **record_len**
  - Numeric
  - Is the number of characters to add as the new record.
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 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,

```
FILEDEF LOGGING DISK baseapp/logging.dat
```

```
PUTDDREC('LOGGING', 7, 'Country:' | COUNTRY, 20, 'I5')
```

would return the value 0, and would write the following lines to logging.dat:

Country: ENGLAND
Country: JAPAN
Country: ITALY
Country: W GERMANY
Country: FRANCE

**SLEEP: Suspending Execution for a Given Number of Seconds**

The SLEEP function suspends execution for the number of seconds you specify as its input argument.

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

\[ \text{SLEEP}(\text{delay, output}); \]

where:

**delay**

Numeric

Is the number of seconds to delay execution. The number can be specified down to the millisecond.

**output**

Numeric

The value returned is the same value you specify for delay.

**Example:** Suspending Execution for Four Seconds

SLEEP suspends execution for four seconds:

\[ \text{-SET \&DELAY = SLEEP(4.0, 'I2');} \]

**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

\[ \text{-SET \&dmvar = SYSVAR('length', '[\&]sysvar[.]', 'outfmt');} \]

where:

**\&dmvar**

Alphanumeric

Is the name of the Dialogue Manager variable to be populated with the value of the z/OS system variable.
**SYSVAR: Retrieving the Value of a 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
```
Simplified Geography Functions

The simplified geography functions perform location-based calculations and retrieve geocoded 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_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**

```plaintext
FILENAME=ESRI-CITIBIKE, SUFFIX=DFIX    ,
DATASET=esri/esri-citibike.csv, $
SEGMENT=CITIBIKE_TRIPDATA, SEGTYPE=SO, $
    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, $
```

310

Information Builders
esri-citibike.acx

SEGNAME=CITIBIKE_TRIPDATA,
   DELIMITER=':',
   ENCLOSEMENT="",
   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
<table>
<thead>
<tr>
<th>Address Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>284, 11/1/2015 0:01, 11/1/2015 0:06, 247, Perry St &amp; Bleecker St,</td>
</tr>
<tr>
<td>40.73535398, -74.00483091, 453, W 22 St &amp; 8 Ave, 40.74475148, -73.99915362,</td>
</tr>
<tr>
<td>17272, Subscriber, 1976, 1</td>
</tr>
<tr>
<td>886, 11/1/2015 0:01, 11/1/2015 0:16, 492, W 33 St &amp; 7 Ave, 40.75019995, -73.99093085, 377, 6</td>
</tr>
<tr>
<td>Ave &amp; Canal St, 40.72243797, -74.00566443, 23019, Subscriber, 1982, 1</td>
</tr>
<tr>
<td>1379, 11/1/2015 0:01, 11/1/2015 0:24, 512, W 29 St &amp; 9 Ave, 40.7500727, -73.99839279, 445, E</td>
</tr>
<tr>
<td>10 St &amp; Avenue A, 40.72740794, -73.98142006, 23843, Subscriber, 1962, 2</td>
</tr>
<tr>
<td>179, 11/1/2015 0:01, 11/1/2015 0:04, 319, Fulton St &amp; Broadway,</td>
</tr>
<tr>
<td>40.711066, -74.009447, 264, Maiden Ln &amp; Pearl St, 40.70706456, -74.00731853,</td>
</tr>
<tr>
<td>22538, Subscriber, 1981, 1</td>
</tr>
<tr>
<td>309, 11/1/2015 0:01, 11/1/2015 0:07, 160, E 37 St &amp; Lexington Ave,</td>
</tr>
<tr>
<td>40.748238, -73.978311, 362, Broadway &amp; W 37 St, 40.75172632, -73.98753523,</td>
</tr>
<tr>
<td>22042, Subscriber, 1988, 1</td>
</tr>
<tr>
<td>616, 11/1/2015 0:02, 11/1/2015 0:12, 479, 9 Ave &amp; W 45 St, 40.76019252, -73.9912551, 440, E</td>
</tr>
<tr>
<td>45 St &amp; 3 Ave, 40.75255343, -73.97282625, 22699, Subscriber, 1982, 1</td>
</tr>
<tr>
<td>852, 11/1/2015 0:02, 11/1/2015 0:16, 346, Bank St &amp; Hudson St,</td>
</tr>
<tr>
<td>40.73652889, -74.00618026, 375, Mercer St &amp; Bleecker St, 40.72679454, -73.99695094,</td>
</tr>
<tr>
<td>21011, Subscriber, 1991, 1</td>
</tr>
<tr>
<td>1854, 11/1/2015 0:02, 11/1/2015 0:33, 409, DeKalb Ave &amp; Skillman St,</td>
</tr>
<tr>
<td>40.6906495, -73.95643107, 3103, N 11 St &amp; Wythe Ave, 40.72153267, -73.95782357,</td>
</tr>
<tr>
<td>22011, Subscriber, 1992, 1</td>
</tr>
<tr>
<td>1161, 11/1/2015 0:02, 11/1/2015 0:21, 521, 8 Ave &amp; W 31 St, 40.75096735, -73.99444208, 461, E</td>
</tr>
<tr>
<td>20 St &amp; 2 Ave, 40.73587678, -73.98205027, 19856, Subscriber, 1957, 1</td>
</tr>
<tr>
<td>917, 11/1/2015 0:02, 11/1/2015 0:17, 532, S 5 Pl &amp; S 4 St, 40.710451, -73.960876, 393, E 5 St</td>
</tr>
<tr>
<td>&amp; Avenue C, 40.72299208, -73.97995466, 18598, Subscriber, 1991, 1</td>
</tr>
</tbody>
</table>

**esri-geo10036.mas**

```plaintext
FILENAME=ESRI-GEO10036, 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, $
```
Functions Reference
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.
**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
```
The output is shown in the following image.

<table>
<thead>
<tr>
<th>Start</th>
<th>End</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Ave &amp; E 30 St</td>
<td>Broadway &amp; W 32 St</td>
<td>.83</td>
</tr>
<tr>
<td>8 Ave &amp; W 31 St</td>
<td>Broadway &amp; E 14 St</td>
<td>1.15</td>
</tr>
<tr>
<td></td>
<td>E 20 St &amp; 2 Ave</td>
<td>1.23</td>
</tr>
<tr>
<td>9 Ave &amp; W 45 St</td>
<td>E 45 St &amp; 3 Ave</td>
<td>1.10</td>
</tr>
<tr>
<td>Bank St &amp; Hudson St</td>
<td>Mercer St &amp; Bleecker St</td>
<td>.83</td>
</tr>
<tr>
<td>Broadway &amp; E 14 St</td>
<td>Cleveland Pl &amp; Spring St</td>
<td>.92</td>
</tr>
<tr>
<td>DeKalb Ave &amp; Skillman St</td>
<td>N 11 St &amp; Wythe Ave</td>
<td>2.13</td>
</tr>
<tr>
<td>E 37 St &amp; Lexington Ave</td>
<td>Broadway &amp; W 37 St</td>
<td>.54</td>
</tr>
<tr>
<td>Fulton St &amp; Broadway</td>
<td>Maiden Ln &amp; Pearl St</td>
<td>.30</td>
</tr>
<tr>
<td>Grand Army Plaza &amp; Central Park S</td>
<td>E 53 St &amp; Lexington Ave</td>
<td>.45</td>
</tr>
<tr>
<td>Great Jones St</td>
<td>Watts St &amp; Greenwich St</td>
<td>.87</td>
</tr>
<tr>
<td>Greenwich St &amp; Warren St</td>
<td>E 52 St &amp; 2 Ave</td>
<td>3.62</td>
</tr>
<tr>
<td>Henry St &amp; Poplar St</td>
<td>State St &amp; Smith St</td>
<td>.78</td>
</tr>
<tr>
<td>Howard St &amp; Centre St</td>
<td>E 31 St &amp; 3 Ave</td>
<td>2.01</td>
</tr>
<tr>
<td>Lexington Ave &amp; E 24 St</td>
<td>Forsyth St &amp; Broome St</td>
<td>1.54</td>
</tr>
<tr>
<td>Perry St &amp; Bleecker St</td>
<td>W 22 St &amp; 8 Ave</td>
<td>.71</td>
</tr>
<tr>
<td>S 5 Pl &amp; S 4 St</td>
<td>E 5 St &amp; Avenue C</td>
<td>1.32</td>
</tr>
<tr>
<td>W 29 St &amp; 9 Ave</td>
<td>E 10 St &amp; Avenue A</td>
<td>1.80</td>
</tr>
<tr>
<td>W 31 St &amp; 7 Ave</td>
<td>E 30 St &amp; Park Ave S</td>
<td>.55</td>
</tr>
<tr>
<td>W 33 St &amp; 7 Ave</td>
<td>6 Ave &amp; Canal St</td>
<td>2.07</td>
</tr>
</tbody>
</table>

**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:**

```
GIS_DRIVE_ROUTE(geo_start_point, geo_end_point)
```

Information Builders
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.

```sql
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_GEOCODE_ADDR: Geocoding a Complete Address

GIS_GEOCODE_ADDR uses a GIS geocoding service to obtain the geometry point for a complete address.

Syntax:

How to Geocode a Complete Address

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.

```
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></td>
</tr>
<tr>
<td>10016</td>
<td>159 E 35th St</td>
<td></td>
</tr>
<tr>
<td>10017</td>
<td>2 United Nations Plz</td>
<td></td>
</tr>
<tr>
<td>403</td>
<td>E 42N4 St</td>
<td></td>
</tr>
<tr>
<td>405</td>
<td>E 42N4 St</td>
<td></td>
</tr>
<tr>
<td>219</td>
<td>E 42N4 St</td>
<td></td>
</tr>
<tr>
<td>330</td>
<td>Madison Ave</td>
<td></td>
</tr>
<tr>
<td>10018</td>
<td>119 W 40th St Fl 10</td>
<td></td>
</tr>
<tr>
<td>10019</td>
<td>31 West 52nd Street</td>
<td></td>
</tr>
<tr>
<td>1301</td>
<td>Ave Of The Americas</td>
<td></td>
</tr>
<tr>
<td>1345</td>
<td>Avenue Of The Americas</td>
<td></td>
</tr>
<tr>
<td>745</td>
<td>Th Ave</td>
<td></td>
</tr>
<tr>
<td>1221</td>
<td>Avenue Of The Americas</td>
<td></td>
</tr>
<tr>
<td>1271</td>
<td>Avenue Of The Americas</td>
<td></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

GIS_GEOCODE_ADDR_CITY( street_addr, city , state [, country])

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
The output is shown in the following image.

<table>
<thead>
<tr>
<th>Zip</th>
<th>Address</th>
<th>GIS_GEOCODE_ADDR_POSTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>10013</td>
<td>125 Worth St</td>
<td>(&quot;spatialReference&quot;: &quot;wkid&quot;: 4326, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -74.00269, &quot;y&quot;: 40.71543})</td>
</tr>
<tr>
<td>10016</td>
<td>159 E 35Th St</td>
<td>(&quot;spatialReference&quot;: &quot;wkid&quot;: 4326, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -73.94483, &quot;y&quot;: 40.65194})</td>
</tr>
<tr>
<td>10017</td>
<td>2 United Nations Plz</td>
<td>(&quot;spatialReference&quot;: &quot;wkid&quot;: 4326, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -73.99715, &quot;y&quot;: 40.75111})</td>
</tr>
<tr>
<td></td>
<td>405 E 42Nd St</td>
<td>(&quot;spatialReference&quot;: &quot;wkid&quot;: 4326, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -73.96656, &quot;y&quot;: 40.74867})</td>
</tr>
<tr>
<td></td>
<td>219 E 42Nd St</td>
<td>(&quot;spatialReference&quot;: &quot;wkid&quot;: 4326, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -73.96956, &quot;y&quot;: 40.74867})</td>
</tr>
<tr>
<td></td>
<td>330 Madison Ave</td>
<td>(&quot;spatialReference&quot;: &quot;wkid&quot;: 4326, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -73.97333, &quot;y&quot;: 40.75030})</td>
</tr>
<tr>
<td>10018</td>
<td>119 W 40Th St FI 10</td>
<td>(&quot;spatialReference&quot;: &quot;wkid&quot;: 4326, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -73.98599, &quot;y&quot;: 40.75398})</td>
</tr>
<tr>
<td></td>
<td>111 West 40Th Street</td>
<td>(&quot;spatialReference&quot;: &quot;wkid&quot;: 4326, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -73.98235, &quot;y&quot;: 40.75245})</td>
</tr>
<tr>
<td>10019</td>
<td>51 West 52Nd Street</td>
<td>(&quot;spatialReference&quot;: &quot;wkid&quot;: 4326, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -73.97776, &quot;y&quot;: 40.76044})</td>
</tr>
<tr>
<td></td>
<td>1301 Ave Of The Americas</td>
<td>(&quot;spatialReference&quot;: &quot;wkid&quot;: 4326, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -73.97945, &quot;y&quot;: 40.76125})</td>
</tr>
<tr>
<td></td>
<td>1345 Avenue Of The Americas</td>
<td>(&quot;spatialReference&quot;: &quot;wkid&quot;: 4326, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -73.97843, &quot;y&quot;: 40.76264})</td>
</tr>
<tr>
<td></td>
<td>745 7Th Ave</td>
<td>(&quot;spatialReference&quot;: &quot;wkid&quot;: 4326, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -72.98240, &quot;y&quot;: 40.76077})</td>
</tr>
<tr>
<td>10020</td>
<td>1231 Avenue Of The Americas</td>
<td>(&quot;spatialReference&quot;: &quot;wkid&quot;: 4326, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;: -73.98129, &quot;y&quot;: 40.75874})</td>
</tr>
<tr>
<td></td>
<td>1271 Avenue Of The Americas</td>
<td>(&quot;spatialReference&quot;: &quot;wkid&quot;: 4326, &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_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.
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.

```
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
```
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
```
The output is shown in the following image.

<table>
<thead>
<tr>
<th>Station ID</th>
<th>Station</th>
<th>Station in zip code 10036?</th>
</tr>
</thead>
<tbody>
<tr>
<td>147</td>
<td>Greenwich St &amp; Warren St</td>
<td>No</td>
</tr>
<tr>
<td>160</td>
<td>E 37 St &amp; Lexington Ave</td>
<td>No</td>
</tr>
<tr>
<td>229</td>
<td>Great Jones St</td>
<td>No</td>
</tr>
<tr>
<td>247</td>
<td>Perry St &amp; Bleecker St</td>
<td>No</td>
</tr>
<tr>
<td>268</td>
<td>Howard St &amp; Centre St</td>
<td>No</td>
</tr>
<tr>
<td>281</td>
<td>Grand Army Plaza &amp; Central Park S</td>
<td>No</td>
</tr>
<tr>
<td>285</td>
<td>Broadway &amp; E 14 St</td>
<td>No</td>
</tr>
<tr>
<td>319</td>
<td>Fulton St &amp; Broadway</td>
<td>No</td>
</tr>
<tr>
<td>346</td>
<td>Bank St &amp; Hudson St</td>
<td>No</td>
</tr>
<tr>
<td>379</td>
<td>W 31 St &amp; 7 Ave</td>
<td>No</td>
</tr>
<tr>
<td>407</td>
<td>Henry St &amp; Poplar St</td>
<td>No</td>
</tr>
<tr>
<td>409</td>
<td>DeKalb Ave &amp; Skillman St</td>
<td>No</td>
</tr>
<tr>
<td>479</td>
<td>9 Ave &amp; W 45 St</td>
<td>Yes</td>
</tr>
<tr>
<td>492</td>
<td>W 33 St &amp; 7 Ave</td>
<td>No</td>
</tr>
<tr>
<td>512</td>
<td>W 29 St &amp; 9 Ave</td>
<td>No</td>
</tr>
<tr>
<td>521</td>
<td>8 Ave &amp; W 31 St</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>8 Ave &amp; W 31 St</td>
<td>No</td>
</tr>
<tr>
<td>532</td>
<td>S 5 Pl &amp; S 4 St</td>
<td>No</td>
</tr>
<tr>
<td>536</td>
<td>1 Ave &amp; E 30 St</td>
<td>No</td>
</tr>
<tr>
<td>537</td>
<td>Lexington Ave &amp; E 24 St</td>
<td>No</td>
</tr>
</tbody>
</table>

**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
```
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:**

```
GIS_POINT(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.

latitude
D20.8
Is the latitude 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.

```
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>Atlanta</td>
<td>-73.76000000</td>
<td>42.60000000</td>
<td>(&quot;spatialReference&quot;: {&quot;wkid&quot;: 4326}, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;:-73.76000000, &quot;y&quot;:42.60000000})</td>
</tr>
<tr>
<td>Austin</td>
<td>-97.75000000</td>
<td>30.40000000</td>
<td>(&quot;spatialReference&quot;: {&quot;wkid&quot;: 4326}, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;:-97.75000000, &quot;y&quot;:30.40000000})</td>
</tr>
<tr>
<td>Baton Rouge</td>
<td>-91.17000000</td>
<td>30.33000000</td>
<td>(&quot;spatialReference&quot;: {&quot;wkid&quot;: 4326}, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;:-91.17000000, &quot;y&quot;:30.33000000})</td>
</tr>
<tr>
<td>Birmingham</td>
<td>-86.27000000</td>
<td>33.44000000</td>
<td>(&quot;spatialReference&quot;: {&quot;wkid&quot;: 4326}, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;:-86.27000000, &quot;y&quot;:33.44000000})</td>
</tr>
<tr>
<td>Boston</td>
<td>-71.14000000</td>
<td>42.35000000</td>
<td>(&quot;spatialReference&quot;: {&quot;wkid&quot;: 4326}, &quot;geometryType&quot;: &quot;esriGeometryPoint&quot;, &quot;geometry&quot;: {&quot;x&quot;:-71.14000000, &quot;y&quot;:42.35000000})</td>
</tr>
</tbody>
</table>

Functions Reference 329
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:**

**How to Calculate a Geometry Area Around a Point**

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
```
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:**

```plaintext
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
```

The output is shown in the following image.
SQL Character Functions

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 343 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

\[
\text{CONCAT}(\text{arg1}, \text{arg2})
\]

where:

\[
\text{arg1}, \text{arg2}
\]

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,

\[
\text{CONCAT('abc', 'def')}
\]

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

\[
\text{DIGITS}(\text{arg})
\]

where:

\[
\text{arg}
\]

Numeric (decimal or integer, not floating-point)

Is the numeric value.

The length of the resulting string is determined by the precision of the argument.
Example: Converting a Numeric Value to a Character String
DIGITS converts a numeric value to a character string. This example,
DIGITS(-444.321)
returns 0000444321.

EDIT: Editing a Value According to a Format (SQL)
The EDIT function edits a numeric or character value according to a format specified by a mask. (It works exactly like the EDIT function in FOCUS.)

A 9 in the mask indicates the corresponding character in the source value is copied into the result. A $ in the mask indicates that the corresponding character is to be ignored. Any other character is inserted into the result.

Syntax: How to Edit a Value According to a Format
EDIT(arg, mask)
where:
arg
Numeric or character string
Is the value to be edited.
mask
character string
Indicates how the editing is to proceed.

This function returns a character string whose length is determined by the mask.

Example: Editing a Value According to a Format
EDIT extracts a character from a string. This example,
EDIT('FRED', '9$$$')
returns F.
This example,
EDIT('123456789', '999-99-9999')
returns 123-45-6789.
**LCASE: Converting a Character String to 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:**

```
LCASE(arg)
```

where:

- **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,

```
LCASE('XYZ')
```

returns xyz.

**LTRIM: Removing Leading Spaces**

The LTRIM function removes leading spaces from a character string.

**Syntax:**

```
LTRIM(arg)
```

where:

- **arg**
  - character string
  - Is the value to be trimmed.

This function returns a varying character string. The data type of the result has a length equal to that of the input argument (although the value may be shorter).
Example:  Removing Leading Spaces

LTRIM removes leading spaces. This example,

LTRIM( ' ABC ')

returns 'ABC '.

POSITION: Finding the Position of a Substring

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:**

How to Remove Trailing Spaces

\[ \text{RTRIM}(\text{arg}) \]

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 Trailing Spaces

RTRIM removes trailing spaces. This example,

\[ \text{RTRIM('ABC')} \]

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 '.

Functions Reference 343
This example,

\[
\text{TRIM(TRAILING FROM ' ABC ')}
\]
\[
\text{TRIM(BOTH 'X' FROM 'XXYYXXXX')} = ('YYY')
\]

returns ' ABC'

This example,

\[
\text{TRIM(BOTH 'X' FROM 'XXYYXXXX')}
\]

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**

\[
\text{UCASE(arg)}
\]

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**

\[
\text{VARGRAPHICarg}
\]
where:

arg

character, graphic, or date

Is the input value.

**Note:** This function can only be used for DB2 and can only be used with Direct or Automatic Passthru. This function returns the value in double-byte character format.
SQL date and time functions perform manipulations on date and time values.

**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

```sql
DAY(arg)
```

where:

- `arg` 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,

```sql
DAY('1976-07-04')
```

returns 4.

This example,

```sql
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:

- **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:

- **arg**
  - Date, time, or timestamp
  - Is the input argument.

- **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,

```sql
HOUR('11:22:33')
```

returns 11.

This example,

```sql
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

```sql
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,

```sql
MICROSECOND('11:22:33.456789')
```

returns 456,789.

This example,

```sql
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

\[
\text{MILLISECOND}(\text{arg})
\]

where:

\[
\text{arg}
\]

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,

\[
\text{MILLISECOND('11:22:33.456')}
\]

returns 456.

This example,

\[
\text{MILLISECOND('2001-01-22 10:11:12')}
\]

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

\[
\text{MINUTE}(\text{arg})
\]

where:

\[
\text{arg}
\]

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,

```plaintext
MINUTE('11:22:33')
```

returns 22.

This example,

```plaintext
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

```plaintext
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,

```plaintext
MONTH('1976-07-04')
```

returns 7.

This example,

```plaintext
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: Return 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,

```
YEAR('1976-07-04')
```

returns 1976.

This example,

```
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:**

```sql
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 361.

**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 360.

**Syntax:** How to Convert a Date, Time, or Timestamp Value to a Standard Format

\[
\text{CHAR}(\text{datetime}, \text{fmt})
\]

where:

- **datetime**
  - Date
  - Is the date, time, or timestamp value to be converted.

- **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:

- **CHAR(CURRENT DATE, USA)** returns 11/17/2011
- **CHAR(CURRENT DATE, ISO)** returns 2011-11-17
- **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:

**DATE**

**DATE**(arg)

where:

**arg**

character string, date, or timestamp

Is the value to be converted.

The DATE function returns a date in YYMD format.

Example: Converting to a Date

DATE converts a value to a date. This example,

**DATE**(‘1999-03-29 14:39:30')

returns 19990329.

DECIMAL: Converting to Decimal Format

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
INT: Converting to an Integer

Is the input value.
This function returns the value in floating-point format.

**Example:** Converting to Floating Point Format

FLOAT converts a number to floating-point format. This example,

```
FLOAT(3)
```

returns 3.0.

INT: Converting to an Integer

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:

\( \text{arg} \)

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

\[ \text{TIME}(\text{arg}) \]

where:

\( \text{arg} \)

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,

\[ \text{TIME}('2004-03-15 01:02:03.444') \]
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

TIMESTAMP (arg)

where:

arg

character string, date, time, or timestamp

Is the input value.

This function returns a timestamp.

Example: Converting to a Timestamp

TIMESTAMP converts a value to a timestamp. This example,

TIMESTAMP ('2004-06-24')

returns 20040624000000.

This example,

TIMESTAMP ('11:22:33')

returns 2001010112233, 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

\[
\text{ABS}(arg)
\]

where:

\[
arg
\]

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,
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
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.
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
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

\[ \text{GREATEST}(value1, value2, \ldots, valuen) \]

where:

\( value1, value2, \ldots, valuen \)

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

\[
\text{LEAST}(\text{value}1, \text{value}2, \ldots, \text{valuen})
\]

where:

\[
\text{value}1, \text{value}2, \ldots, \text{valuen}
\]

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**

```
MOD (n, m)
```

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`.

```
MOD (N, M)
```

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**

```
POWER (arg1, arg2)
```

where:

- `arg1`
  Numeric
  Is the value to be raised to the power specified by `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:

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.
SQL Miscellaneous Functions

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:**

- 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_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
```

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
```

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

```
IF(test, val1, val2)
```

where:

- **test**  
  Condition  
  Is an SQL search condition, which evaluates to true or false.

- **val1, val2**  
  Are expressions of compatible types.

This function returns a value of the type of val1 and val2. If test is true, val1 is returned, otherwise val2 is returned.

**Example:** Testing a Condition

This example tests COUNTRY. If the value is ENGLAND, it returns LONDON. Otherwise, it returns PARIS.

```
IF(COUNTRY = 'ENGLAND', 'LONDON', 'PARIS') =  
  'LONDON' if COUNTRY is 'ENGLAND'  
  'PARIS' otherwise.
```

This example tests COUNTRY. If the value is ENGLAND, it returns LONDON. If the value is FRANCE, it returns PARIS. Otherwise, it returns ROME.
IF(COUNTRY = 'ENGLAND', 'LONDON',
  IF(COUNTRY = 'FRANCE', 'PARIS', 'ROME')) =
  'LONDON' if COUNTRY is 'ENGLAND'
  'PARIS' if COUNTRY = 'FRANCE'
  'ROME' otherwise.

LENGTH: Obtaining the Physical Length of a Data Item

The LENGTH function returns the actual length in memory of a data item.

Syntax: How to Obtain the Physical Length of a Data Item

```
LENGTH(arg)
```

where:

arg

Any type

Is the length of the argument. It can be between 1 and 16 bytes.

This function returns an integer value.

Example: Obtaining the Physical Length of a Data Item

LENGTH returns the length in memory of a data item. This example,

```
LENGTH('abcdef')
```

returns 6.

This example,

```
LENGTH(3)
```

returns 4.

VALUE: Coalescing Data Values

Note: The SQL function VALUE is not supported. Instead, use the SQL operator COALESCE. For more information see COALESCE: Coalescing Data Values on page 383.
VALUE: Coalescing Data Values
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 `test-expr` is compared to `value-expr-1, ..., value-expr-n` in turn:

- If any of these match, the value of the result is the corresponding `result-expr`.
- If there are no matches and the ELSE clause is present, the result is `else-expr`.
- If there are no matches and the ELSE clause is not present, the result is NULL.

In the second format below the values of `cond-1, ..., cond-n` are evaluated in turn.

- If any of these are true, the value of the result is the corresponding `result-expr`.
- If no conditions are true and the ELSE clause is present, the result is `else-expr`.
- If no conditions are true and the ELSE clause is not present, the result is NULL.

**Format 1**

```sql
CASE test-expr
    WHEN value-expr-1 THEN result-expr-1
    . . .
    WHEN value-expr-n THEN result-expr-n
[ ELSE else-expr ]
END
```

Functions Reference
CASE: SQL Case Operator

**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:** How to Use the NULLIF Operator

```
NULLIF(arg1, arg2)
```

where:

```
arg1, arg2
```

Any type

Are data values. The types of the two arguments must be compatible.

This operator returns the compatible type of the arguments.

**Example:** Using the NULLIF Operator

NULLIF operator returns NULL if two values are equal. This example,

```
NULLIF(IDNUM, -1)
```

returns NULL if the identification number is -1, otherwise it returns the number.
Index

A

ABS function 267, 268, 367
alphanumeric strings 249
analytic functions 29
ARGLEN function 82, 114, 115
ASIS function 82, 83
ATODBL function 249, 250
AYMD function 206

B

bit strings 85, 86
BITSON function 84, 85
BITVAL function 85, 86
BUSDAYS parameter 167
business days 167
BUSDAYS parameter 167
BYTVAL function 86, 87

C

CASE operator 381
CAST function 359
CEILING function 263
CHAR function 241, 360, 361
CHAR_LENGTH function 64, 335
character functions, simplified 63
CHAR_LENGTH 64
CONCAT 64
DIGITS 65
character functions, simplified 63
LAST_NONBLANK 66
LOWER 67
LPAD 68
LTRIM 69
PATTERNS 70
POSITION 70
REGEX 71
REPLACE 72
RPAD 73
RTRIM 74
SPLIT 75
SUBSTRING 76
TOKEN 77
TRIM 78
UPPER 79

Functions Reference
character functions
GETTOK 93
LCWORD 94, 96
LCWORD2 95, 96
LCWORD3 96
LJUST 97
LOCASE 97, 98
OVRLAY 98
PARAG 99–101
POSIT 102, 103
RJUST 104
SOUNDEX 104, 105
SPELLNM 105, 106
SQL 335
SQUEEZ 106, 107
STRIP 107–109
SUBSTR 109–111, 121
TRIM 111, 112
TRIMV 122
UPCASE 112, 113
variable length 117
XMLDECOD 113
XMLENCOD 114
character strings 82, 97
extracting characters 92
extracting substrings 93, 109, 111, 121
finding substrings 102
format 87
justifying 97, 104
measuring length 82
overlaying 98
reducing spaces 106
right-justifying 104
spelling out numbers 105
translating characters 86, 89, 90
CHECKMD5 function 139
CHECKSUM function 140
CHGDAT function 207, 209
CHKFMT function 87–89
CHKNUM function 89
CHKPCK function 268
CLSDDREC 305
COALESCE operator 383
components 222
CONCAT function 64, 336
conversion functions, simplified 241
CHAR 241
CTRLCHAR 242
EDIT2 244
HEXTYPE 244
TO_INTEGER 246
TO_NUMBER 246
conversion functions, simplified
    PHONETIC 245
converting formats 249
COUNTRY function 375
cross-referenced data sources 152
CTRAN function 89, 90
CTRLFLD function 91
CTRLCHAR function 242
CURRENT_DATE function 347
CURRENT_TIME function 348
CURRENT_TIMESTAMP function 348
date and time functions 166, 218
    arguments and 222
    AYMD 206
    CHGDAT 207, 209
    DA 210
    DADMY 210
    DADYM 210
    DAMDY 210
    DAMYD 210
    DATEADD 173
    DATECVT 175
    DATEDIF 177
    DATEMOV 179
    DATETRAN 183
    DAYDM 210
    DAYMD 210
    DOWK 212
    DOWKL 212
    HADD 223
    HCNVRT 225
    HDATE 226
    HDIFF 226, 227
    HDTTM 228
    HGEC 229, 230
    HGETZ 230
    HHMSS 231
    HHMS 232
    HINPUT 233
    HMIDNT 234
    HNAME 235
data source functions 139
    FIND 151
    LAST 151, 152
    LOOKUP 152, 153
data sources 139
    cross-referenced 152
    decoding values 149
    retrieving values 151, 152
    values 139
    verifying values 151
data type conversion functions 359
date and time functions 166, 218
HPART 235, 236
HSETPT 236
HTIME 237, 238
JULDAT 215
legacy 206
SQL 347
standard 166
TODAY 204
YM 215
date formats
international 183
DATE function 362
date functions, simplified 155
DT_CURRENT_DATE 156
DT_CURRENT_DATETIME 156
DT_CURRENT_TIME 157
DTPART 161
DTRUNC 162
date functions
work days 167
date-time values
moving dates 179
retrieving components 235
retrieving time 231, 232
returning dates 204
setting time 234
storing 229, 230
subtracting 206
DATEADD function 173
DATECVT function 175
DATEDIF function 177
DATEFNS parameter 206
DATEFORMAT parameter 219
DATEMOV function 179
DATETRAN function 183, 191
DAY function 349
DAYDM function 210
DAYMD function 210
DAYS function 350
DB_EXPR function 140, 376
DB_LOOKUP function 147
COMPUTE command 147
DEFINE 147
MODIFY 147
TABLE COMPUTE 147
DCTRAN function 125
DECIMAL function 362
DECODE function 149, 150
decoding functions 139, 149, 150
decoding values 149
from files 149
in a function 149, 150
DEDIT function 126
DIGITS function 65, 337
DMOD function 269, 270
DMY function 211
double exponential smoothing 38
FORECAST_DOUBLEEXP 38
double-byte characters 125, 128
DOWK function 212
DOWKL function 212
DSTRIp function 128, 129
DSUBSTR function 129
DSUBSTR function 129
DT_CURRENT_DATE function 156
DT_CURRENT_DATETIME function 156
DT_CURRENT_TIME function 157
DTADD function 157
DTDIFF function 159
DTPART function 161
DTRUNC function 162
DTSTRICT parameter 221

E
EDIT function 92, 251, 338
EDIT2 function 244
ENCRYPT function 296
environment variables 301
assigning values 301
retrieving values 301
error messages 300
EXP function 270, 271, 371
EXPN function 271
exponential moving average 35
FORECAST_EXPAVE 35
EXTRACT function 350

F
FEXERR function 300
FGETENV function 301
FIND function 150, 151
FIQTR function 201
FIYR function 199
FIYYQ function 203
FLOAT function 363
FLOOR function 264
FMOD function 269, 270
FORECAST_DOUBLEEXP
double exponential smoothing 38
FORECAST_EXPAVE exponential moving average 35
FORECAST_LINEAR linear regression equation 44
FORECAST_MOVAVE simple moving average 29
FORECAST_SEASONAL triple exponential smoothing 40
format conversion functions
ATODBL 249, 250
EDIT 251
format conversion functions

FPRINT 251
FTOA 252, 253
HEXBYT 253, 254
ITIONUM 254
ITOPACK 256
ITOZ 256, 257
PCKOUT 257, 258
PTOA 258, 259
TSTOPACK 259
UFMT 260, 261

format conversions 249
packed numbers 257
to alphanumeric 252, 258
to characters 253
to hexadecimal 260
to zoned format 256

formats 249
alphanumeric 251
converting 249
FPRINT function 251
FPUTENV function 301, 302
FTOA function 252, 253

function types

data source 139
decoding 139
numeric 267
system 299

functions 147
analytic 29

functions 147
character 335
data type conversion 359
date and time 166, 206, 218, 347
FIND 151
FIQTR 201
FIYR 199
FIYYQ 203
numeric 367
SLEEP 306
SQL 335, 347, 359, 367, 375
STRREP 108
variable length character 117

G

greenhouse functions 309
greenery functions

GIS_DISTANCE 314
GIS_DRIVE_ROUTE 316
GIS_GEOCODE_ADDR 318
GIS_GEOCODE_ADDR_CITY 319
GIS_GEOCODE_ADDR_POSTAL 321
GIS_GEOMETRY 322
GIS_IN_POLYGON 324
GIS_LINE 326
GIS_POINT 328
GIS_SERV_AREA_XY 332
GIS_SERVICE_AREA 330
GETENV function 297
GETTOK function 93
GETUSER function 303, 304
GIS_DISTANCE function 314
GIS_DRIVE_ROUTE function 316
GIS_GEOCODE_ADDR function 318
GIS_GEOCODE_ADDR_CITY function 319
GIS_GEOCODE_ADDR_POSTAL function 321
GIS_GEOMETRY function 322
GIS_IN_POLYGON function 324
GIS_LINE function 326
GIS_POINT function 328
GIS_SERV_AREA_XY function 332
GIS_SERVICE_AREA function 330

H

HADD function 223
hash value 139, 140
HCNVRT function 225
HDATE function 226
HDIFF function 226, 227
HDTTM function 228
HEX function 377
HEXBYT function 253, 254
HEXTYPE function 244
HGETC function 229, 230
HGETZ function 230
HHMMSS function 231
HHMS function 232
HINPUT function 233
HMIDNT function 234
HNAME function 235
holidays 167, 168, 170
holiday files 168, 170
HOUR function 351
HPART function 235, 236
HSETPT function 236
HTIME function 237, 238
HTMTOTS function 238
HYYWD function 239

I

IF function 378
IMOD function 269, 270
INT function 272, 364
INTEGER function 364
international date formats 183
ITONUM function 254
ITOPACK function 256
ITOZ function 256, 257

J

JOBNAME function 303
JULDAT function 215

K

KKFCUT function 135

L

lag values 58
LAST function 151, 152
LAST_NONBLANK function 66
LCASE function 339
LCWORD function 94, 96
LCWORD2 function 95, 96
LCWORD3 function 96
LEADZERO parameter 172
legacy date functions
   DATEFNS parameter 206
   DMY 211
   legacy versions 206
   MDY 211
   YMD 211
LENGTH function 379
linear regression equation 44
   FORECAST_LINEAR 44
LJUST function 97
LOCAS function
   variable length 118
LOCASE function 97, 98
LOG function 273, 371
LOOKUP function 152, 153
LOWER function 67, 339
LOWERCASE function 339
LPAD function 68
LTRIM function 69, 339
MICROSECOND function 352
MILLISECOND function 353
MIN function 273, 274
MINUTE function 353
MODIFY data source functions 151
MONTH function 354

N

NORMSDST function 274, 276
NORMSINV function 274, 276, 277
NULLIF operator 384
numbers 267
   absolute value 267
   calculating remainders 269
   generating random 277, 278
   greatest integer 272
   logarithms 273
   maximum 273
   minimum 273
   raising to a power 270
   square root 279
   standard normal deviation 274–276
   validating packed fields 268
numeric functions 267, 367
   ABS 267, 268
   CHKPCK 268
   DMOD 269, 270
   EXP 270, 271
   FMOD 269, 270
   IMOD 269, 270

M

MAX function 273
MD5 hash value 139
MDY function 211
numeric functions 267, 367
   INT 272
   LOG 273
   MAX 273
   MIN 273, 274
   NORMSDST 274, 276
   NORMSINV 274, 276, 277
   PRDNOR 277, 278
   PRDUNI 277
   RDNORM 278
   RDUNIF 278
   SQRT 279
numeric values 267

O
   OVRLAY function 98

P
   packed numbers, writing to an output file 261
   PARAG function 99–101
   PARTITION_AGGR 49
   PARTITION_REF 58
   PATTERN function 101
   PATTERNS function 70
   PCKOUT function 257, 258
   PHONETIC function 245
   POSIT function 102, 103
   POSITION function 70, 340
   POWER function 372
   PRDNOR function 277, 278
   PRDUNI function 277
   prior values 58
   process IDs 303
   PTOA function 258, 259
   PUTDDREC 305
   PUTENV function 297
Q
   QUARTER function 355
R
   RDNORM function 278
   RDUNI function 278
   REGEX function 71
   REPLACE function 72
   retrieving environment variable values 301
   REVERSE function 103
   RJUST function 104
   rolling calculations 49
   RPAD function 73
   RTRIM function 74, 341
S
   SECOND function 354
   SET parameters 167
      BUSDAYS 167
      DATEFNS 206
      DTDISTRICT 221
      HDAY 168, 170
      LEADZERO 172
SFTDEL function 136
SFTINS function 137
simple moving average 29
    FORECAST_MOVAVE 29
simplified character functions 63
simplified conversion functions 241
simplified date functions 155
simplified system functions 295
single-byte characters 125, 128
SLEEP function 306
SMALLINT function 364
SOUNDEX function 104, 105
SPELLNM function 105, 106
SPLIT function 75
SQL functions 335, 347, 359, 367, 375
SQL operators 381
SQRT function 279, 373
SQUEEZ function 106, 107
standard date and time functions 166
standard normal deviation 274–276
string replacement 108
STRIP function 107–109
STRREP function 108
SUBSTR function 109–111, 121, 341
    variable length 121
SUBSTRING function 76, 341
substrings 92
    extracting 92, 93, 109, 111, 121
    finding 102
    overlaying character strings 98
system functions 299
    FEXERR 300
    FGETENV 301
    FPUTENV 301, 302
    GETUSER 303, 304
    JOBNAME 303
    SYSVAR 307
    SYSVAR function 307
T
    TIME function 365
    TIMESTAMP function 366
    TO_INTEGER function 246
    TO_NUMBER function 246
    TODAY function 204
    TOKEN function 77
    TRIM function 111, 112, 343
    TRIM_ function 78
    TRIMV function 122
    triple exponential smoothing 40
      FORECAST_SEASONAL 40
    TSTOPACK function 259
U
    UCASE function 344
    UFMT function 260, 261
    UPCASE function 112, 113
    UPPER function 79, 344
    UPPERCASE function 344
    user IDs 303
V

VALUE function 379
values 149
decoding 149
verifying 151
variable length character functions 117

W

WEEKDAY function 356
WEEKFIRST parameter 219
work days 167
business days 167

work days 167
holidays 167, 168, 170

X

XMLDECOD function 113
XMLENCOD function 114
XTPACK function 261

Y

YEAR function 356
YM function 215
YMD function 211
Index
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.